BIG DATA MINING AND ANALYTICS
ISSN 2096-0654 04/15 pp315–339
DOI: 10.26599/BDMA.2023.9020014
Volume 7, Number 2, June 2024

Extending OpenStack Monasca for Predictive Elasticity Control

Giacomo Lanciano, Filippo Galli, Tommaso Cucinotta*, Davide Bacciu, and Andrea Passarella

Abstract: Traditional auto-scaling approaches are conceived as reactive automations, typically triggered when predefined thresholds are breached by resource consumption metrics. Managing such rules at scale is cumbersome, especially when resources require non-negligible time to be instantiated. This paper introduces an architecture for predictive cloud operations, which enables orchestrators to apply time-series forecasting techniques to estimate the evolution of relevant metrics and take decisions based on the predicted state of the system. In this way, they can anticipate load peaks and trigger appropriate scaling actions in advance, such that new resources are available when needed. The proposed architecture is implemented in OpenStack, extending the monitoring capabilities of Monasca by injecting short-term forecasts of standard metrics. We use our architecture to implement predictive scaling policies leveraging on linear regression, autoregressive integrated moving average, feed-forward, and recurrent neural networks (RNN). Then, we evaluate their performance on a synthetic workload, comparing them to those of a traditional policy. To assess the ability of the different models to generalize to unseen patterns, we also evaluate them on traces from a real content delivery network (CDN) workload. In particular, the RNN model exhibites the best overall performance in terms of prediction error, observed client-side response latency, and forecasting overhead. The implementation of our architecture is open-source.

Key words: elasticity control; auto-scaling; predictive operations; monitoring; OpenStack; Monasca

1 Introduction

Over the last decade, information and communications technologies (ICTs) have been evolving non-stop at an extremely rapid pace. The ever-growing availability of low-cost high-bandwidth connectivity has been one of the key enablers paving the way for the impressive growth in the adoption of distributed computing paradigms. Cloud computing[1] emerged as the defacto standard for developing and deploying large-scale production-grade applications. This paradigm allows for completely decoupling the management of physical infrastructures from the services deployed on top of them, by heavily relying on virtualization. This enabled to make a more efficient use of physical resources and have a higher resiliency degree for the hosted applications. However, cloud computing has significantly evolved and is not only limited to the infrastructure-as-a-service (IaaS) provisioning model, according to which users can access compute instances (e.g., virtual machines (VMs)) deployed on top of

• Giacomo Lanciano and Filippo Galli are with the Scuola Normale Superiore, Pisa 56126, Italy. E-mail: giacomo.lanciano@sns.it; filippo.galli@sns.it.
• Tommaso Cucinotta is with the Real-Time Systems Laboratory (RETIS), Telecommunications, Computer Engineering, and Photonics Institute (TeCIP), Scuola Superiore Sant'Anna, Pisa 56127, Italy. E-mail: tommaso.cucinotta@santannapisa.it.
• Davide Bacciu is with the Department of Computer Science, University of Pisa, Pisa 56127, Italy. E-mail: davide.bacciu@unipi.it.
• Andrea Passarella is with the National Research Council, Pisa 56127, Italy. E-mail: andrea.passarella@iit.cnr.it.
* To whom correspondence should be addressed.
Manuscript received: 2022-10-22; revised: 2023-04-24; accepted: 2023-06-16

shared physical servers and operated by the provider. Nowadays, the so-called everything-as-a-service (XaaS) provisioning model enables an application developer to realize cloud-native services, by leveraging on a wide range of orchestration, load-balancing, storage, and monitoring solutions, completely managed by the provider[2].

To operate their infrastructures 24/7, cloud providers need operation teams ready to promptly address and fix any kind of issue that might occur, including hardware faults and software defects. In production-grade cloud infrastructures, this is only feasible when such systems are designed following well-established practices (e.g., fault-independent zones, redundant powering and cooling infrastructures, multi-path networking topologies, etc.) and operated using appropriate tools (e.g., monitoring systems, resource managers, effective automation rules, etc.).

In this regard, a key enabling factor is the presence of fine-grained monitoring services, on top of which automation rules can be built, ensuring high reliability for the hosted services and performance levels that are as stable as possible, despite sudden changes in traffic conditions. This refers to elasticity, which is the capability of cloud services to automatically adapt their set of allocated resources (e.g., VMs, containers, or even physical nodes) as the workload changes over time. Elasticity is typically implemented by means of a control loop that decides which actions to take in order to keep the service running smoothly (e.g., scale-out or scale-in). Such scaling decisions are usually made on the basis of system-level resource consumption metrics (e.g., CPU utilization, network traffic, and storage load), as well as key performance indicators (KPIs) at application-level (e.g., response times and connection timeouts/errors).

1.1 Problem presentation

Classical auto-scaling mechanisms are inherently based on reactive automation rules that scale a service whenever some metric breaches predefined thresholds. Traditionally, after a scaling action is actually triggered, an elastic system enters a cooldown period that prevents further scaling actions until it expires. This is done such that the elasticity controller can take its subsequent decisions by actually considering the effects of the previous one. In addition, to make the overall mechanism more robust with respect to transient changes in the workload, it is common to require the threshold to be breached for a few consecutive observations before triggering any action.

Developing and tuning such automation rules become particularly cumbersome when dealing with large-scale production environments. There are many challenges to be addressed, like determining the KPIs to accurately estimate the status of the system, setting the frequency at which the scaling decision should be evaluated, adapting the scaling policy to changes in the workload to prevent unnecessary scaling actions, estimating the amount and type of instances to add to handle the new conditions, and taking into account non-negligible time to set up the additional instances. For instance, in this work we put great emphasis on the latter challenge because, in large-scale production environment, spawning new instances might indeed take from a few minutes to even half an hour[3]. This is not only the time needed to instantiate and boot a new VM but also the time needed to configure the new instance, possibly install any missing software, in case the same image is re-used as a base for a number of different roles requiring customized software set-ups, or just install some minimally required security updates, sometimes copy onto the image a minimum set of information or local database needed for the software to operate correctly, start the actual service and register it into a load-balancing group, and finally some further time is needed for the new instance to progressively pick new traffic. Therefore, in such settings, anticipating scale-out operations becomes critical. Despite the mentioned precautions, traditional control loops are still inherently "dumb", as they do not factor the rich dynamic of the observed metrics in their decisions. For instance, consider the CPU utilization evolution depicted in Fig. 1. A classical scaling policy would treat scenarios *A* and *B* in pretty much the same way: as soon as the CPU utilization breaches the threshold (i.e., the red line), a scaling action is

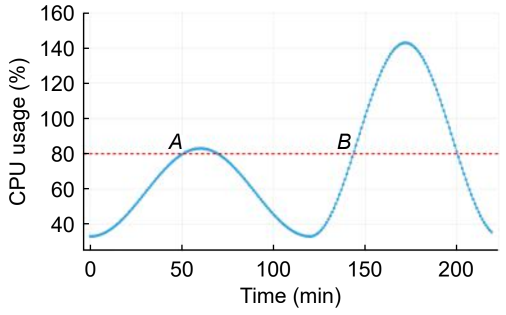


Fig. 1 Overview of OpenStack key components.

triggered. However, a human operator, based on its prior experience gained while operating such a fictional service, could easily distinguish between scenarios *A* and *B*. While *A* might be safely ignored, it is clear that *B* would require urgent actions to be taken.

In the context of large-scale cloud environments, given the complex relationships among their components and the abundance of operational data they generate (e.g., event logs, application metrics, source code, etc.), treating operations as a data science problem[4] seems a promising approach to develop "intelligent" automations. In particular, time-series forecasting techniques based on machine learning (ML) may play a fundamental role in enhancing the capabilities of elasticity controllers to prevent services from saturating their capacity. For instance, AWS currently provides native support for predictive scaling with EC2[5], demonstrating the suitability of this type of approaches at supporting cloud operations[6]. On a related note, ML-based approaches have also been shown to be beneficial for efficient and sustainable management of resources in cloud data centers[7].

