AidOps: A Data-Driven Provisioning of High-Availability Services in Cloud

Diego Lugones Jordi Arjona Aroca Yue Jin Alessandra Sala Volker Hilt
Nokia Bell Labs
{first.last}@nokia-bell-labs.com

ABSTRACT

The virtualization of services with high-availability requirements calls to revisit traditional operation and provisioning processes. Providers are realizing services in software on virtual machines instead of using dedicated appliances to dynamically adjust service capacity to changing demands. Cloud orchestration systems control the number of service instances deployed to make sure each service has enough capacity to meet incoming workloads. However, determining the suitable build-out of a service is challenging as it takes time to install new instances and excessive re-configurations (i.e. scale in/out) can lead to decreased stability. In this paper we present AidOps, a cloud orchestration system that leverages machine learning and domain-specific knowledge to predict the traffic demand, optimizing service performance and cost. AidOps does not require a conservative provisioning of services to cover for the worst-case demand and significantly reduces operational costs while still fulfilling service quality expectations. We have evaluated our framework with real traffic using an enterprise application and a communication service in a private cloud. Our results show up to *4X* improvement in service performance indicators compared to existing orchestration systems. AidOps achieves up to 99.985% availability levels while reducing operational costs at least by 20%.

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*; *Availability*; • **Applied computing** → *Forecasting*;

KEYWORDS

Cloud orchestration, workload forecasting, high-availability

ACM Reference Format:
Diego Lugones Jordi Arjona Aroca Yue Jin Alessandra Sala Volker Hilt. 2017. AidOps: A Data-Driven Provisioning of High-Availability Services in Cloud. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 13 pages.
https://doi.org/10.1145/3127479.3129250

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3129250

1 INTRODUCTION

Virtualization of computing and networking resources is a key trend in industry. However, the adoption of virtualized mission-critical services is still an operational challenge and service operators continue to rely on expensive highly-available solutions and purpose-built appliances [48].

To accelerate adoption, operators must evolve from traditional upfront capacity planning and incorporate elasticity as a 'first class' property [51]. Commercial cloud providers [3, 17, 35] offer elasticity usually triggered by comparisons between resource usage and user-defined thresholds to instantiate or terminate virtual resources. Unfortunately, these solutions react *on-demand* to fluctuating loads rather than anticipating to them, putting performance and quality of service at risk [8, 26, 49].

Recent research has explored predictive elasticity. Some proposals leverage short-term predictions to provision service components that can be scaled within a few *minutes*, e.g. web servers and database instances [9, 36]. Instead, other proposals target longer term predictions to plan for a baseline capacity over e.g. a *few days* or even a week, and turn to on-demand elasticity to deal with traffic variations and prediction errors [21].

Applying these solutions to provision high-availability services has some limitations. For example, on-demand scaling using thresholds on individual resources can be insufficient to control end-to-end performance, particularly for non-trivial services that consist of several distributed components chained together to provide multiple functions [47]. Also, the delays for provisioning new instances can vary significantly from very few to dozens of minutes. Hence, short-term predictions may not suffice to cope with such variability and fail to provision capacity in time [50]. Long-term predictions alleviate this problem as forecasting one or more days of traffic allows for timely provisioning of services. However, when these predictions fail, orchestration systems switch to on-demand elasticity for capacity adjustments which can cause performance penalties. Accounting for provisioning delays and end-to-end performance is critical to satisfy quality expectations and service level objectives (SLO). Otherwise, frequent and uncontrolled scaling actions can increase SLO violations and impact availability [25, 53].

In figure 1, we plot the availability achieved with on-demand provisioning as well as short and long term predictions. The example service is a stateful high-availability media gateway composed of multiple *virtual network functions* (VNFs) across various service chains where scaling actions take time to propagate. Usually, cloud providers offer compensations when availability falls below 99.9% [34] – see line ⓐ. Telecom providers, in turn, have more stringent availability requirements for VNFs, e.g. more than 99.985% (line ⓒ). Observe that on-demand elasticity achieves insufficent availability for this service. According to [7], values lower than 99.8% can

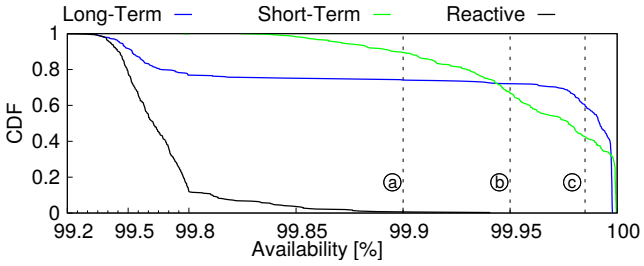

Figure 1: A virtualized media gateway as example of a high-availability service. Availability achieved using 1) reactive (on-demand) elasticity, 2) short-term predictions based on ARIMA models, and 3) long-term predictions based on frequency decomposition. Availability is measured as request completion rate per day over a one-year long dataset. Details of the service and techniques used can be found in § 6.

cause penalties up to 25% of the customer bill. Techniques based on short-term predictions improve availability, still penalties up to 10% may apply to values between 99.8% - 99.95% ⓑ [2, 18, 34]. Long-term predictions improve availability when predictions are accurate, more than 99.95%. However, when predictions fail these techniques react on-demand reducing availability in several days of the year.

Service providers are usually conservative and over-provision critical services as a remedy for SLO violations due to provisioning delays, inaccurate predictions or reconfiguration failures. Over-provisioning, however, is known to increase costs and resource inefficiencies, particularly in large systems.

We argue that including traffic models in the orchestration system allows for a higher availability and lower over-provisioning costs. Leveraging knowledge of recurrent patterns in workloads [15, 53] can dramatically reduce SLO violations while optimally fulfilling expectations of cost and service quality. In essence, our proposal is to identify demand patterns to improve predictions with more information about traffic seasonality, service-specific events and user behavior. This way, capacity management can be improved with a detailed utilization of service components to avoid the availability degradation shown in figure 1.

In this paper, we introduce *AidOps* a systematic approach for capacity provisioning of services in cloud. Rather than on-demand elasticity, AidOps enables service providers to specify a desirable number of reconfigurations (i.e. scaling in/out) as an *input*, then it calculates the capacity configurations required to serve the demand optimally. Applying dynamic programming over traffic models extracted from historical data, we optimize provisioning in terms of the number of reconfigurations, elasticity lead time and spare (margin) capacity. This enables a great flexibility in the provisioning, and allows providers to fine tune service stability and resource utilization while controlling costs. Our contributions are as follows:

- A method for dynamic capacity provisioning that leverages domain knowledge to assist cloud orchestration systems to streamline autonomic operations. This method enables service providers to move from worst-case provisioning towards a dynamic cost-effective solution while still achieving high availability.
- A clustering method to identify utilization patterns and model their statistical properties to learn traffic specific nuances and optimal capacity requirements.
- A classification algorithm that allows to accurately select traffic models and to quickly recognize seasonal patterns and workload changes. These changes can be either induced by providers themselves such as limited-time promotions or bulk provisioning of users, but also by external events affecting the service, e.g. increasing content or service popularity [10].
- An online controller to adjust prediction accuracy by detecting deviations between real and predicted traffic. This allows to quickly adapt the configuration in presence of unpredictable events. If deviation occurs, we use autoregressive models to adjust capacity on the fly.

The rest of the paper is organized as follows: Section 2 introduces the AidOps architecture. Traffic modeling and metadata-based classification are detailed in § 3. Section 4 elaborates on pattern-based prediction and methods to enforce the correct service configuration. Online control and prediction adjustments are described in § 5. We have implemented AidOps and evaluated its benefits using production-derived traffic in two commercial communication services running on virtual machines and containers, results are shown in § 6. We discuss related work in § 7, and conclude in § 8.

2 OVERVIEW

Figure 2 illustrates the AidOps components and interfaces to the cloud infrastructure. The system is composed of a frontend for configuration purposes, and a backend running analytics and capacity provisioning algorithms. The historical traffic data (D) is adapted to the proper time series format and input to the backend, where it is processed using distance-based hierarchical clustering to extract workload patterns (P). These patterns can be discovered offline and updated periodically – currently we perform the clustering once a day but it can be quickly configured to work with different time frames according to operator expectations and available data. The resulting clusters are classified using multinomial logistic regression to find relevant statistical characteristics of the patterns and, also, complementary features such as external events and calendar related information. As a result, data is classified with features (metadata) that improve the runtime predictions. Both clustering and classification algorithms are explained in § 3.

AidOps predicts daily patterns using the clusters and classification metadata such as day of the week (DoW), date, likelihood of external events, etc. The predicted pattern is used to *optimize* a configuration plan that programs the service orchestrator to change capacity at specific times of the day. Service operators can customize the optimization in terms of maximum number of reconfigurations ($\#_R$), spare capacity (S_C) and lead time (L_T). We use integer linear programming (ILP) with $\#_R$, S_C and L_T as constraints to create a provisioning plan encompassing the predicted demand, see § 4.2 for more details. Once the provisioning plan is in place, AidOps calculates the necessary number of instances using a service model (§ 4.3) and delivers the plan to the orchestrator.