1.2 Contribution

In this paper, we propose an open-source software architecture for integrating predictive analytics within an OpenStack cloud platform. The paper provides three major contributions. First, a general architecture for performing predictive operations on a cloud infrastructure based on time-series forecasting techniques. Second, an open-source implementation of the forecasting component within OpenStack, leveraging on Monasca[8], that automatically computes forecasts and makes them available as additional metrics. Our implementation also includes a few reference implementations of metric predictors, i.e., linear regression (LR)[9], autoregressive integrated moving average (ARIMA)[10], multi-layer perceptron (MLP), and recurrent neural network (RNN), showing that the proposed architecture is flexible, as it allows for easy customization. Third, an extensive experimental validation of our architecture, using both synthetic and real content delivery network (CDN) workload traces, where we set up a synthetic elastic application, exploit the native capabilities of OpenStack, and compare the performance of several predictive elasticity controllers based on the aforementioned reference predictors. When compared to related research in the field, our proposed approachcan be distinguished in that it is the only work providing an open-source architecture for extending the orchestration capabilities of OpenStack with modular predictive analytics, enabling forecast-driven decision-making for generic elastic cloud services. More details can be found in Section 2.

This paper extends our preliminary work appeared at the IEEE/ACM UCC 2021 conference[11]. We add a more comprehensive experimental validation that includes experimentation with not only synthetic traces as done in Ref. [11] but also real CDN workload traces from the RECAP[12] EU project. Furthermore, we expand the set of predictors and their configurations, used for the validation. Indeed, in Ref. [11], we only considered a single input size configuration for LR, MLP, and RNN. On the other hand, in this paper, we add ARIMA to the set of predictors and also considered 3 different input size configurations for each of them, such that we could assess how this parameter influences the overall performance of the resulting scaling policies. Also, we provide additional implementation and configuration details in the description of our proposed approach and significantly extended the discussion of related research in the area.

1.3 Paper organization

This paper is organized as follows. Section 2 provides a detailed overview of the related research literature, highlighting how the proposed technique is positioned in the current landscape. In Section 3, we provide key background concepts about OpenStack and time-series forecasting techniques for a better comprehension of the following material. Section 4 describes the approach proposed in this paper, while Section 5 presents its experimental validation on an OpenStack deployment. Some final remarks are enclosed in Section 6, along with the discussion of possible ideas for future works on the topic.

2 Related Work

In the research literature, a number of authors applied data-driven techniques to automated elasticity control, both for public and private cloud. In what follows, we start by reporting key research works dealing with predictive elasticity based on metric forecasting for public cloud. Then, we provide a similar overview of related research for private cloud with a particular focus on network function virtualization (NFV)[13], due to the increasing interest gained by ML in the context of network service chains. Finally, we provide a brief review of elasticity-control solutions based on reinforcement learning (RL).

2.1 Predictive elasticity in cloud computing

A variety of data-driven techniques have been proposed to provide accurate short-term predictions of workloads and resource consumption patterns of elastic clusters to achieve a more timely and fine-tuned allocation of resources.

In Ref. [3], the authors described a simple scaling strategy based on predicting the aggregate sum of transmitted and received bytes of a service cluster, considering resource setup delays and limited deployment throughput. The approach leverages on a workload model to estimate a percentile of the resource demand and a probabilistic function that describes the cost of over-/under-provisioning the cluster. The authors presented promising results by evaluating the technique against data from more than 40 000 real services deployed as auto-scaling groups on AWS.

In Ref. [14], the authors proposed a model-predictive control based approach[15], combining three major techniques: a 2nd-order ARMA filter for workload prediction; a customer behavior modeling graph (CBMG)[16], optimized on web logs, capturing the behavior of users while browsing a web application; and a look-ahead optimization to trade-off between the advantages arising from dynamic elasticity and the cost of scaling decisions and cluster reconfiguration at each control period. They empirically evaluated their technique against data from the 1998 world cup web-site traffic.

In Ref. [17], the authors tackled the problem of non-instantaneous instance provisioning when using elastic scaling in cloud environments. They proposed a predictive strategy based on a resource prediction model using ANNs and LR. The method was applied on an e-commerce application scenario emulated through the TPC-W[18] workload generator and benchmarking application, deployed on AWS EC2. ANNs improve the accuracy by reducing the MAPE by roughly 50% compared to LR.

In Ref. [19], an ANN with a single hidden layer was proposed for predicting the resource utilization and duration of continuous integration tasks for several repositories from the Travis open data. However, the evaluation focused on predicting the task duration only, using a per-repository model using the number of files and the repository size as inputs. Results show an accuracy at least 20% and up to 89% better than a baseline LR.

In Ref. [20], the authors proposed supervised learning methods to tackle the problem of predictive auto-scaling for multi-tier elastic applications, considering unstable performance of individual VMs. In particular, LR is applied to the traffic arrival rate time-series to predict short-term arrival rates, which were in turn used to predict the evolution of the response times, using polynomial regression (PR). Such estimates were then fed to a random decision forest (RDF), designed to learn a configurations map associating the order of configurations—required to maintain the specified service-level objective (SLO)—to the experienced request arrival rates and system response times. The training data for the RDF were generated by executing a few system (static) scaling policies.

In the RScale framework[21], Gaussian process (GP) regression was used to predict end-to-end tail-latency of distributed microservices workflows with generic direct acyclic graph-like topologies. RScale was evaluated on the Chamaleon test-bed☆ and achieved similar accuracy but a smaller predicted uncertainty with respect to ANNs. However, it exhibited reduced inference overheads and superior adaptability to dynamically changing workload/interference conditions.

In Ref. [22], Bayesian networks (BNs) were used in a predictive framework to support automatic scaling decisions in cloud services. The method was evaluated on synthetic applications with exponentially distributed duration and workload inter-arrival patterns.

In Ref. [23], decision trees (DTs) were used to predict CPU, memory, and network usage of hive-based MapReduce queries over a Hadoop cluster. The authors used a 4-machines cluster to perform queries with different structures over a number of different datasets, using a per-resource decision tree to classify the query within the high or low resource-usage class. The presented results give insights as to the parameters mostly affecting the consumption level for each resource. However, as the authors used a fixed-size cluster, the technique does not seem to be useful in the context of elasticity control, albeit the investigation may be useful to design effective elasticity rules.

☆https://www.chameleoncloud.org/

In Ref. [24], the authors proposed a proactive auto-scaling mechanism for edge computing applications on Kubernetes. The approach leverages on ARMA and long short-term memory (LSTM) to estimate the raw number of additional compute instances needed, given the observed resource utilization patterns. The authors also provided a mechanism to either automatically retrain from scratch or incrementally update the underlying model.

In Ref. [25], the authors introduced an extensive set of traces exported from Azure's internal infrastructure. They proposed Resource Central, an approach that collects VM utilization metrics and periodically trains prediction models on them offline. Such models can then be queried online by resource management systems and/or human operators. While the approach is in theory agnostic to the underlying models, the authors considered RDF, gradient boosted trees (GBTs) and fast fourier transform (FFT) for their experiments. The authors validated their approach by integrating it with Azure's VM scheduler, showing performance improvements also in over-subscription scenarios.

In Ref. [26], the authors proposed a framework to forecast the workload of a cloud system, such that a resource manager can take informed scaling decisions. Their approach is based on self-directed learning (SDL), which consists in including recent forecast errors in the input to the underlying model, such that it can be used as feedback to improve the accuracy of future predictions. The model is a feed-forward ANN, whose weights are optimized via an improved version of the blackhole algorithm proposed by authors. The authors validated their approach against 6 different datasets exported from real systems.

In Ref. [27], the authors proposed a proactive resource scaling approach that leverages on a workload prediction module. The underlying forecasting model is based on ARIMA. The approach uses the predicted information to resize a cloud application accordingly, e.g., anticipating peaks. The authors used real traces exported from web servers of Wikipedia to train ARIMA to predict request patterns. They also validated their approach, in terms of impact on the quality of service (QoS) of a cloud application, by running simulations on CloudSim.

In Ref. [28], the authors proposed CloudInsight, a workload prediction framework that can be used to proactively scale cloud applications. The authors leveraged on an ensembling approach (i.e., combining the outputs from several models) to effectively handle irregular, dynamically changing workloads. The weight of an individual model is continuously re-evaluated by a mechanism based on support vector machine (SVM), such that the system can adapt to the current shape of the workload. The authors validated their approach against 3 different classes of workloads (exported from real systems) and compared its performance to several baseline predictors.

In Ref. [29], the authors proposed LoadDynamics. Similar to Ref. [28], they put the emphasis on the sensitivity to workload changes that are observed in most workload prediction frameworks. Their solution is an LSTM-based approach that is trained and evaluated on data exported from real systems, which describes requests arrival rates in different application scenarios (e.g., public cloud, HPC, web, etc.).

There exist other approaches that perform dynamic resource allocation based on instantaneous monitoring, rather than on resource estimations. For instance, the authors of Ref. [30] proposed a vertical elasticity management approach for containers, to dynamically adapt the allocated memory in Kubernetes, to support the co-location of containers having heterogeneous QoS requirements. However, for brevity, we omit this type of approaches from our overview, as they fall within the research literature on classical reactive elasticity control.

2.2 Predictive elasticity for NFV services

Predictive analytics have also been investigated in private cloud scenarios, notably for NFV and software defined networks (SDNs)[31] in order to adapt and fine-tune the allocation of virtualized resources to the conditions of the network. In this way, operators can benefit from proactive automation mechanisms to ensure QoS for their cloud-native service-chains. In Ref. [32], the authors highlighted the main challenges to be tackled to obtain an effective predictive mechanism: assessment of the bottleneck components that need to be scaled; development of mechanisms for optimal consolidation of the virtual resources within the physical infrastructure; and the design and implementation of predictive models preventing under- or over-provisioning of virtualized resources.

In Ref. [33], an ML-based approach was proposed to realize an effective auto-scaling mechanism. The authors evaluated several predictive models (e.g., DT, RDF, MLP, and BN) on load traces exported from a real virtual network function (VNF) environment, also taking into account the different costs and start-up time related to different virtualization technologies.

RNNs proved to be a powerful tool for time-series analysis in forecasting[34, 35] and classification[36, 37] tasks. For instance, in Ref. [38], LSTM was used to forecast the future demand of deployed VNFs. However, the authors mainly focused on feature-selection aspects rather than achievable accuracy. Also, the authors of Ref. [39] showed that the sequence-to-sequence architecture, wide-spread in the natural language processing (NLP) research field, yields surprisingly good results. Such architecture consists of an encoder and a decoder module. In NFV, these models have been used to capture complex relationships between VNF and infrastructure metric time-series[40]. Remarkably, since the encoder and the decoder only exchange information consisting in the hidden state values, it is also possible to use different sets of metrics for input and output.

Additional information about the deployed VNFs (e.g., graph-like diagrams depicting the interactions among the VMs belonging to the same VNFs) can also be used to boost forecasting accuracy. For instance, in Ref. [41], the authors proposed a topology-aware forecasting approach built on top of graph neural networks (GNNs)[42].

2.3 Elasticity control with reinforcement learning

Straightforward heuristics like static thresholding[43] can yield amazing results, when dealing with elasticity control for simple systems. However, thresholds require careful ad-hoc tuning, resulting in an approach that can hardly be adopted at scale (as it will eventually lead to over- or under-provisioning). Addressing these shortcomings, dynamic thresholding mechanisms, like the ones based on RL, offer the capability to automatically adapt thresholds to the current status of the system.

However, RL algorithms usually impose demanding computing requirements, which may limit their applicability on real systems. For instance, in Ref. [44], the authors described a Q-learning approach for managing a real telco system. The developed agent is observed to take several unexpected decisions, before converging to an optimal policy. When deploying the approach in production, this is clearly not desirable. On the other hand, in Ref. [45], the authors proposed a successful RL-based approach to deploy VNF service chains, which works by jointly minimizing operation costs and maximizing requests throughput, and also took into consideration heterogeneous QoS requirements.

In Ref. [46], the authors proposed an adaptive mechanism to automatically learn scaling policies for NFV, based on Q-learning and GP. They leveraged on GP to iteratively improve the learned policy before taking the final scaling decision, using the average response time of the system as reward signal. They evaluated their approach on a simulated NFV environment, showing that it outperforms both a standard threshold based policy and a Q-learning based one (not based on GP).

In Ref. [47], the authors proposed two different novel auto-scaling strategies based on the combination of a fuzzy logic (FL) controller with two different RL approaches: Q-learning (i.e., off-policy approach) and state action reward state action (SARSA) (i.e., on-policy approach). According to the authors, employing RL algorithms makes the overall mechanism self-adaptive (e.g., considering the response time as reward signal), while the FL controller enables it to work at a higher level of abstraction. Both strategies are implemented and integrated with OpenStack. The evaluation is performed on two different real web application workloads.

In Ref. [48], the authors proposed a strategy based on a deep Q network (DQN) for tuning the scaling thresholds used by the auto-scaling rules of microservices deployed in Kubernetes. The application-level response time is extracted from the log files of a Twitter analytics application and used as reinforcement signal for the RL algorithm.

In Ref. [49], the authors investigated on using RL-based techniques to handle resource allocations and scaling in the context of a serverless computing framework. They focused on request-based scaling and developed a mechanism to automatically adapt the concurrency level of a serverless application instance (i.e., the maximum number of requests that a single instance should handle).

2.4 Summary

Table 1 reports a schematic comparison among the major related works and our proposed approach (at the bottom of the table). The comparison takes into account the following aspects: which data-driven technique was used and which input data was applied to; whether the method applied to generic workloads,

Table 1 Related works comparison (legend: G.A. = generally applicable; S.O. = spawning overhead; E. = elasticity; and O.S. = open-source).

Reference	Technique	Input	Validation	G.A.	S.O.	E.	O.S.
[3]	Heuristic	Network data	40K AWS auto-scaling groups	Y	Y	Y	N
[14]	ARMA, CBMG	Utilization data	'98 world cup	N	Y	Y	N
[17]	ANN, LR	Utilization data	E-commerce (TPC-W)	Y	Y	Y	N
[19]	LR, MLP	Utilization data	CI pipeline	N	N	Y	N
[20]	PR, RDF	Request rate and response time	E-commerce (RUBiS)	N	N	Y	N
[21]	GP	Utilization data	Robot shop	Y	N	Y	N
[22]	BN	Utilization data	Synthetic workload	Y	N	Y	N
[23]	DT	Query type and structure	DB queries (Hive)	N	N	N	N
[24]	ARMA, LSTM	Utilization data	Kubernetes edge app	Y	N	Y	N
[25]	RDF, GBT, FFT	Utilization data	Real Azure VM traces	Y	Y	Y	N
[26]	MLP, SDL, blackhole	Request rate and utilization data	6 different real datasets	Y	N	Y	N
[27]	ARIMA	Request rate	Real Wikipedia traces	Y	Y	Y	N
[28]	Ensembling	Request rate	3 classes of real workloads	Y	N	Y	N
[29]	LSTM	Request rate	3 classes of real workloads	Y	N	Y	N
[32]	Heuristic	Utilization data	Skype traces	N	Y	Y	N
[33]	DT, RDF, MLP, BN	Utilization data	Real VNF workload	N	Y	Y	N
[41]	GNN	Utilization data and topology	Real VoIP workload	N	N	N	N
[44]	Q-learning	Utilization data	Real telco system	N	N	Y	N
[46]	Q-learning, GP	Utilization data	Synthetic VNF workload	N	N	Y	N
[47]	FL, Q-learning, SARSA	Utilization data	'98 world-cup, Wikipedia	Y	N	Y	N
[48]	DQN	Utilization data	Twitter analytics app	N	N	Y	N
[49]	Q-learning	Utilization data	Synthetic serverless app	Y	N	Y	N
Ours	ARIMA, LR, MLP, RNN	Utilization data	Synthetic and real CDN traces	Y	Y	Y	Y

or it was designed for specific applications; which data or use-case was used for validation; whether the work considered also overheads or delays related to spawning new instances; whether the work was actually aiming at realizing elasticity-control loops; and whether the implementation of the proposed solution was open-source, such that other researchers can reproduce the work and possibly improve it. Overall, ours is the only work providing an open-source architecture for extending the orchestration capabilities of OpenStack with predictive analytics, enablingforecast-driven decision-making for generic elastic cloud services. Also, our implementation is modular, such that it can easily be extended with custom models developed using established modeling frameworks for the Python language, such as Scikit-learn, Statsmodels, PyTorch, and TensorFlow.

3 Background

In this section, we provide useful background concepts to understand the approach proposed in Section 4. First, we provide details on a number of key OpenStack components, with reference to Fig. 2. Then, we recall fundamental background concepts around some time-series forecasting techniques we used in the experimental validation described in Section 5.