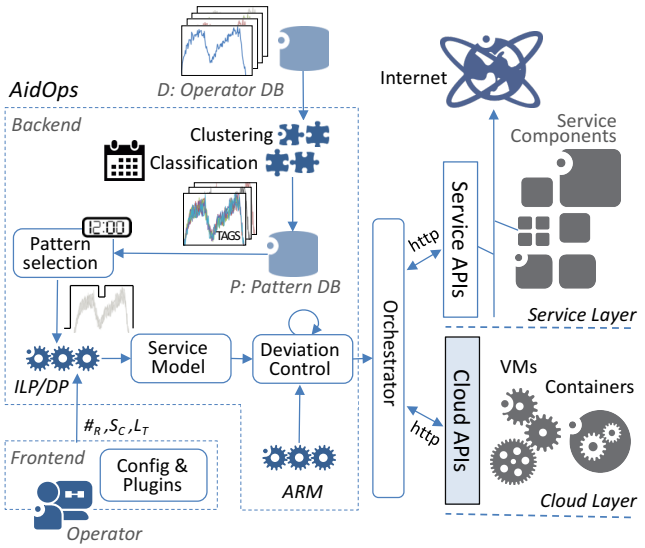

Figure 2: *AidOps* architecture.

The accuracy of predictions is evaluated at run time by analyzing deviations between current and predicted traffic. If deviation occurs (i.e. the prediction fails), AidOps adjust capacity automatically using autoregressive models (AR) as described in § 5. We have created a monitoring API based on Prometheus [42] that provides traffic and system metrics to measure traffic deviation and service performance. Similarly, we have developed a control API which provides plugin functionality to VM/container-based orchestrators. Currently, we support Rancher [43] and OpenStack [38] platforms.

Patterns are stored in a database (P) and are used by the backend to dynamically provision capacity, a simplified pseudocode is shown in *algorithm 1*. The configuration plan is created at regular intervals (e.g. midnight) and adapted dynamically if necessary. Initially, AidOps predicts the traffic demand of the current day (d_K) using the classified patterns (see *algorithm 1*: lines 1 to 3) and schedules the orchestrator to enforce the configuration plan (line 17). As the day evolves, the traffic is monitored to test prediction accuracy (lines 4 to 7). If the actual traffic differs from the predicted one, we perform a second pattern search including the traffic observed so far (line 8). This allows to better discriminate among patterns with similar likelihood as well as detecting and planning for less frequent workloads. If an alternative pattern ($\overline{P}$) is found, the configuration plan is recalculated using S_C, L_T and the number of reconfigurations left for the day (i.e. $\overline{\#_R} \leq \#_R$), see line 9. However, if no alternative pattern is found, AidOps considers the workload as unexpected (line 10) and triggers an emergency mode to adjust capacity. This mode quickly reacts by performing short-time predictions using autoregressive models (lines 11 to 16).

In the following sections, we detail our contributions for realizing dynamic capacity provisioning based on domain-specific patterns. To improve readability of sections, we provide an appendix with a glossary of mathematical definitions and notation in § 9.

Algorithm 1 Dynamic capacity provisioning

Input: $S_C, \#_R, L_T, P$

1: status ← normal
2: p ← *get_pattern*(d_K, P) ▷ see §3.1, 3.2, 4.1
3: cfg ← *capacity_plan*$(p, S_C, \#_R, L_T)$ ▷ see §4.2
4: **while** ! EoD **do**
5: w ← *traffic_mon()*
6: **if** status = normal **then**
7: **if** *deviation*(p, w) **then** ▷ see §3.3, 5.1
8: $\overline{p}$ ← *get_pattern*(w, P)
9: **if** $\overline{p}$ **then**
10: cfg ← *capacity_plan*$(\overline{p}, S_C, \overline{\#_R}, L_T)$
11: **else:** status ← abnormal
12: **else**
13: $\widetilde{w}$ ← *autoregression*(w, L_t)
14: cfg ← *adjust_capacity*$(\widetilde{w}, S_C)$ ▷ see §5.2
15: **if** ! *deviation*$(\{p|\overline{p}\}, w)$ **then**
16: status ← normal
17: cfg ← *capacity_plan*$(\{p|\overline{p}\}, S_C, \overline{\#_R}, L_T)$
18: **if** cfg.has_changed() **then:** *provision(cfg)* ▷ see §4.3
19: *update_DB*$(d_K \rightarrow D, P)$ ▷ see §4.4

3 TRAFFIC ANALYSIS

This section describes the clustering and classification algorithms for the creation of traffic models. We use historical data from services grouped by days, this data is denoted as $D = \{d_0, \ldots, d_{|D|}\}$. Each day d_i has a set of features (or tags) $F_i = \{f_0, f_1, \ldots\}$ that can be related to the date, such as the weekday (DoW) or month; can be calendar events or holidays such as Christmas, or operator defined events that span from promotions in particular periods to changes in the subscriber base. Each day d_i is described by a time series $S_i = \{T, V\} = \{(t_0, v_0), (t_1, v_1), \ldots, (t_{|T|}, v_{|T|})\}$ with traffic volume V at time slots T. Thus, $D = \{(F_0, S_0), \ldots, (F_{|D|}, S_{|D|})\}$ as illustrated table 1. For simplicity, we use the same T across all days d_i. The interval between time slots T depends on the nature of the service. For example, the service demand in data communication services is not as bursty as it is for web browsing, and session lengths are in the order of minutes or larger (e.g. the average mobile call time in the US was ~ 2 minutes in 2012). A small time interval T, compared to the serving time of requests, increases the amount of data without adding information to the algorithms. That is, the processing time increases without improving the results. Therefore, we configure T in the order of the request lifespan. This allows for capturing changes in the workload, as well as processing data more efficiently. A suitable time interval for the evaluation of section 6 is 1 minute, meaning there are $|T|$=1440 time slots per day.

3.1 Clustering Algorithm

We use *hierarchical clustering* to detect traffic patterns. This technique provides more consistent results than other known methods like, for instance, *k-means* that may yield different results depending on the selection of initial centroids and requires previous knowledge about the number of clusters, which is not available in our case. *Algorithm 2* describes the procedure. We construct the hierarchical tree bottom-up. That is, each day is initially a singleton cluster and

	Features F			$S = \{T, V\}$
Date	DoW	Holiday	...	Data traffic time series < *time*, *magnitude* >
2015-11-01	Friday	All Saint's Day	...	[(1, 10006.07), (2, 5037.21), ... , (3, 6940.56), ...]
2015-11-02	Saturday	None	...	[(1, 8708.31), (2, 4419.21), ... , (3, 6293.28), ...]
...				

Table 1: Raw data with features/tags followed by traffic time series

we successively merge them by pairs, according to their "distance", until all clusters have been merged into one cluster. The *distance* between two days i and i' is the *2-norm* of the traffic volume differences over all time slots: $dist(i, i') = \|V_i - V_{i'}\|$ (lines 1 to 4). We use the complete linkage criterion (also known as farthest neighbor) to separate clusters based on such distance. This method is known to find compact clusters of similar diameters and, therefore, to group traffic with similar patterns.

We find the clusters by cutting the hierarchical tree iteratively. At each cut, we compute the coefficients of variation $CV(j, t)$ and their median $CVM(k, j)$ (lines 5 to 11). We need the CVM to decide whether to stop the cut or not. If $CVM(k, j)$ is lower than a value φ, then the variance of the pattern is well defined. So, we iterate over $k = 2, 3, \ldots, |D|$ until we find the smallest k such that all the k clusters are within φ (lines 12 to 16). We use $\varphi = S_C/2.33$, where S_C is the spare capacity parameter, and 2.33 is the factor of standard deviations that covers 99% of the density in a normal distribution. This approximation is accurate with sufficient data and allows to generalize the clustering process.

Last, we create a pattern that models the traffic in each cluster c_j. The pattern contains the average traffic (lines 17 to 19), and the standard deviations (line 20). The clustering extends the set of features

Algorithm 2 Clustering and pattern detection

Input: $D = \{(F_0, S_0), \ldots, (F_{|D|}, S_{|D|})\}$
1: **for** $i \leftarrow 1, |D|$ **do**
2: **for** $i' \leftarrow i + 1, |D|$ **do**
3: $dist(i, i') \leftarrow \|V_i - V_{i'}\|$
4: Construct the hierarchical tree with $dist(i, i')$
5: **for** $k \leftarrow 2, |D|$ **do**
6: Cut the hierarchical tree into k clusters
7: **for** $j \leftarrow 1, k$ **do**
8: **if** $|c_j| > 1$ **then**
9: **for** $t \leftarrow 1, |T|$ **do**
10: $\text{CV}(j, t) \leftarrow \text{Std}(v_t|V_i)/\text{Mean}(v_t|V_i), \forall i \in c_j$
11: $\text{CVM}(k, j) \leftarrow \text{Median}(\text{CV}(j, t)) \, , \forall t \in [1, T]$
12: $k^* \leftarrow |D|$
13: **for** $k \leftarrow 2, |D|$ **do**
14: **if** $\text{CVM}(k, j) < \varphi$ for all $j \in [1, k]$ **then**
15: $k^* \leftarrow k$
16: break
17: **for** $j \leftarrow 1, k^*$ **do**
18: **for** $t \leftarrow 1, T$ **do**
19: $\hat{v}_t|c_j \leftarrow \text{Mean}(v_t|V_i)$ over $i \in$ cluster c_j
20: $\lambda_t|c_j \leftarrow \text{Std}(v_t|V_i)$ over $i \in$ cluster c_j

with a cluster identifier I_j and the number of days θ_j within the cluster c_j. This is illustrated in table 2 with $\hat{F}_j = \{I_j, \theta_j\}$, and the resulting pattern: P_j={T, $\hat{V}$, Λ}={(t_0, $\hat{v}_0$, λ_0), (t_1, $\hat{v}_1$, λ_1), .., ($t_{|T|}$, $\hat{v}_{|T|}$, $\lambda_{|T|}$)} where $\hat{V}$ and Λ are the average traffic volume the corresponding standard deviation at each time T.

The computational complexity of the algorithm is $O(|D|^2 \cdot |T| + |D|^2 \log(|D|))$. The first term corresponds to the calculation of distances, whereas the second is the complexity of constructing the hierarchical tree. Other tasks in the algorithm have similar or less complexity.

After clustering, the number of days within each cluster will likely differ. Although, a high number of days within a cluster is desirable for statistical significance, we also want to quickly detect newly emerging traffic patterns. So, we define the clusters with a large number of days as *consolidated* (C) whereas clusters with a low number of days are *not consolidated* ($\not C$). The distinction is done by comparing to a parameter θ whose value can be generalized to 30 by using the central limit theorem, but can be adjusted to a lower value depending on data properties – see §6.

3.2 Data Classification

We use the features of days to improve the selection of patterns. For this purpose, we have built a classification scheme based on a multinomial logistic regression model. This approach allows to generalize the model to multi-class problems with low variance and avoiding over-fitting. We focus on consolidated clusters, and define the largest cluster as the reference group j^0. Our model is $\log(p_j/p_{j^0}) = \sum_l \alpha_{j,l} f_l + \alpha_{j,0}$. Where p_j is the probability of a day belonging to cluster c_j. f_l is the binary value of the feature. For instance, if 'DoW' is 'Monday', $f_{\text{'Monday'}} = 1$ and it is equal to 0 for other values of 'DoW'. Furthermore, $\alpha_{j,l}$ are the coefficients that we learn through training, and $\alpha_{j,0}$ is the constant of the regression model. Each value of a feature is represented by a coefficient $\alpha_{j,l}$. For instance, there are 7 $\alpha_{j,l}$ for the values of DoW: Monday, Tuesday, etc. We denote the vector with the coefficients for the multinomial logistic regression model as $\Sigma_j = \{\alpha_{j,0}, \alpha_{j,\text{'Monday'}}, \alpha_{j,\text{'Tuesday'}}, \ldots\}$.

As a result, this classification expands the features $\hat{F}$ in the clusters as follows: $\hat{F}_j = \{I_j, \theta_j, \Sigma_j\}$. Table 3 provides an example. In terms of computational complexity, fitting a logistic regression model involves matrix inversion with a complexity of $O(|\Sigma|^3)$, and a matrix multiplication with complexity $O(|\Sigma| \cdot |D|)$. Since we have $|C|$ clusters, the overall complexity is $O(|C|(|\Sigma|^3 + |\Sigma| \cdot |D|))$.

Augmented Features $\hat{F}$			$P = \{T, \hat{V}, \Lambda\}$
Cluster ID	Number of Days	...	Traffic pattern time series $< time, average, stddev >$
1	36	...	[(1, 791.19, 99.21), (2, 41.2, 246.84), ..., (3, 2.77, 93.92), ...]
2	10	...	[(1, 720.44, 58.33), (2, 36.6, 970.33), ..., (3, 9.09, 73.96), ...]
...			

Table 2: Traffic patterns extracted from clusters

Augmented Features $\hat{F}$						P
Cluster ID	Number of days	$\alpha_{j,l}$ for 'DoW' feature	$\alpha_{j,l}$ for 'Holiday' feature	...	$\alpha_{j,0}$	Traffic pattern
1	36	[0.41, 0.38, 0.29, 0.18, ...]	[0.62, 0.26, 0.46, 0.29, ...]	...	0.35	(T, V_j, Λ_j)
2	10	[0.13, 0.62, 0.01, 0.13, ...]	[0.38, 0.68, 0.05, 0.25, ...]	...	0.43	...
...						

Table 3: Cluster data extended with the coefficients of the classification model

3.3 Prediction Accuracy Model

AidOps continuously monitors incoming traffic to detect deviations from the predicted pattern. We leverage existing data to construct a logistic regression model that is trained to determine whether a given traffic time series $\tilde{S} = \{\tilde{T}, \tilde{V}\}$ belongs to a consolidated cluster c_j. In other words, this model determines whether the current traffic fits in the expected pattern. *Algorithm 3* describes the training process. We consider the days $d_i \notin c_j$ as 'atypical' and those $d_i \in c_j$ as 'normal'. We sample randomly among the 'normal' and 'atypical' days (line 2) and label them accordingly (lines 3 to 6). We subtract $\tilde{V}_j$ from the traffic values in the training data and keep the absolute values of the deviation (line 7). Our logistic regression model is $\text{logit}(p) = \sum_{l=0}^{L-1} \delta_{j,l} r_{t-l} + \delta_{j,L}$. Here p is the probability of a day being atypical, r_{t-l} is the absolute difference of traffic at time $t-l$, whereas $\delta_{j,l}$ and $\delta_{j,L}$ are the coefficients to be learned with training (line 8). Note that the training requires to know the time window of size L that will be used to detect deviations in runtime. Values of $L \sim 5...10$ time slots allow to quickly detect anomalies while minimizing the detection of false positives. Larger values of L do not improve accuracy, and delay the detection.

After learning the coefficients of the detection model, denoted as $\Delta_j = \{\delta_{j,0}, \delta_{j,1}, \ldots, \delta_{j,L}\}$, we expand for a last time the features $\hat{F}_j$. Therefore, the final full-featured form of a cluster c_j is $\hat{F}_j = \{I_j, \theta_j, \Sigma_j, \Delta_j\}$. Similarly to the data classification of section 3.2, the computational complexity is affected by a matrix inversion, $O(L^3)$, and a matrix multiplication, $O(L \cdot |D_s| \cdot |T|)$, where $|D_s|$ is the number of days in the training data. Therefore, the overall complexity for training for one cluster is $O(L^3 + L \cdot |D_s| \cdot |T|)$.

Algorithm 3 Deviation model - training phase

Input: $D = \{(F_0, S_0, c_0), \ldots, (F_{|D|}, S_{|D|}, c_{|D|})\}$

1: **for** $j \leftarrow 1, |C|$ **do**
2: Randomly sample days from D, inside and outside of c_j, and form the training set D_s
3: **for** $i \leftarrow 1, |D_s|$ **do**
4: **for** $t \leftarrow 1, |T|$ **do**
5: **if** $c_i = c_j$ **then:** $p_{i,t} \leftarrow 0$
6: **else:** $p_{i,t} \leftarrow 1$
7: $r_{i,t} \leftarrow |(v_t|V_i) - (\tilde{v}_t|\tilde{V}_j)|$
8: Learn $\delta_{j,l}$ ($l \in [0, L]$), with $r_{i,t}$ and $p_{i,t}$ ($i \in [0, |D|]$, $t \in [0, |T|]$)

4 PATTERN BASED OPERATION

We use the models created in § 3 to identify a set of possible traffic patterns that are likely to occur during a day. Patterns are retrieved using metadata queries to access the classification model, and sorted by probability. Then, we conduct an integer linear programming (ILP) optimization over the predicted pattern to create a capacity plan that optimally serves the demand.

4.1 Cluster Selection

Cluster selection is performed according to operator's schedule (e.g. midnight) by processing the date to obtain the features F_i for the current day d_i. F_i is composed of at least three types of features or tags: calendar-based (e.g. holidays), operator-defined (e.g. promotions), and external events (e.g. a new release of popular applications or updates). These features are retrieved by simple date queries to a database storing a metadata-annotated calendar. Features of d_i are the input of the classification model described in Section 3.2. As a result, we retrieve an array with values p_j indicating the probability of day d_i to belong to each cluster c_j. We use the dominant cluster to perform the optimization, i.e. the cluster with a probability significantly higher than all the others. If there is not a dominant cluster, the final selection combines conservatively those patterns with higher probability by taking the maximum traffic value at each point in time.

4.2 Capacity Plan

We use the predicted traffic pattern to optimize provisioning in a daily basis. Our objective is to minimize capacity during each day, such that all the demand is covered without exceeding a given number $\#_R$ of reconfigurations (i.e. scale in/out operations). We formulate the problem using ILP and develop a dynamic programming algorithm to achieve optimality in polynomial time.

Problem modeling. We consider the capacity control problem over the time horizon $|T|$. The demand over the time horizon is the predicted pattern $\hat{V}_c$ of cluster c (see §4.1). We allow at most $\#_R$ reconfigurations. Each reconfiguration can change capacity, while in-between two consecutive reconfigurations the capacity levels stay the same. In a graphical way, this would be an horizontal line with height equal to the capacity level. A feasible solution is then composed of several horizontal lines above the demand curve where the next line starts at where the previous line ends, figure 3 provides an example. The height of each line is determined by the demand between two consecutive reconfigurations.