3.1 OpenStack component

3.1.1 Nova, Cinder, and Glance

Nova[50] is the OpenStack component that provides compute resources (e.g., VMs, bare metal servers, and containers) management functionalities. It leverages on Cinder[51] for block storage management and Glance[52] for image provisioning. The core of Nova's architecture is the compute process, which manages the underlying hypervisor (using libvirt). Such process communicates with the shared central database through the conductor process. Finally, the scheduler process is the interface between the compute process and the instance placement service. All processes exchange requests via remote procedure call (RPC).

3.1.2 Neutron

Neutron[53] is the OpenStack component that provides networking functionalities. It offers the possibility to

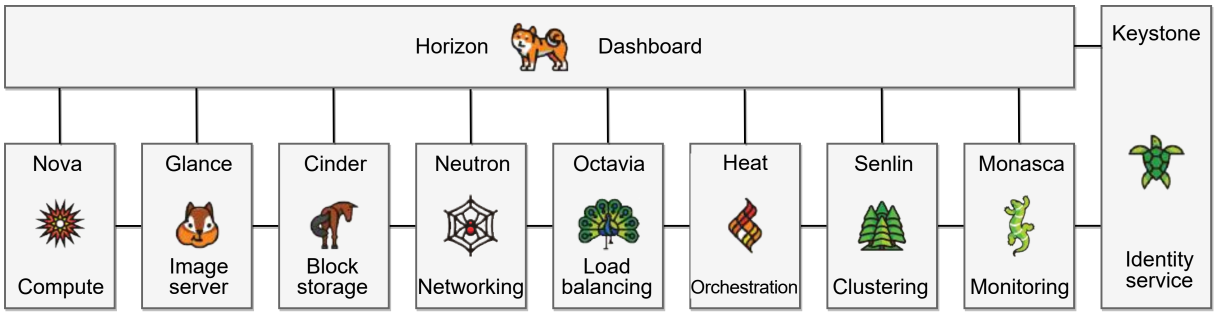


Fig. 2 Overview of OpenStack key components.

manage per-tenant virtual networks (e.g., having their own IP numbering and DHCP settings) and can be equipped with security-related features, like firewalls and virtual private networks (VPNs).

3.1.3 Monasca

Monasca[8] is an advanced multi-tenant, highly scalable, and fault-tolerant monitoring solution. It is designed as a collection of microservices, including an efficient time-series database (DB), a streaming alarm engine, a notification engine, a message queue, etc. Monasca also provides an agent module that is deployed on the physical machines hosting the compute services, such that it can collect metrics and forward them to the DB through the message queue. Monasca is also compatible with Kubernetes.

3.1.4 Senlin

Senlin[54] is the OpenStack component that offers tools to effectively operate clusters of homogeneous OpenStack resources (e.g., Nova instances). In particular, it is possible to define and attach policies to such clusters, specifying how their resources must be treated under specific conditions. For instance, one can use scaling policies to automatically resize the cluster, load-balancing policies to distribute the workloads, or health policies to handle faulty instances. Compared to Heat[55], Senlin offers more effective operation support tools and a finer-grained control over the underlying resources. Indeed, Senlin is being successfully used to operate large-scale deployments, like the on-line gaming use-case reported by Ref. [56].

3.1.5 Octavia

Octavia[57] is the former neutron load-balancing-as-a-service (LBaaS) and, as the name suggests, offers scalable load-balancing (LB) functionalities. LB is crucial to enable fundamental cloud properties like elasticity and high-availability. An Octavia LB consists of a horizontally-scalable pool of Nova instances (i.e., amphorae), leveraging on HAProxy. The controller is the core of Octavia's architecture, consisting in a number of sub-components whose jobs include handling requests and orchestrating the amphorae.

3.2 Time-series forecasting

In the empirical evaluation, we confront the performance of forecasting components leveraging different data-driven models. The dynamic nature of the data involved in cloud and distributed systems naturally calls for learning models which can effectively tackle time-series data, where observations are not assumed to be independent and identically distributed but are rather influenced by the sequential ordering in which they are observed. To this end, in the following we review two fundamental approaches for time-series forecasting, namely autoregressive models and recurrent neural networks. For the sake of completeness, in our empirical analysis we also consider baseline learning models which do not consider the sequential nature of time-series data, namely LR (see Ref. [9]) and MLPs (see Ref. [58]). The reader is referred to consolidated literature references for further details on such foundational models.

3.2.1 ARIMA

The autoregressive moving average (ARMA) model is an effective tool for time-series forecasting (see Ref. [10]). Given the observations $\{x_t\}$ up to time $t-1$, the forecast $\hat{x}_t$ is given by

$$\hat{x}_t = \phi_0 + \sum_{i=1}^{p} \phi_i x_{t-i} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j} \tag{1}$$

where ϕ_0 is a constant typically set to 0 if there is no evidence of a trend in the input data, otherwise, it can be manually set based on domain knowledge or automatically learned; $\{\phi_i\}_{i=1}^{p}$ are the learnable parameters regulating the linear dependency between $\hat{x}_t$ and the most recent p observations; and $\{\theta_j\}_{j=1}^{q}$ are the learnable parameters that regulate the linear dependency between $\hat{x}_t$ and the q errors $\left\{\epsilon_{t-j} \triangleq \hat{x}_{t-j} - x_{t-j}\right\}_{j=1}^{q}$ that the model made in the most recent q predictions.

The ARIMA model is an extension of ARMA that can deal with non-stationary time-series (see Ref. [10]). Indeed, given a $d \in \mathbb{N}^+$, ARIMA is obtained by applying ARMA to estimate $x_t^{(d)}$, which is the d-order differenced signal obtained from x_t as $x_t^{(1)} \triangleq x_t - x_{t-1}$ for $d = 1$, and $x_t^{(d)} \triangleq x_t^{(d-1)} - x_{t-1}^{(d-1)}$ for $d > 1$. The resulting forecasts $\hat{x}_t^{(d)}$ for the differenced signal

$$\hat{x}_t^{(d)} = \phi_0 + \sum_{i=1}^{p} \phi_i x_{t-i}^{(d)} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j}^{(d)} \tag{2}$$

are cumulatively summed up to reconstruct the forecast of the original signal $\hat{x}_t$. Given that ARIMA is characterized by the 3 meta-parameters p, d, and q, Eq. (2) is conventionally referred to as ARIMA(p,d,q). The meta-parameter d is typically chosen such that the resulting the d-order differenced signal $x_t^{(d)}$ is stationary (i.e., whose auto-correlation, mean, and variance are independent of t).

3.2.2 Recurrent neural networks

Among artificial neural networks (ANNs), RNNs are commonly used for multi-variate time-series analysis (see Ref. [58]) and forecasting in particular. RNNs predict the one-step (or k-step) ahead value of a time-series based on the current I-dimensional input $x_t \in \mathbb{R}^I$ and a compressed history of the inputs, stored in an H-dimensional state vector $s \in \mathbb{R}^H$ computed recurrently by the hidden neurons. The model evolution is described by

$$s_t = f_s(s_{t-1}, x_t) \tag{3}$$

$$o_t = f_o(s_t, x_t) \tag{4}$$

where (1) $f_s : \mathbb{R}^{H+I} \to \mathbb{R}^H$ operates on the concatenation c of s and x, and is defined, e.g., as $f_s = \tanh(W_s c + b_s)$, where W_s is the weight, and b_s is the bias; (2) f_o : $\mathbb{R}^{H+I} \to \mathbb{R}^O$ is the output function, similarly defined, e.g., as $f_o = \mathrm{ReLU}(W_o c + b_o)$, where ReLU denotes the rectified linear unit function. The learnable parameters $\theta = \{\theta_j\} = \{W_s, W_o, b_s, b_o\}$ are typically trained through gradient descent on the loss function. In this work, we consider a stochastic update rule with momentum, so that the j-th parameter is updated at the k-th optimization step as follows:

$$\mu_{j,k} = \beta \mu_{j,k-1} + \nabla J_{\theta_{j,k}}(D) \tag{5}$$

$$\theta_{j,k+1} = \theta_{j,k} - \lambda \mu_{j,k} \tag{6}$$

where D is a dataset of input-output pairs, and $\nabla J_{\theta_{j,k}}(D)$ is the gradient of the loss function $J_{\theta_{j,k}}(D)$ with respect to parameter θ_j computed at step k. The term β determines in which proportion the momentum $\mu_{j,k}$ is applied during the gradient descent step, and λ is the learning rate. The training process continues until the validation loss, computed over the validation dataset every K optimization steps, stops decreasing. The model corresponding to the minimum achieved on the validation loss is taken as output. A particular type of RNNs is the so-called LSTM networks[59]. LSTM is an improved version of the standard RNN architecture, where the so-called input, output, and forgetting gates regulate the amount of information flowing into the hidden-state representation. In this way, LSTM is able to capture both long- and short-term dependencies among the input variables, also reducing the risk of incurring in numerical instability problems, such as the vanishing gradient[60].

4 Proposed Approach

As mentioned in Section 1, conventional scaling techniques are reactive, as they adjust resources when certain metrics breach specific threshold values. They may be configured using a cautionary approach, triggering on very early degradation signs. Or, they may use an optimistic approach by having thresholds very close to critical values. A cautionary approach is mostly necessary when scaling operations need a significant amount of time to become effective. However, the risk is, for instance, to waste resources (i.e., over-provisioning) due to unnecessary scale-out decisions. On the other hand, an optimistic approach limits resource waste but can lead to affecting the QoS perceived by users in case scaling operations do not take effect before the system saturates its current capacity. Our approach mitigates the aforementioned issues by adopting a predictive auto-scaling strategy that triggers scaling actions on the basis of forecasts of one or more target metrics. For instance, given an imminent growth in the flow of requests, our approach allows for triggering a scale-out sufficiently ahead of time. We implemented our approach in OpenStack, specifically extending the Monasca monitoring system. As shown in Fig. 3, we assume that the orchestration is performed by Senlin leveraging on our forecasting component (plus the required Monasca resources) to

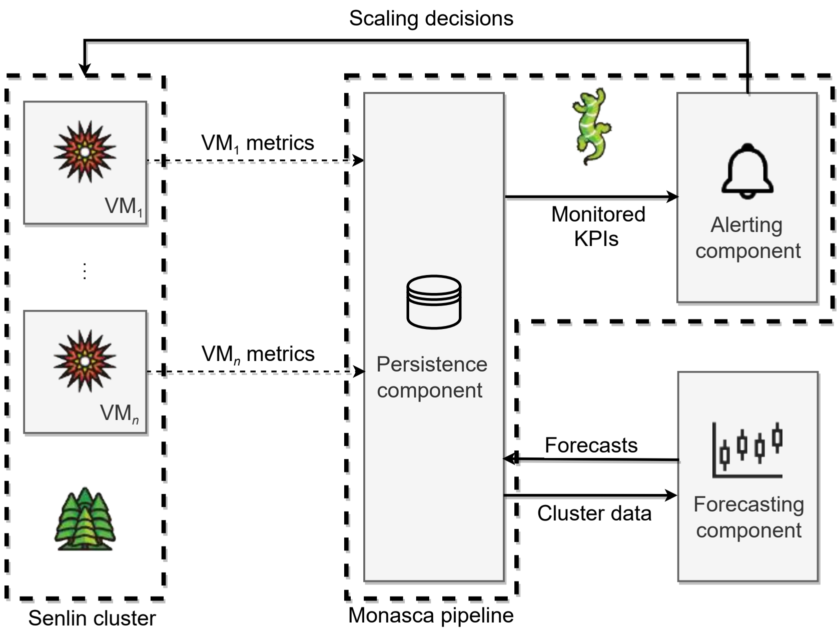


Fig. 3 Architectural diagram of the proposed predictive auto-scaling approach.

integrate predictions in the scaling operations. We consider a cluster of Nova VMs as the specific compute instances to be elastically scaled. However, notice that our approach is agnostic with respect to such implementation details. Although there exist two main scaling strategies, horizontal and vertical (see Ref. [1]), our experimentation focuses on horizontal scaling only. This is definitely the most used strategy in cloud computing, as it is simpler to manage and supports scaling out to an arbitrary number of instances, differently from vertical scaling, which is limited by the specific capabilities of the underlying hardware. Indeed, modern cloud applications are typically designed such that their components can be highly distributed and replicated among different nodes to enhance both availability and resiliency. However, our proposed architecture may easily be leveraged for vertical scaling[61] as well, as our approach is agnostic with respect to how the actual scaling operations are implemented.

Our approach works as follows. VMs periodically generate system-level metrics, which are ingested by Monasca. Such metrics are indicators for the current load sustained by the system. Such metrics are basically a set of time series, which is the input to our forecasting component. Such data are periodically fetched by the forecasting component, whose task is to generate forecasts of the input metrics over a given time window. The frequency and the time window of the forecasts, as well as the amount of historical data to be provided as input, are configurable. The generated forecasts are finally persisted and become available to all the components of the infrastructure management system via Monasca APIs. In particular, they can also be visualized by operators through dashboards (e.g., Grafana). In our case, forecasts are fed to a threshold-based scaling policy, and scale-out actions are triggered as soon as the predicted values of the considered KPIs breach the threshold for a given number of subsequent observations. Threshold checks are performed via the Monasca alerting pipeline, and the scaling operations are actuated by Senlin.

The underlying predictors generally need to be trained before use and constantly updated in case of dynamic changes in the statistic behavior of the time-series[62]. Based on the specifics of the metric dynamics, this may require longer or shorter training histories and periodicity of updates. In this work, for simplicity we assume that training is performed offline, but the integration of automatic model updates is planned to be handled in a short future. As discussed in Section 5, we compare four different predictors, i.e., a linear regressor, an ARIMA model (see Section 3.2.1), an MLP, and an RNN (see Section 3.2.2).

Implementation details

Our forecasting component, also known as Monasca-predictor, is implemented in Python, the main programming language used in the OpenStack ecosystem, and released under the Apache 2.0 open-source license[63]. Such component is realized in compliance with the microservice-oriented architectural pattern used by Monasca. In particular, it is designed to be eventually integrated into the Monasca-agent (see Section 3.1.3).

Monasca-predictor is configured using a YAML file similar to the one shown in Fig. 4. The Api block contains the information required to make authenticated calls to OpenStack APIs. As Monasca-predictor performs tasks similar to those of the Monasca-agent, it must be provided with similar permissions. The Main block contains the configurations related to the actual predictive tasks of Monasca-predictor. Notice that the forwarder_url and hostname fields must be filled with the pointers to the forwarder process of the Monasca-agent. The inference_frequency_seconds field specifies the frequency at which forecasted values must be generated. The predictions field is a list containing the individual configurations for the different forecasts. For each item in the list, additional fields can be specified as follows. The tenant_id field must be filled with ID of the OpenStack project containing the resources to be monitored. The dimensions field is a map that specifies additional properties required to identify such resources (e.g., the ID of the elastic group

```
Api:
  # Fill with configs similar to monasca-agent-forwarder
Main:
  forwarder_url: ...
  hostname: ...
  inference_frequency_seconds: 60
  predictions:
      -tenant_id: # Fill with OpenStack project ID
       dimensions:
         scale_group: # Fill with scaling group ID
       metrics: [cpu. utilization_perc]
       group_by: ["*"]
       merge_metrics: false
       time_aggregation_statistics: [avg]
       time_aggregation_period_seconds: 60
       space_aggregation_statistics: [sum]
       lookback_period_seconds: 1200
       prediction_offset_seconds: 900
       out_metric: pred. group. sum. cpu. utilization_perc
       model_path: /path/to/model. dump
       scaler_path: /path/to/scaler. dump
Logging:
  enable_logrotate: true
  disable_file_logging: false
  predictor_log_file: /path/to/predictor.log
  log_level: INFO
```

Fig. 4 Example of forecasting component config file.

of compute instances). The metrics field is a list of the metrics whose measurements are to be used as input to the predictor. In Monasca, a metric is identified by its name (e.g., cpu.utilization_perc) and the set of properties (i.e., dimensions) of the resource that generates measurements for said metric (e.g., resource_id, hostname, etc.). The group_by field is a list of such properties to be used by Monasca API for grouping measurements in different time-series, when fetching data by metric name. For instance, one could simply specify resource_id (or even *, standing for "all fields") to group measurements by resource. Depending on the boolean value specified for the merge_metrics field, the resulting time-series can also be merged into a single one, with measurements ordered according to their timestamp. The time_aggregation_statistics field is a list of operators (e.g., avg, sum, and max) to be applied on the retrieved time-series individually, binning their measurements according to the resolution specified with the time_aggregation_period_seconds field, such that the result is possibly a multivariate—resampled—time-series. Similarly, the space_aggregation_statistics field is a list of operators to be applied, on top of the temporal aggregation result, in order to aggregate the measurements of a set of monitored resources (e.g., the compute instances belonging to the same elastic group). The lookback_period_seconds field defines the time window that measurements must fall into for them to be included in a Monasca API response, as a (backward) difference with respect to the current timestamp. The prediction_offset_seconds field defines the time window of the forecast as a (forward) difference with respect to the timestamp of the most recent measurement. Notice that, when persisted, a forecast is associated with the very same timestamp, such that the forecasted metric appears backward-shifted by prediction_offset_seconds with respect to the input metric. The out_metric field is the metric name to be associated with the generated forecasted values. The model_path field is the path to the dump of the underlying predictive model. Similarly, the scaler_path field is the path to the dump of the scaler to be used for pre-processing the input data. At the moment, monasca-predictor only supports models built using the following frameworks: Scikit-learn[64], Statsmodels[65], PyTorch[66], and TensorFlow[67]. Finally, the Logging block allows for configuring logs management. A thorough explanation of the available tunables can also be found in the documentation within the code of our forecasting component[63].