Suppose that consecutive reconfigurations take place at i and $j+1$, then the total capacity needed between reconfigurations at i and $j+1$ is given by:

$$cost_{ij} = (j - i + 1) \cdot \max_{t \in [i,j]} (\hat{v}_t | \hat{V}_c)$$

To decide which lines to include in the optimal solution, we define variables x_{ij} where $x_{ij} = 1$ if there is a line from i to j (i.e. there are reconfigurations at i and $j+1$) or $x_{ij} = 0$ otherwise. Our formulation is:

$$\min \quad \sum_{(i,j):i \leq j} cost_{ij} \cdot x_{ij} \tag{1}$$

$$s.t. \quad \sum_{(i,j):i \leq j} x_{ij} \leq \#_R + 1 \tag{2}$$

$$\sum_{j \geq 1} x_{1j} = 1 \tag{3}$$

$$\sum_{i \leq T} x_{iT} = 1 \tag{4}$$

$$\sum_{i \leq t} x_{i,t} = \sum_{j > t} x_{t+1,j}, \forall t \in (0, T) \tag{5}$$

$$\sum_{i \leq t} x_{i,t} \leq 1, \forall t \in (0, T) \tag{6}$$

$$\sum_{j \geq t} x_{t,j} \leq 1, \forall t \in (0, T) \tag{7}$$

Where, constraint 2 allows at most $\#_R$ reconfigurations ($\#_R + 1$ lines). Constraints 3 and 4 ensure that the lines cover the first and last time periods. Constraint 5 ensures the next line starts at where the previous line ends. Finally, constraints 6 and 7 guarantee one endpoint per line in each period.

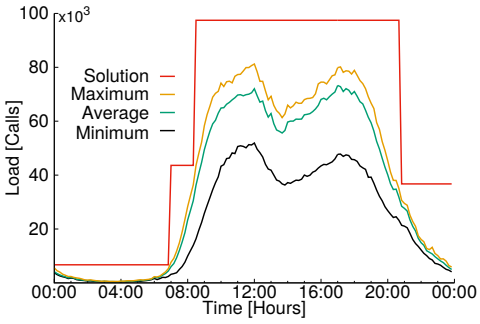

Figure 3: Pattern and envelope capacity (optimization)

Algorithm 4 Reconfiguration plan

Input: $\hat{V}_c$: the daily pattern of cluster c
Input: $\#_R$: the number of reconfigurations
1: **for** $i \leftarrow 1, |T|$ **do**
2: **for** $j \leftarrow i, |T|$ **do**
3: $cost_{ij} \leftarrow (j - i + 1) \cdot \max_{t \in [i,j]} (\hat{v}_t | \hat{V}_c)$
4: **for** $j \leftarrow 0, \#_R + 1$ **do**
5: $A_{|T|,j} \leftarrow 0$
6: **for** $i \leftarrow |T| - 1, 0$ **do**
7: **for** $k \leftarrow 0, \#_R + 1$ **do**
8: $A_{i,k} \leftarrow \min_j \{cost_{ij} + A_{j+1,k-1}\}$

Dynamic programming algorithm. The problem formulated in § 4.2 is well suited for dynamic programming. *Algorithm 4* describes the procedure. The problem is cast as finding the shortest path between "nodes". The nodes are the time slots, and the arch's length between two nodes i and j is the total capacity $cost_{ij}$ to cover the demand between these nodes (line 1 to 3). The optimal solution is the shortest path between the first and last nodes that is composed of $\#_R + 1$ arches. We start from the end of the time horizon and conduct a backward recursion. We define $A_{i,k}$ as the minimum capacity from node i to the end of the time horizon with k arches. The boundary conditions are $A_{T,k} = 0$ for $0 \leq k \leq \#_R + 1$ (lines 4 and 5). The backward recursion is $A_{i,k} = \min_j \{cost_{ij} + A_{j+1,k-1}\}$ (line 6 to 8), and the objective is $A_{0,\#_R+1}$. The solution is finally adjusted by adding the spare capacity S_C and lead time L_T. The algorithm complexity is $O(|T|^2 \cdot \#_R)$.

4.3 Service Models

AidOps produces a configuration plan to provide for the service capacity needs throughout the day. In order to enforce such plan, we also need to convert the capacity prediction to the actual amount of virtual resources. To this purpose, we use empirical service models – created offline by service providers using benchmarking – to dimension resources for different workload ranges and state information. The models contain 1) the number and size of virtual instances (VMs or containers) required for different magnitudes and types of workloads, 2) the service chains as lists of VMs or containers – used to scale service components and instances concurrently, and 3) the recipes for provisioning such components once the resources are available, e.g. Ansible playbooks [4]. More sophisticated models can be used nonetheless, but this is beyond the scope of the paper. In § 7 we provide a taxonomy of alternatives that can be used to model virtualized services.

4.4 Cluster Update

At the end of a day, AidOps updates the database with the time series of day d_i and recomputes the clustering as well as the coefficients (Σ and Δ). The time taken to perform these operations depends on the data size. We have indicated the computational complexity of each algorithm sections § 3.1,§ 3.2 and §3.3. Table 4 shows actual time measurements of these operations for different sizes of stored data. Observe that processing 5 years of data takes ~12 minutes. Although the computation is fast, it also is possible to schedule

Algorithm 5 Deviation model - online detection

Input: Incoming traffic $\tilde{S}$ and traffic pattern $\hat{V}_c$ of cluster c
1: **if** cluster c is consolidated **then**
2: Retrieve Δ of the expected cluster c
3: **for** $j \leftarrow 0, L-1$ **do**
4: $r_{t-j} \leftarrow |v_{t-j} - (\hat{v}_{t-j}|\hat{V}_c)|$
5: Compute: logit $(p_t) = \sum_{j=0}^{W-1} \delta_j r_{t-j} + \delta_L$
6: **else**
7: **for** $j \leftarrow 0, L-1$ **do**
8: $r_{t-j} \leftarrow v_{t-j}/(\hat{v}_{t-j}|\hat{V}_c)$
9: $r^m \leftarrow \max_{0 \le j \le L-1}\{r_{t-j}\}$

the clustering and re-computation of Δ and Σ less often (e.g. in a weekly basis) as this calculation is not critical in normal periods with known patterns.

5 REACTING TO THE UNEXPECTED

AidOps constantly monitors the deviation between predicted and actual traffic using the deviation models explained in § 3.3. These models allow to identify two types of anomalies that can be treated differently: unpredictable events and infrequent events. With the latter type, we can search for similar patterns that have not been observed enough times (i.e. not consolidated clusters, §3.1). However, if traffic is unpredictable we use a short term prediction method to adaptively control the capacity.

5.1 Deviation Detection

Algorithm 5 describes the procedure to detect deviations. When the predicted traffic pattern is from a consolidated cluster (line 1), we apply the logistic regression model described in section 3.3. We compute the absolute value of the difference r_t between the incoming and the predicted traffic over the time window L (lines 3 and 4) and insert the differences into our logistic regression model to determine the probability of the incoming traffic to be within the expected pattern (line 5). However, if the predicted traffic pattern is from an unconsolidated cluster, we compute the magnitude ratio r_t between the incoming traffic and the expected pattern over the time window L (lines 7 and 8). Then, we find the maximum value of the ratio r_t (line 9) that indicates how different the incoming traffic is from the pattern, this is essentially a shape comparison because not consolidated clusters have not enough information to create accurate statistical models.

Volume	Clustering	Σ	Δ
1 year	28.16 s	1.52 s	0.09 s
3 years	247.91 s	3.50 s	0.31 s
5 years	740.13 s	5.64 s	0.75 s
10 years	3122.86 s	10.50 s	0.97 s

Table 4: Running times for clustering and computation of Δ, Σ coefficients for various volumes of data.

Algorithm 6 Online adaptation

Input: $\hat{F}$: Clusters' features
1: **for** $f \in F_i$ **do**
2: **if** $F_i[f] \in \hat{F}$ **then** remove f from F_i.
3: **if** $F_i == \emptyset$ **then**
4: autoregression $\leftarrow x_{t+k}$
5: **else**
6: $\mathcal{C} \leftarrow c_j \; \forall \; c_j \mid F_j \cap \hat{F} \neq 0$
7: $\overline{P} = combine(c_i \in \mathcal{C})$
8: return $\overline{P}$

5.2 Adaptive Provisioning

If deviation is detected, AidOps attempts to differentiate the anomaly using the features of the day. We use *algorithm 6* to analyze whether the current day has new features F_i that are not present in $\hat{F}_j$ of cluster c_j – see § 3.3. A feature is discarded if found (line 2). If there is any remaining feature, we query our database to retrieve the not consolidated clusters $\mathcal{C}$ having matching features (lines 5 and 6). Again, if more than one cluster is found we combine them into a final expected pattern $\overline{P}$ (line 7 and 8) and apply the optimization described in § 4.2.

However, if all features are discarded (line 3), there is not certainty about the workload. So, AidOps switches to short term predictions to provision capacity for the next timeslot (line 4) as fallback. Note that, in this case, the workload has not been previously seen by the controller. Therefore, we need to minimize the effect of past traffic on the forecast, as the influence of underlying trends or seasonality would be counterproductive and may reduce the responsiveness of predictions. For this reason, we discard methods that consider seasonality of data. Instead, we use an AutoRegressive (AR) model that captures short-term trends and provides high responsiveness, required in this case [33]. AR prediction is given by: $x_{t+k} = c + \varepsilon_t + \sum_{i:1}^{n} a_i . x_{t+k-i} \; , \; \forall k : 0, ..., L_T - 1$
Where n and a_i are the order and coefficients of the model, while c and ε_t are an adjustable constant and the noise distribution; k indicates the time slots to predict.

6 EVALUATION

We have evaluated AidOps and compared it to other state of the art solutions considering different use cases. The evaluation is done on commercial communication products, as examples of high-availability services. The experiments are based on production traces from two different scenarios:

1) A corporate collaborative service. We use a container based collaborative software in a medium-sized enterprise. This service provides different types of messaging ranging from human chat-based interactions to machine to machine communications. It is composed of several dozens of containers, and the workload is measured in chats where multiple users exchange diverse data and file types. The dataset shows patterns of working days, holidays (vacation), bank holidays, *project deadlines*[1], etc.

[1]Our data corresponds to an enterprise environment with well defined project deliverables and deadlines particularly before holiday periods

2) A session border controller in a European city. We use a virtualized SBC, a key component of the IP Multimedia System (IMS) which provides convergence of voice, data and multimedia for different access technologies. This dataset has a rich set of patterns for working days, weekends, holidays, and other events that influence the workload – e.g. service promotions and external events. The SBC contains signaling and data planes with a few dozens of virtual machines used by multiple service chains.

Experimental setup. Our experiments run in two different development clusters, one composed of HP DL380p servers with 48-cores Intel Xeon E5-2600 processors whereas the other has HP DL385p servers with 32-core AMD Opteron 6300 processors. Both clusters run CentOS 7 with OpenStack Kilo and are connected by 10 Gbps switches.

Metrics. We consider performance and cost metrics in the evaluation. Related to performance, we measure the availability as the ratio between arriving and served traffic during each day and show it as a cumulative distribution function (CDF). We also measure the SLO fulfillment achieved with each technique. The SLO of the collaborative service is to keep the latency below 3 seconds. In the SBC scenario, the SLOs are to keep rejected calls below 0.3%, and packet drops below 1%.

Cost is measured as the sum of the cost of provisioning resources (C_P) and the cost of SLO violations (C_S):

$$Cost = C_P + C_S \ where, \tag{8}$$

$$-C_P = k_1 . \sum_{i:0}^{T} c_i$$

$$-C_S = k_2 . \sum_{i:0}^{T} (t_i - c_i), \forall t_i > c_i$$

C_P is the cost of the virtual resources provisioned to serve the demand. It is the accumulated capacity c_i (measured as the requests that a service can handle with such resources) multiplied by a constant k_1 which is essentially the cost per request.

C_S is the cost associated to the *lack* of virtual resources due to inaccurate predictions, provisioning delays or reconfiguration failures. It accounts for all the events in which the capacity is below the actual traffic ($t_i - c_i$). The difference between traffic and capacity is also measured as service requests and multiplied by a constant k_2 that indicates the cost of SLO violation per request.

Values of k_1 and k_2 will be discussed later in this section. Note that calculations of C_P and C_S do not account for all capital and operational costs, but just focus on how cost is affected by predictions in terms of over and under utilization. For comparison purposes, we provide cost figures in relation to the *ideal* cost. That is, $Cost/C_I$ where:

$$C_I = k_1 . \sum_{i:0}^{T} t_i \tag{9}$$

C_I is the (ideal) cost of the *exact* amount of virtual resources required to serve the traffic at each time.

Techniques. For comparison purposes we use 1) a solution based on the fast Fourier transform *(FFT)* – similar to the technology behind Scryer [22], and 2) a predictor based on *ARIMA* models, resembling novel proposals such as [44, 55]. Note that we have implemented the core functions for a qualitative comparison rather than the exact prediction techniques in the referenced articles. Essentially, FFT leverages frequency decomposition to identify most dominant traffic patterns. Our implementation works with weekly periodicity and its model considers traffic windows of 2 and 3 weeks for the collaborative service and SBC, respectively. These windows are a good compromise, as shorter periods give high relevance to outliers while longer periods slow down the adaptation to seasonal variations. Similarly, we have implemented an ARIMA(p,d,q) predictor where d is the degree of differencing, and p and q are the orders of the autoregressive and moving average terms, respectively [55]. We fix $d = 2$ to provide statistical stationarity and compute p and q online to maximize the accuracy in presence of traffic variations. Predictions are performed from 5 to 10 minutes ahead depending on the experiment.

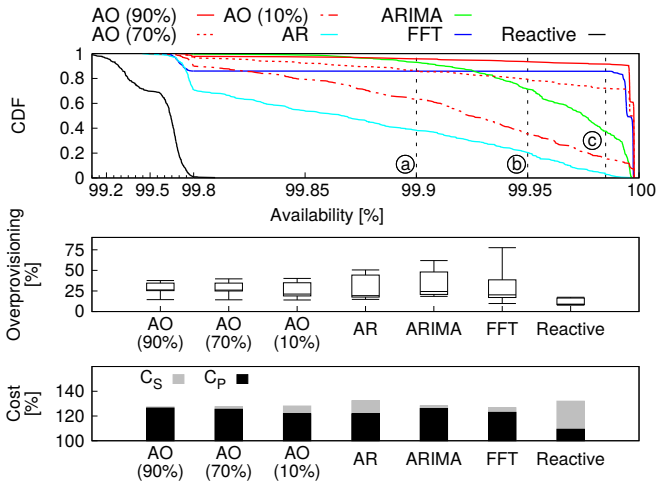

Figure 4: Performance and cost comparison between ARIMA, FFT and AO (with prediction accuracy of 90%, 70%, and 10%). Availability SLOs are indicated for reference: ⓐ 99.9%, ⓑ 99.95% and ⓒ 99.985%. AidOps outperforms FFT and ARIMA, meeting requirements ⓐ and ⓒ in 95.6% and 91.5% of the days. AidOps improves long term predictions by successfully predicting infrequent patterns. ARIMA adapts to varying traffic but fails to provision the service upon surges. The collaborative service has not stringent availability requirements, hence violation penalties are low. The cost is not affected, being actually similar to ARIMA's and FFT's.

6.1 Use case 1. A collaborative service

Figure 4 shows the availability, over-provisioning and cost of AidOps (AO), ARIMA, and FFT over one year. We also show the results of online scaling (i.e. Reactive) as a baseline. In order to stress AidOps and understand its robustness to prediction inaccuracy, we have randomly introduced traffic variations that do not follow any pattern found in the data set. Therefore, AO(90%), AO(70%) and AO(10%) indicate that the experiment had 10, 30 and 90% of data randomized, respectively. Note that, for example, 30% of inaccurate predictions means that ~100 days in a year do not follow any previous pattern.

We also provide results for AidOps working only in AutoRegressive mode (AR), which indicates that not suitable patterns are ever found and the system works only with short-term predictions. That is, AR is equivalent to AO(0%). These extreme cases are very unfavourable for AidOps but useful to explore the limits of our solution. Note that our initial assumption is that there are well-defined patterns in the data. Therefore, we expect high prediction accuracy in a representative data set.

AO is configured with $\#_R \leq 15$, $L_T = 5min$ and $S_C = 10\%$. We also use FFT and ARIMA with spare capacity $S_C = 10\%$ for fairness. In section § 1, we introduced the availability SLOs offered by some cloud operators. These values are also highlighted in figure 4 – typical SLOs range from 99.9% ⓐ to 99.95% ⓑ, whereas telecom VNFs may require up to 99.985% ⓒ.

Results show that FFT can meet availability (A_v) SLOs ⓐ and ⓒ in 85.7% and 84.3% of the days, respectively. With an average of 99.937% throughout the year. During these days, FFT achieves very high A_v values. However, some weekly predictions fail around 15% of the days which leads to underprovisioning. This happens because some situations associated to a workload increase cannot be captured by the model; for instance, project deadlines at the example enterprise, or seasonal changes – e.g. end of the summer at September . Therefore, FFT ends up relying mostly on reactive (on-demand) policies that reduce availability as shown by the curved elbow around ⓒ.

ARIMA, in turn, can meet A_v SLOs ⓐ and ⓒ in 92.8% and 37.8% of the days, respectively. Season changes or unusual events do not affect ARIMA given the short-term nature of its predictions. However, sudden workload variations and traffic burstiness throughout a day can cause ARIMA to scale in/out chaotically and inaccurately. ARIMA does not learn from these errors and, therefore, repeats them throughout the year. This causes underprovisioning in many days, explaining the low availability around ⓒ. However, the provisioning errors are solved in the short-term, reducing the errors per day as indicated by the results around ⓐ, and by the average availability of 99.963% throughout the year.

Note that A_v values can be much higher when the workload follows patterns learnt by AidOps. As is the case for AO(90%) that achieves A_v SLO ⓐ and ⓒ in 95.6% and 91.5% of the days, respectively; reaching up to 99.985% throughout the year. In this case, AO(90%) reduces the workload affected by underprovisioning in 2.5X and 4X when compared to ARIMA and FFT, respectively. These results reveal that the metadata classification and pattern recognition allow to identify and adequately provision unusual isolated events like bank holidays and workload surges (caused by project deadlines in this scenario), as well as repetitive workload induced by seasonal changes.