Monasca-predictor was developed as a prototype that users can only configure via the YAML configuration file (see Fig. 4). Such limitation implies that a user needs full administrator privileges for the OpenStack deployment, as such file contains both system-wide (e.g., authentication credentials) and application-specific configurations (e.g., the details of the resources to be monitored). We plan to improve the usability of our component by separating such different types of configurations, so that multiple—possibly unprivileged—users can leverage on the same forecasting capabilities (even though the installation must still be performed by an OpenStack administrator). Similar to what Monasca and the other OpenStack projects offer, our idea is to develop both a command-line interface and a heat orchestration template (HOT) integration for monasca-predictor, such that users can manage their (application-specific) configurations in the way they see fit. Independently of the chosen interface, such configurations will be eventually persisted in a database (e.g., the MySQL instance included in any OpenStack deployment) to improve the reliability of our component.

5 Experiment

This section includes the results of an experimental validation of the approach described in Section 4. It provides a comparison of the performance of several predictive scaling policies with a traditional reactive one, considering a synthetic elastic application deployed on OpenStack.

5.1 Synthetic elastic application

We use distwalk[68], an open-source distributed processing emulation tool developed by us, to test the proposed predictive auto-scaling approach. The tool consists of a server module that accepts TCP/IP connections from one or more clients. Clients can request the server to perform computational, networking, and/or input/output (I/O) activities, enclosing within each request the amounts of resources to be consumed. Clients can submit requests with constant or exponentially distributed inter-arrival time, payload sizes, or I/O transfer sizes. Also, they may emulate ramp-up/ramp-down scenarios or use a file trace specifying the requests rate to be submitted over a time horizon. For instance, we use this feature to replay traces from a real CDN workload (see Section 5.5). Per-request round-trip response-time can be measured and reported in a log file on experiment termination. Clients can also spawn multiple threads submitting traffic in parallel, and they can emulate the establishment of multiple sessions, by closing and re-establishing their TCP/IP connections.

5.2 Experimental set-up

Our OpenStack deployment is hosted on a Dell R630 dual-socket test-bed, equipped with 2 Intel Xeon E52640 v4 CPUs (each having 10 hyper-threaded cores, i.e., 20 hyper-threads) running at 2.40 GHz (with a turbo-boosting frequency of 3.40 GHz); 64 GB of RAM; Ubuntu 20.04.2 LTS operating system; and Version 5.4.0 of the Linux kernel. We use an all-in one OpenStack deployment (Victoria release), installed using the tools provided by Kolla[69], resulting in each service being operated within Docker containers. As detailed in Section 4, we deploy an elastic control loop using the following services: (1) Senlin to orchestrate a horizontally-scalable cluster of Nova VMs; (2) Octavia to provide the cluster with load balancing capabilities; (3) Monasca to ingest the system-level metrics and to trigger the scaling actions; and (4) the forecasting component, developed by us, to enable the predictive auto-scaling strategy.

The Senlin cluster has a minimum of 2 active instances and could expand up to 5. Each instance is provided with 1 vCPU and 2 GB of RAM available and with an Ubuntu 20.04 cloud image including the server module of distwalk (see Section 5.1). We artificially impose a delay of 10 min before starting the distwalk server. The purpose is to emulate a scenario with non-negligible set-up times for new instances, as it may be needed in real cloud workloads, where it is commonplace that spawning new instances may take from a few minutes to even half an hour[3]. In such a scenario, performing scale-out operations well ahead of time becomes critical. The application server instances are made reachable through an Octavia LB, set to distribute the traffic according to a round-robin strategy. The distwalk client is configured to spawn 6 threads, such that in the beginning of our runs, each server instance has to handle the aggregated requests from 3 threads on average. Each thread follows a 4 h long trace reporting the operation rates (i.e., requestsper second) to be maintained for an interval of 1 min each. Also, each thread is set to break its work in

1000 sessions, such that a new connection is opened with the LB every 15 s, allowing for it to select a new target instance. Monasca is set such that new measurements are collected each minute. The forecasting component is set to output a new prediction with the same interval, leveraging on the most recent measurements. The input to the underlying forecasting model consists in a time-series reporting the sum of the CPU usage measurements of the currently active instances. The output of the model is the estimated value of the same time-series in 15 min (i.e., 50% more than how long a new instance takes to activate). Such output is then divided by the number of currently active instances to get an estimate of the average CPU usage expected in 15 min, assuming the cluster size to remain constant and then persisted in Monasca.

The purpose of our experimentation is to show the effectiveness of the proposed architecture and not to evaluate a novel ML model for time-series forecasting that can outperform existing predictive elasticity approaches. Indeed, the novelty of our work consists in proposing an open-source scalable architecture, compatible with Monasca, which can be easily configured by practitioners to plug virtually any type of time-series forecasting model into their data-driven control loops. Therefore, we decide to stick to a simple example where the elasticity controller uses only CPU usage as input. However, our component can be configured to predict multiple metrics per monitored instance and/or perform multi-variate time-series forecasting.

To implement the predictive scaling strategy, Monasca is set to trigger a scale-out whenever the predicted average CPU usage of the cluster reached 80% for 3 times in a row. On the other hand, to implement the reactive scaling strategy, it is set to do the same but consider the actual average CPU usage. In any case, a scale-in is triggered whenever the actual average CPU usage reached 15% for 3 times in a row. We choose not to use the predicted metric to decide whether to trigger scale-in actions. While a cloud provider may want to anticipate traffic peaks by spawning additional resources in advance, disposing of superfluous resources can be much quicker[3] and is typically done after making sure that all residual traffic has been drained from them. Otherwise, the risk is to overload the remaining instances in case they start taking the traffic relieving the being-released instance too early. Predicting such a condition is hard, and in practice, it is often more convenient to minimize service-level agreement (SLA) violations, rather than costs. However, our framework does not exclude this possibility. Waiting for 3 consecutive violations imposes a delay of at least 3 min for an action to be triggered. However, this is a well established practice in elasticity control loop design, as it helps with making the mechanism more robust to fluctuations. Each scaling action could adjust the size of the cluster by 1 instance only and could only take effect if it is triggered after a cooldown period of 10 min since the last scaling action. The cooldown is also useful considering the 10-min delay forcibly added before new instances activate.

5.3 Predictors configuration

To implement the underlying forecasting models, we use: (1) Scikit-learn for the LR; (2) Statsmodels for ARIMA; and (3) PyTorch for MLP and RNN. To evaluate how the amount of past information given as input influences the prediction, we consider 3 different settings, i.e., 5, 10, and 20 min worth of measurements (see Section 5.4). Apart from LR, which is fitted every time on a different input, all models are trained offline on the same synthetic dataset (and on the same machine where OpenStack is deployed). Such data consist of sinusoidal traffic patterns, with different frequencies and amplitudes, to provide models with expected behaviors for a wide range of operational modes (the dataset is open-source, see Section 5.6). Note that we do not conduct an optimal hyperparameters search, as we believe such a process goes beyond the scope of this work, whose focus is the integration of time-series forecasting techniques in the elasticity-control loop infrastructure, rather than ML models development. However, in what follows we provide some indications on why we take specific design choices, aiming at conducting a fair comparison among the implemented models.