The benefits of using AR as a fallback (short-term) predictor for AidOps can be seen in AO(70%), note that although prediction accuracy is only 70%, the A_v achieved is similar to FFT's even when accuracy of FFT is actually 15% higher. In addition, we show how AidOps is degraded more when more random traffic arrives. In the case of AO(10%) and AO(0%) (i.e. AR) there is little or none repeatability in the workload, indicating that traffic is almost completely randomized. In these extreme cases, ARIMA performs better than AR and could be used as the fall-back predictor. However, deciding whether ARIMA is more suitable to use when an anomaly

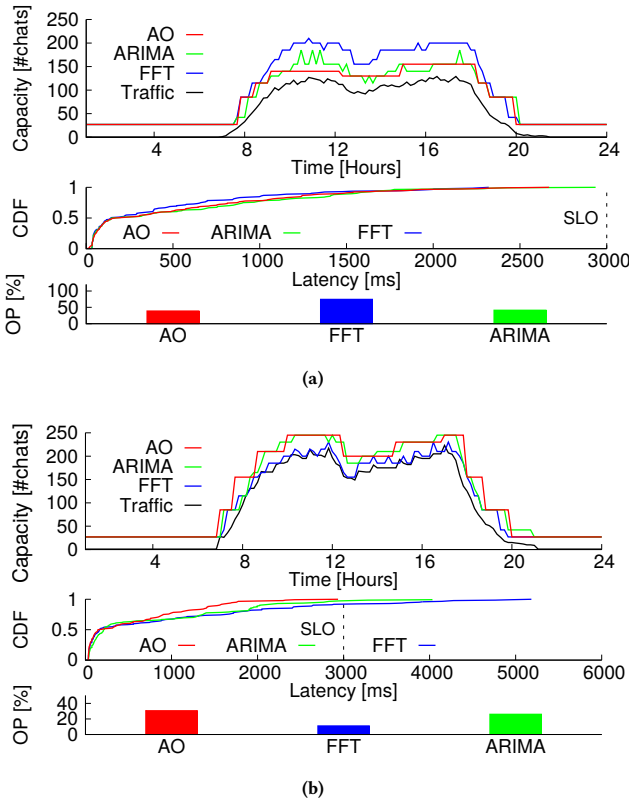

Figure 5: AidOps predicts seasonal changes using metadata analysis. ARIMA fails to provision the service upon fast or bursty traffic variations. (a) FFT incurs in large over-provisioning when transitioning to the holiday period as it model needs to adapt gradually to the lower traffic (June→ July). (b) When transitioning back to the regular working period (August→ September), FFT predictions fall short as it needs to adapt to higher traffics reactively. This results in more service reconfigurations and latency SLO violations.

is detected depends highly on the type of anomaly, the length of the data window used to create the AR or ARIMA model, and the actual data in the window. We opted for an AR model because it provides a rapid provisioning after traffic deviation. The moving average (MA) component of ARIMA can reduce the responsiveness of the controller upon the occurrence of a workload surge. That is, the data in the window previous to the anomaly could favor the selection of a model with a high MA order or even a pure MA model – which leads to a slower resource provisioning.

Results related to over-provisioning are shown in the box plots of figure 4 (middle), FFT reaches up to 20.3% in median, while ARIMA over-provisions 24.4%, and AO(90%) reaches ~ 26%. Although slightly higher, AO's over-provisioning is less variable (closer quartiles in the box plots) than FFT's or ARIMA's. The ability of clustering and classifying patterns in the data allows AO to optimize $\#_R$ and S_C to manage the tradeoff between capacity and

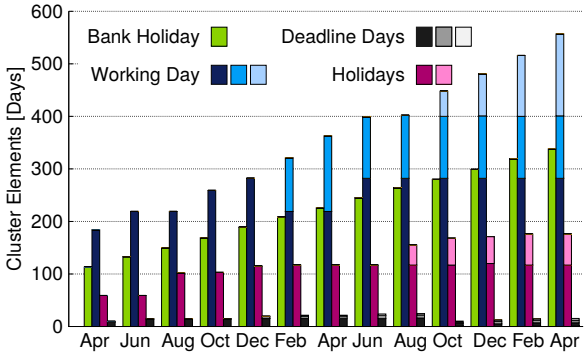

Figure 6: Traffic patterns over time observed in the collaborative service. AidOps captures traffic changes and dynamically creates patterns and updates existing traffic models.

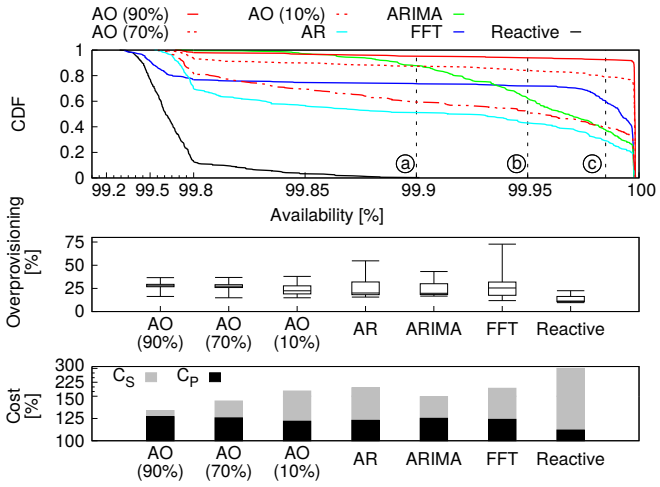

Figure 7: Performance and cost comparison between ARIMA, FFT and AO (with prediction accuracy of 90%, 70%, and 10%) for a commercial session border controller (SBC) with an A_v SLO of 99.985%. Only AidOps meets this requirement over the entire year. In this case, SLO violation penalties are high – which impacts the cost significantly. AO shows a total cost reduction with respect to ARIMA and FFT in ~ 18% and ~ 55%, respectively.

availability while achieving a more stable provisioning – here AO reconfigures ~ $4X$ less than ARIMA and FFT.

Last, we analyze the cost in eq. 8 normalized with eq. 9, that is: $(C_P + C_S)/C_I$. We have calculated k_2 and k_1 using data from commercial providers [2, 18, 34]. For the collaborative service we use k_2/k_1~50. Results are shown in figure 4 (bottom). A cost breakdown shows that AO almost eliminates the cost of violating SLO, C_S is ~ 3% and 5.5% lower than ARIMA and FFT. However, the total cost reduction is only ~ 1%; meaning that, these costs are essentially the same in this scenario – even similar to the cost of the on-demand solution. This happens because the SLOs for this service are quite relaxed and the penalties associated to violations are not excessively higher than provisioning costs. Therefore, although AO significantly improves the availability of the service, the cost is not affected much by violation penalties. In section 6.2 we show how rigid SLOs and higher penalties have a negative impact in the total cost.

Response to seasonal changes. We analyze transitions from working periods (with higher traffic) to vacation periods (lower traffic) and vice versa. Figure 5 shows the capacity provisioned by AidOps, FFT and ARIMA during the first working day in July (figure 5a) and September (figure 5b). Figures show the demand and capacity provisioned throughout the day, as well as CDFs of latency and over-provisioning. FFT over-provisions the service by a 75% in July due to the influence of the immediate past traffic. In September, however, its prediction falls short as it is influenced by summer lower traffic. This implies more scale in/out operations than in July and higher latency – the latency SLO is violated during 9% of the day. The season change does not affect ARIMA though. Still, bursty or fast traffic variations may impact the short term predictions leading to inaccurate scale outs – as happens between 7-8am in Figure 5b which causes SLO violations in 3% of the day. AidOps can predict these low-frequency seasonal changes and provision the service accordingly. As a result, SLOs are kept within limits without more over-provisioning than the other techniques.

Evolving traffic trends. Traffic is rarely stable as services and users change in time. AidOps is designed to learn from changes and recognize new patterns. Figure 6 shows the evolution of clusters over time, we have studied how new patterns are detected throughout one year of stable usage followed by two years of gradually increasing subscriber traffic – up to 20% more. The creation of a cluster depends on the median of the standard deviation of its elements. For this reason, clusters whose days differ the most will split to create new patterns. The *working day* pattern, for example, splits into two clusters in February. However, some of the new days are still assigned to the original cluster (increasing its size) because the distance between medians is lower than $2.\sigma$. This overlapping stops when the new cluster has a sufficient number of days. Last, patterns with very low standard deviation (e.g. bank holidays) remain in a single cluster throughout the two years.

6.2 Use case 2. A session border controller

In this section we experiment on a virtualized SBC, a carrier grade service with expected availability of 99.985%, i.e. SLO ⓒ. Figure 7 (top) shows the availability results. Here, only AO(90%) can achieve the expected availability in ~ 94% of the days, which yields an average availability of 99.985% over the year. ARIMA and FFT solutions achieve A_v of 99.94% and 99.85% throughout the year, respectively. FFT is highly affected by events that change the expected traffic – causing the solution to correct predictions often in many days, which leads to ~ 10 times more scale out/in operations than AidOps. Higher A_v allows AO to serve ~ $4X$ more traffic than ARIMA with a more stable provisioning. In figure 7 (middle) we show over-provisioning results. AO(90%) achieves 27% almost without any variability and outliers, while ARIMA and FFT reach up to 25% and 24%, respectively.