ARIMA meta-parameters are configured such that $p = \{5, 10, 20\}$, $d = 1$, and $q = 0$. While p is somewhat constrained by the input size, we choose $d = 1$ for the stationarity assumptions, and $q = 0$ as we do not observe any benefit from using this feature of the model. For ARIMA, we observe an average training time of 2.89 s. MLP consists of an input layer (with units varying in $\{5, 10, 20\}$, as per the input size constraints), two hidden layers of 10 units each (as we want to keep the complexity low), and an output layer of 1 unit (as we needed a scalar output). We also apply a leaky ReLU non-linearity to the output of each hidden layers, as it generally speeds up the training. MLP is trained with stochastic gradient descent (SGD) for 1000 iterations, using a decaying learning rate (between 0.1 and 0.001), a momentum set to 0.8, and a batch size of 500 input samples. These values are generally considered sensible defaults, and given the observed performance, we do not feel the need to fine-tune them. The 3 considered variants of MLP consist of 181, 231, and 331 learnable parameters, when the input layer size is set to 5, 10, and 20, respectively. For MLP, we observe an average training time of 136.15 s. RNN consists of 3 (stacked) recurrent layers, each one composed by 200 units and using ReLU as activation function (i.e., PyTorch's defaults), followed by a fully-connected layer of 1 unit (as we need a scalar output). Such model is trained with SGD for 10 000 iterations, using a decaying learning rate (between 0.01 and 0.001), a momentum set to 0.5, and a batch size of 300 input samples. With respect to MLP, we have to fine-tune the latter parameters for the model to converge to an acceptable performance. RNN consists of 201 601 learnable parameters, independently of the size of the input sequences. For RNN, we observe an average training time of 436.07 s.

To facilitate comparison between models, we choose not to leverage on the capability of RNN to handle variable-length sequences and train it using fixed-length input sequences, like the other models. Also, during the runs, all forecasting models are re-loaded from disk each time they have to be queried (i.e., once per minute). In this way, we do not leverage on the hidden state of RNN to be updated after each query, which should theoretically allow the model to retain the observed dynamics and allow for computing forecasts even when provided with just a single new measurement as input.

5.4 Validation on synthetic workload

In this section we report the results obtained by applying five different scaling strategies to a synthetic workload similar to the one depicted in Fig. 1. Distwalk is set such that the average CPU usage of the cluster ramps up twice during a single run: first, with a rather soft slope, peaking at 70% (around the 60th minute) and progressively fading out until the 120th minute; then, with a much steeper slope, (theoretically) peaking at 120%, exceeding the cluster capacity. The first peak is designed to expose the behavior of a scaling strategy when facing a workload that might lead to saturation but, instead, decreases before reaching the threshold (80%). In that case, a classical strategy would not react, whereas a predictive one may inaccurately forecast the evolution of the workload and trigger unnecessary actions. This scenario is useful to assess how sensitive to fluctuations and, thus, how prone to yielding false alarms a strategy is.

In what follows, in the CPU usage plots (e.g., Fig. 5a), the blue curve represents the workload that each distwalk thread exercises on the cluster (i.e., the ideal usage we would observe if a single thread submitted requests to a single VM). As requests are submitted through the LB, the result is that, eventually, each VM in the cluster handles an equal share of the cumulative workload (see Section 5.2). In other words, the resource consumption curves do not closely follow the blue curve because, in the beginning of each run, there are 6 distwalk threads submitting requests to a total of 2 VMs through the LB. Therefore, each VM initially handles the aggregated requests coming (on average) from 3 threads. Instead, the red curve in Fig. 6a (left) refers to the predicted average CPU usage of the cluster, assuming the size of the cluster to remain constant. On the other hand, client-side response time plots (Fig. 5b) provide a view of the system

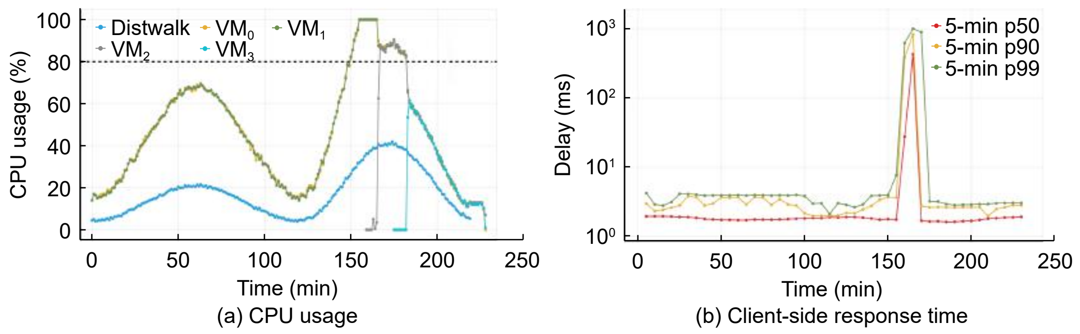
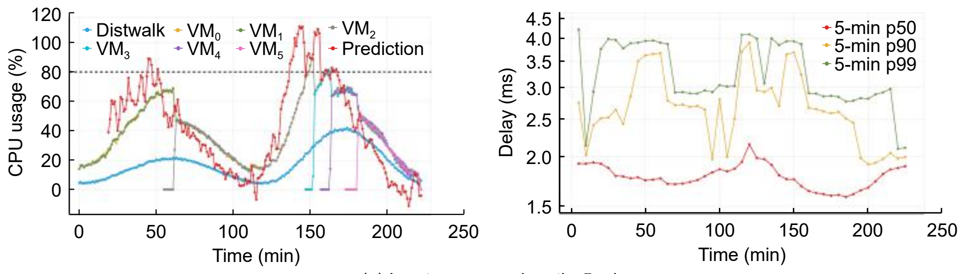
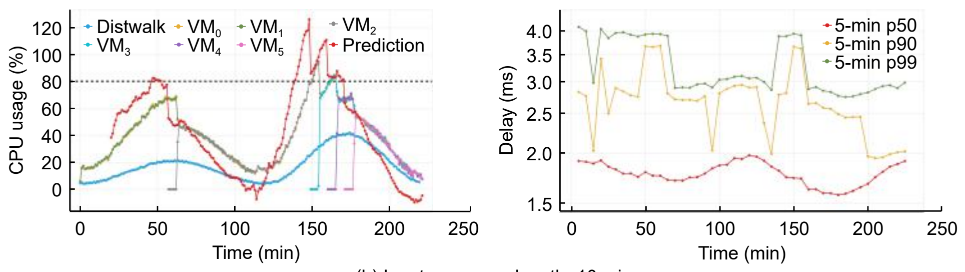


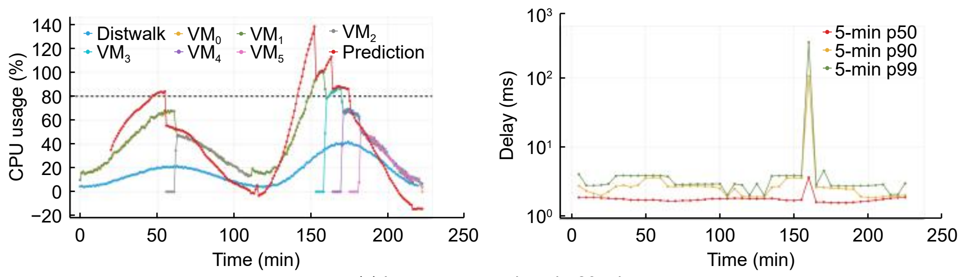
Fig. 5 Experimental results for the traditional (static) scaling policy.



(a) Input sequence length=5 min



(b) Input sequence length=10 min



(c) Input sequence length=20 min

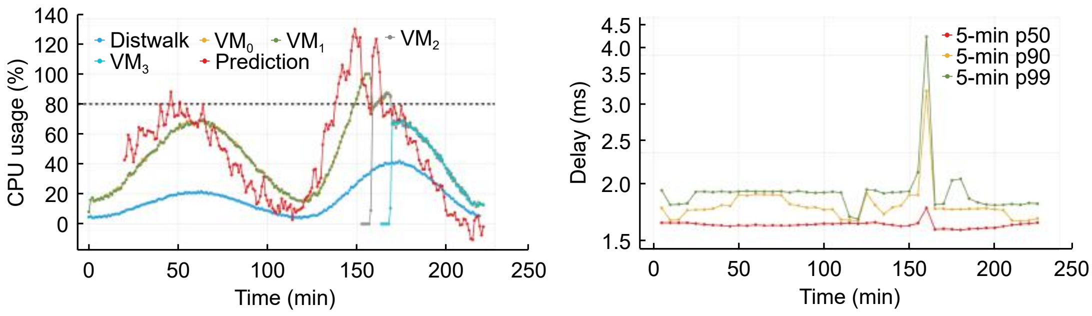
Fig. 6 Experimental results for the LR-based scaling policy (left: CPU usage; right: client-side response time). Note that the sub-pictures on the right side use a log-scale *Y* axis for the delay.

performance as observed by the distwalk client. Each plot reports curves that show (on the *Y* axis) the evolution of some response-time statistics (median, the 90th percentile (p90), and the 99th percentile (p99)) over time (on the *X* axis), where each data point refers to a statistic aggregated over the data of a moving window of the previous 5 min.