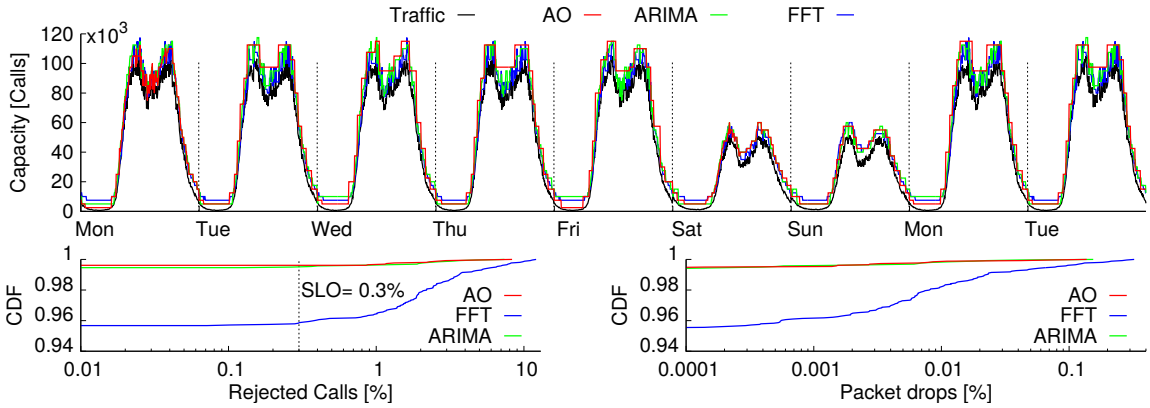

Figure 8: An operator promotion causing increasing traffic demands in the SBC. There is traffic deviation between predicted and current traffic because of this new event. Therefore, the prediction based on the autoregressive model is triggered. This allows to provision the service with almost no SLO violations on the first day. AidOps rapidly learns how to provision capacity for the following days, though. While short-term prediction strategies like ARIMA can adapt to these changes, long term ones like FFT fail as they require some time and data to adapt the model.

We also show the total cost achieved in this scenario, see figure 7 (bottom). Similarly to §6.1, we calculate k_2 and k_1 using cost data from commercial providers. In this case, we used cost figures of VM flavors that provide the hardware acceleration required by the SBC bearer plane, and SLO violation penalties of 10% and 25% of the provisioning cost for availability values lower than 99.985% and 99.9%, respectively. In this case, $k_2/k_1 = 450$.

AidOps reduces the total cost by approximately 18% and 55% compared to ARIMA and FFT. In this case, improving the availability reduces the cost significantly given the more strict SLOs and higher penalties. Note that on-demand elasticity is impractical for this service (as it reaches up to 300% cost), although it was still applicable to the collaborative service in section 6.1.

Learning new patterns and events. Last, we show how AidOps responds to external conditions that affect the workload. The data traffic used in this paper has traces of such conditions, like for instance: increasing content popularity, increasing subscriber base after a merger, and operator promotions. As an example, we focus on a 9-days promotion that causes an unexpected traffic increase. These 9 days are tagged as *P1* in the metadata-annotated calendar. Results are shown in figure 8. To provide a fair comparison, we assume that AidOps has no information about P1 during the first promotion day (i.e. there is not any manual capacity planning even though the service operator knows the traffic will grow); therefore, after detecting a traffic deviation, AidOps applies the autoregressive predictor described in § 5.2. At the end of the day, AidOps recreates the clusters to account for the new traffic. On day 2, this new pattern is not selected initially because it only happened once. However, AidOps will actually find it among the not consolidated patterns after detecting the traffic deviation – using the metadata classification. After day 5, AidOps considers P1 as a consolidated pattern and selects it right away without incurring in traffic deviation. Days 6 and 7 correspond to a weekend with promotion, this is essentially a new traffic with a similar effect than days 1 and 2. AidOps performs better than ARIMA as it uses AR predictions in only 2 out of 9 days. FFT, however, cannot predict the extra traffic during this week which leads to service degradation (i.e. rejected calls).

7 RELATED WORK

Commercial [3, 17, 35] and open-source [5, 38] cloud solutions offer ***on-demand auto-scaling*** to create instances based on predefined thresholds. As mentioned in § 1, this approach has limitations when dealing with high-availability services as provisioning delays and dynamic workloads can cause service degradation. Some solutions [22] complement threshold-based rules with traffic predictions but without control on the scaling actions. That is, the number of reconfigurations is arbitrary and driven by the traffic, which increases the probability of SLO violations [8, 21], as shown in § 6.

Other proposals use ***theoretical models*** [1, 19] with a simple mathematical structure and adjustable parameters that are calibrated with techniques that span from *Kalman* filtering [14, 28] to reinforcement learning [56]. Some solutions combine control or queuing theory with predictive algorithms [54] and performance models [6, 13]. Simple models, however, can be insufficient to represent the characteristics of complex distributed services. Queue models, for example, need intricate modifications to capture transient behaviors properly, whereas automation is required to adjust parameters online in heterogeneous systems.

Other techniques leverage ***empirical models*** created from experience, or profiling and benchmarking the full stack [31, 40]. These models can deal with the complexity and non-linearities across infrastructure layers, as well as the intricate dependencies and complex features of high-availability applications [24]. They can be represented with rather simple datasheet-style information [41] or more sophisticated approaches based on fuzzy logic [23]. This approach, however, requires substantial knowledge about the service configuration and components to profile the system properly.

Some solutions perform ***time series analysis*** to forecast service demands. Many of them focus on short-term trends – examples are

ARMA [44, 55], ARIMA [9, 32], and EMA [25]. Recent research propose to combine the above predictors in runtime [12, 20]. Other proposals identify patterns by using from signal processing [16, 22, 46] and wavelets [36] to hidden Markov models [29], Bayes classifiers [11, 39] and neural networks [37]. In general, Bayes-based methods are more accurate for long term predictions while autoregressive models perform well in the short term and are simpler.

Instead of predicting capacity for the immediate demand, AidOps analizes time series in historical data to extract domain-specific, time-defined traffic models and use them to optimize capacity in terms of scaling actions, anticipative scaling time and spare capacity. This way, we can effectively combine proactive cyclic scaling [52] with on-demand provisioning to achieve higher availability.

Last, it is worth to note that some of the solutions above can be found in slightly different domains that require efficient resource scheduling, as for example scientific computing [30, 45] or business-critical data analytics [27]. The key aspect such domains is performance predictability (i.e. meeting job deadlines) as the uncertainty in these applications comes from the amount of time to complete a given demand rather that the amount or volume of demand itself.

8 CONCLUSION

AidOps automates the creation of traffic models to assist cloud orchestrators in dynamically provisioning capacity according to operator cost and SLO expectations. We use machine learning to extract patterns of customer traffic in order to create accurate predictions for provisioning high-availability services in advance. Our algorithms extract and classify utilization patterns from historical data to forecast the best service configuration while quickly detecting seasonal or sudden traffic changes. We have shown that including traffic models in the orchestration system allows for a joint prediction and optimization of resources, leading to higher availability and lower over-provisioning. That is, AidOps can provision services more efficiently, minimize SLO violations and reduce reconfigurations with a more controlled operation. Thus, service providers can make conscious trade-offs between over-provisioning, stability and service quality by simply parameterizing our algorithms.

The evaluation was conducted using services with very different availability requirements, specifically, an enterprise collaborative service with relaxed response times and a real-time communication service. Experiments revealed that including the cost of SLO violations in the orchestration process has several implications in the system elasticity –particularly for high-availability services– requiring an optimized resources dimensioning in terms of available capacity and provisioning times. Results show that AidOps fulfills availability SLOs of up to 99.985% with up to 20% cost reductions.

In the future, we plan on improving the service models used to translate capacity predictions to resource demand by including active resource measurements instead of empirical models. Also, we will continue to investigate on alternatives to use as short-term predictors after AidOps detect unseen workloads to achieve better accuracy without impacting responsiveness. Last, although we experimented with unrelated services running on different virtualization technologies, we still need extend the evaluation with more applications to fully assess the generality of AidOps.

9 APPENDIX.

Notation and definitions.

D : historical data set
F_i : set of feature values of day i
T : set of time slots
V : traffic volume at time slots in T of a day
S_i : a time series of traffic volume of day i. $S_i = \{T, V_i\}$
$CV(j,t)$: coefficients of variation of cluster c_j at time t
$CVM(k,j)$: median of $CV(j,t)$ when there are k clusters in total
$\hat{F}_j$: set of feature values of cluster c_j
I_j : identifier of cluster c_j
θ_j : number of days within cluster c_j
P_j : traffic pattern of cluster c_j
$\hat{V}$: a series of average traffic volume of a cluster
Λ : a series of standard deviation of a cluster
M : minimum number of days in a consolidated cluster
C : set of consolidated clusters
$\mathcal{C}$: set of unconsolidated clusters
D_s : training data for prediction accuracy, random samples of D
p_j : probability of a day belong to cluster c_j
f_l : binary indicator of feature value l
$\alpha_{j,l}$: coefficient for feature value l of cluster c_j
$\alpha_{j,0}$: constant of multinomial logistic regression for cluster c_j
Σ_j : coefficients of features of cluster c_j
p : probability of the current traffic being atypical
r_t : absolute value of traffic difference at time slot t
L : size of time window in the logistic regression model
$\delta_{j,l}$: coeff. of traffic difference at relative time l for cluster c_j
$\delta_{j,L}$: constant in the logistic regression model for cluster c_j
Δ_j : coefficients of traffic differences of cluster c_j
$\#_R$: number of reconfigurations
$cost_{ij}$: total capacity to cover the demands between reconfigurations at i and $j+1$ in the dynamic programming
x_{ij} : decision variables in the dynamic programming on whether there are reconfigurations at i and $j+1$
$A_{i,k}$: the minimum capacity from node i to the end of the time horizon with k arches in the dynamic programming.

REFERENCES

[1] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *2012 IEEE Network Operations and Management Symposium*, pages 204–212, April 2012.

[2] Amazon. EC2 Service Level Agreement. https://aws.amazon.com/ec2/sla/. Accessed Apr 2017.

[3] Amazon. Web Services Auto Scaling. https://aws.amazon.com/autoscaling/. Accessed Apr 2017.

[4] Ansible. Automation for everyone. https://www.ansible.com/. Accessed Apr 2017.

[5] Apache CloudStack. Open Source Cloud Computing. http://cloudstack.apache.org/. Accessed Apr 2017.

[6] D. A. Bacigalupo, J. van Hemert, A. Usmani, D. N. Dillenberger, G. B. Wills, and S. A. Jarvis. Resource management of enterprise cloud systems using layered queuing and historical performance models. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

[7] S. A. Baset. Cloud SLAs: Present and Future. *SIGOPS Oper. Syst. Rev.*, 46(2):57–66, July 2012.

[8] R. Buyya, S. K. Garg, and R. N. Calheiros. SLA-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 1–10, Dec 2011.

[9] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya. Workload Prediction Using ARIMA Model and Its Impact on Cloud Application's QoS. *IEEE Transactions on Cloud Computing*, 3(4):449–458, Oct 2015.

[10] M. Clougherty and V. Hilt. chapter The future of the cloud, pages 163–198. CRC Press, Mar 2016.

[11] S. Di, D. Kondo, and W. Cirne. Google hostload prediction based on Bayesian model with optimized feature combination. *Journal of Parallel and Distributed Computing*, 74(1):1820 – 1832, 2014.

[12] H. Fernandez, G. Pierre, and T. Kielmann. Autoscaling Web Applications in Heterogeneous Cloud Infrastructures. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 195–204, March 2014.

[13] A. Gambi and G. Toffetti. Modeling Cloud performance with Kriging. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1439–1440, June 2012.

[14] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Adaptive, Model-driven Autoscaling for Cloud Applications. In *USENIX 11th International Conference on Autonomic Computing*, ICAC 2014, pages 57–64, 2014.

[15] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Resource Pool Management: Reactive Versus Proactive or Let's Be Friends. *Comput. Netw.*, 53(17):2905–2922, Dec. 2009.

[16] Z. Gong, X. Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16, Oct 2010.

[17] Google. Compute Engine Auto Scaler. https://cloud.google.com/compute/docs/autoscaler/. Accessed Apr 2017.

[18] Google. Compute Engine SLA. https://cloud.google.com/compute/sla/. Accessed Apr 2017.

[19] R. Han, M. M. Ghanem, L. Guo, Y. Guo, and M. Osmond. Enabling Cost-aware and Adaptive Elasticity of Multi-tier Cloud Applications. *Future Gener. Comput. Syst.*, 32:82–98, Mar. 2014.

[20] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn. Self-adaptive workload classification and forecasting for proactive resource provisioning. *Concurrency and Computation: Practice and Experience*, 26(12):2053–2078, 2014.

[21] R. H. Hwang, C. N. Lee, Y. R. Chen, and D. J. Zhang-Jian. Cost Optimization of Elasticity Cloud Resource Subscription Policy. *IEEE Transactions on Services Computing*, 7(4):561–574, Oct 2014.

[22] D. Jacobson, D. Yuan, and J. Neeraj. Scryer: Netflix's predictive auto-scaling engine. The Netflix Tech Blog. Accessed Apr 2017.

[23] P. Jamshidi, A. Ahmad, and C. Pahl. Autonomic Resource Provisioning for Cloud-based Software. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 95–104, 2014.

[24] J. Jiang, V. Sekar, H. Milner, D. Shepherd, I. Stoica, and H. Zhang. CFA: A Practical Prediction System for Video QoE Optimization. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 137–150, Berkeley, CA, USA, 2016. USENIX Association.

[25] F. Jokhio, A. Ashraf, S. Lafond, I. Porres, and J. Lilius. Prediction-Based Dynamic Resource Allocation for Video Transcoding in Cloud Computing. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 254–261, Feb 2013.

[26] B. Jose and S. D. M. Kumar. Telecom grade cloud computing: Challenges and opportunities. In *Signal Processing, Informatics, Communication and Energy Systems (SPICES), 2015 IEEE International Conference on*, pages 1–5, Feb 2015.

[27] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 117–134, Berkeley, CA, USA, 2016. USENIX Association.

[28] E. Kalyvianaki, T. Charalambous, and S. Hand. Self-adaptive and Self-configured CPU Resource Provisioning for Virtualized Servers Using Kalman Filters. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, pages 117–126, Barcelona, Spain, 2009. ACM.

[29] A. Khan, X. Yan, S. Tao, and N. Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *2012 IEEE Network Operations and Management Symposium*, pages 1287–1294, April 2012.

[30] I. K. Kim, J. Steele, Y. Qi, and M. Humphrey. Comprehensive Elastic Resource Management to Ensure Predictable Performance for Scientific Applications on Public IaaS Clouds. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 355–362, Washington, DC, USA, 2014. IEEE Computer Society.

[31] S. Lange, A. Nguyen-Ngoc, S. Gebert, T. Zinner, M. Jarschel, A. Kopsel, M. Sune, D. Raumer, S. Gallenmuller, G. Carle, and P. Tran-Gia. Performance Benchmarking of a Software-based LTE SGW. In *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM)*, CNSM '15, pages 378–383, Washington, DC, USA, 2015. IEEE Computer Society.

[32] H. Lin, X. Qi, S. Yang, and S. Midkiff. Workload-Driven VM Consolidation in Cloud Data Centers. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 207–216, May 2015.

[33] S. Makridakis and M. Hibon. The M3-Competition: results, conclusions and implications. *International journal of forecasting*, 16(4):451–476, 2000.

[34] Microsoft. Azure SLA. https://azure.microsoft.com/en-gb/support/legal/sla/. Accessed Apr 2017.

[35] Microsoft. AzureScale. http://azure.microsoft.com/features/autoscale/. Accessed Apr 2017.

[36] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, San Jose, CA, 2013. USENIX.

[37] A. Y. Nikravesh, S. A. Ajila, and C. H. Lung. Towards an Autonomic Auto-scaling Prediction System for Cloud Resource Provisioning. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 35–45, May 2015.

[38] OpenStack. Open source software for creating clouds. https://www.openstack.org/. Accessed Apr 2017.

[39] J. Panneerselvam, L. Liu, N. Antonopoulos, and Y. Bo. Workload Analysis for the Scope of User Demand Prediction Model Evaluations in Cloud Environments. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 883–889, Dec 2014.

[40] P. Patil, P. Kulkarni, and U. Bellur. VirtPerf: A Performance Profiling Tool for Virtualized Environments. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 57–64, July 2011.

[41] Project Clearwater. Clearwater Deployment Sizing. http://www.projectclearwater.org/wp-content/uploads/2013/05/Clearwater-Deployment-Sizing-10-Apr-13.xlsx. Accessed Apr 2017.

[42] Prometheus. Prometheus. From metrics to insight. https://prometheus.io/. Accessed Apr 2017.

[43] Rancher. A Platform for Running Containers. http://rancher.com//. Accessed Apr 2017.

[44] N. Roy, A. Dubey, and A. Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507, July 2011.

[45] A. Ruiz-Alvarez, I. K. Kim, and M. Humphrey. Toward Optimal Resource Provisioning for Cloud MapReduce and Hybrid Cloud Applications. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 669–677, June 2015.

[46] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, Cascais, Portugal, 2011. ACM.

[47] J. Soares, C. GonÃğalves, B. Parreira, P. Tavares, J. Carapinha, J. P. Barraca, R. L. Aguiar, and S. Sargento. Toward a telco cloud environment for service functions. *IEEE Communications Magazine*, 53(2):98–106, Feb 2015.

[48] P. Suthar and M. Stolic. Carrier grade Telco-Cloud. In *Wireless and Mobile (APWiMob), 2015 IEEE Asia Pacific Conference on*, pages 101–107, Aug 2015.

[49] T. Taleb. Toward carrier cloud: Potential, challenges, and solutions. *IEEE Wireless Communications*, 21(3):80–91, June 2014.

[50] P. Tang, F. Li, W. Zhou, W. Hu, and L. Yang. Efficient Auto-Scaling Approach in the Telco Cloud Using Self-Learning Algorithm. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, Dec 2015.

[51] H. L. Truong and S. Dustdar. Programming Elasticity in the Cloud. *Computer*, 48(3):87–90, Mar 2015.

[52] J. Varia. *Best Practices in Architecting Cloud Applications in the AWS Cloud*, pages 457–490. John Wiley & Sons, Inc., 2011.

[53] A. Wolke, M. Bichler, and T. Setzer. Planning vs. dynamic control: Resource allocation in corporate clouds. *IEEE Transactions on Cloud Computing*, PP(99):1–1, 2015.

[54] W. Xu, X. Zhu, S. Singhal, and Z. Wang. Predictive Control for Dynamic Resource Allocation in Enterprise Data Centers. In *2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006*, pages 115–126, April 2006.

[55] J. Yang, C. Liu, Y. Shang, Z. Mao, and J. Chen. Workload Predicting-Based Automatic Scaling in Service Clouds. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 810–815, Washington, DC, USA, 2013. IEEE Computer Society.

[56] Q. Zhu and G. Agrawal. Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments. *IEEE Transactions on Services Computing*, 5(4):497–511, Fourth 2012.