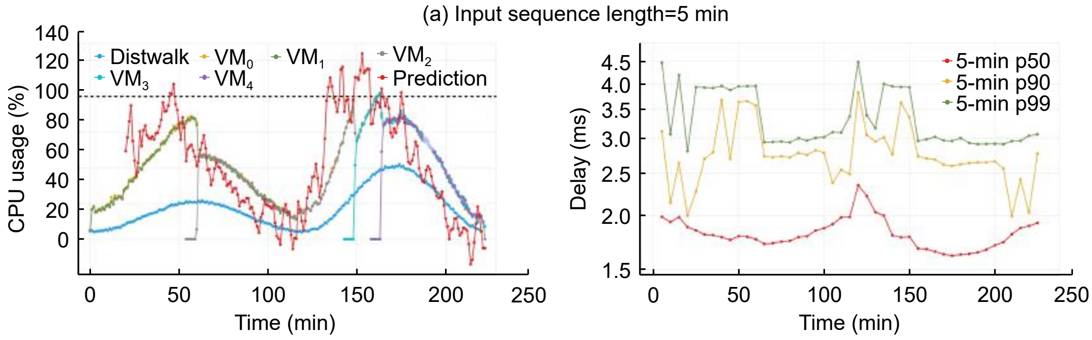
As shown in Fig. 5a, the static scaling strategy fails to scale-out the cluster on time when facing the second peak. Starting from the 155th minute, it is possible to observe the system saturating its capacity (i.e., the CPU usage is 100%) and remaining in such a state for 10 min. In the meantime, the requests submitted to the cluster pile up and the client-side response time starts growing up to 1 s, as visible in Fig. 5b. Such a performance degradation occurs because the static strategy waits for 3 consecutive violations of the threshold before triggering the scale-out. Furthermore, due to the artificial set-up delay, the new VM takes 10 min before starting to serve requests. Therefore, while the scale-out decision happens approximately at the 152nd minute, the new VM starts responding only approximately at the 166th minute and only for new established sessions (occurring every 15 s, see Section 5.2). Such a scenario exposes the need for more intelligent strategies that are able to take scaling decisions ahead of time. We use 4 different time-series forecasting algorithms to implement different predictive scaling strategies, namely, LR, ARIMA, MLP, and RNN. For each strategy, we consider 3 different values for the amount of past information to be fed to the underlying model (i.e., 5, 10, and 20 measurements, with minute granularity) when computing an estimate of the average CPU usage in 15 min. In this way, we could assess how sensitive to the size of the input the predictive capability of a given model is.

Figure 6 reports the results obtained using an LR-based policy, which generally performs a better job than the static policy at scaling the cluster before the second peak, as it correctly predicts the growth in the CPU usage. However, this predictor also tends to be overly sensitive to the noise in the input and to overestimate. As a consequence, in all the runs, the LR-based policy triggers an unnecessary scaling action also before the first peak. For instance, when the input size is set to 5 min (see Fig. 6a), even though the cluster scales on time before the second peak, the policy seems to be overly sensitive to even small variations in the input, such that the resulting prediction is very noisy and, thus, not reliable in general. On the other hand, when the input size is set to 20 min (see Fig. 6c), the cluster reaches 100% CPU usage for 3 min (with client-side response time going up to 100 ms), before the scaling action takes effect (around the 159th minute). This is due to the input size being too large, such that the LR cannot detect the growth soon enough. In other words, at the beginning of the ramp-up, the contribution of the newer measurements is outweighed by the older ones, generating a sort of momentum that delays the detection of the peak. We instead observe a nice behavior when reducing the input size to 10 min (see Fig. 6b), with the scaling action taking effect when the average CPU usage is at 95% (around the 155th minute) and client-side response time going up to 3 ms.

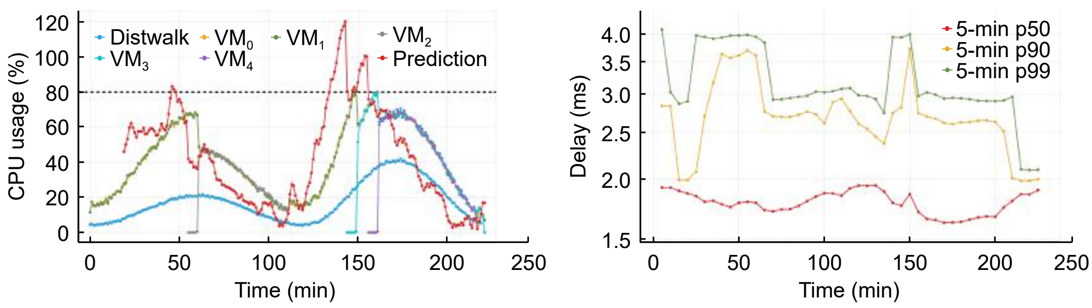
Figure 7 shows the results obtained with an ARIMA-based policy. Similar to the LR-based one, this policy seems to be generally overly sensitive to small variations in the input. In some cases (see Figs. 7b and 7c), such behavior generates unnecessary scaling actions before the first peak. When the input size is set to 10 and 20 min (see Figs. 6b and 6c), the cluster is successfully scaled before the second peak, with the scaling action taking effect around the 150th minute. However, when the input size is set to 10 min, the



(a) Input sequence length=5 min



(b) Input sequence length=10 min

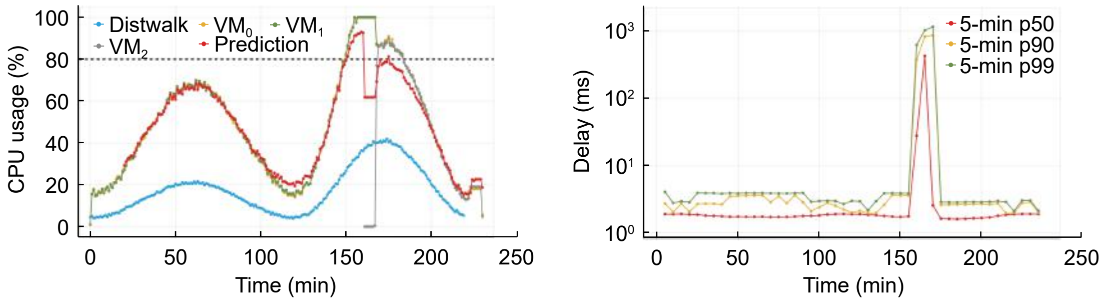


(c) Input sequence length=20 min

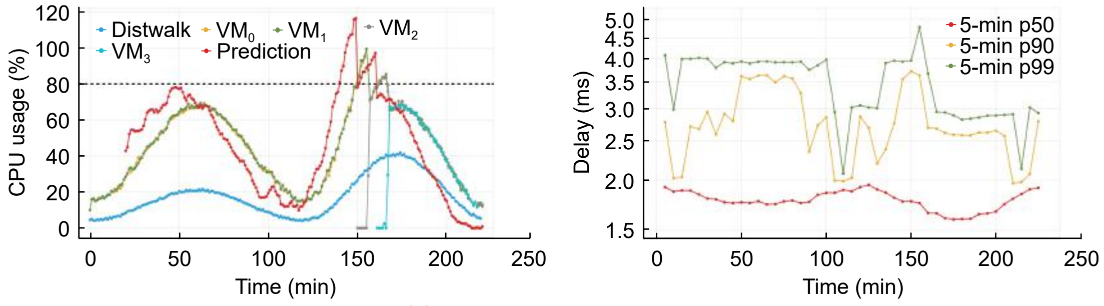
Fig. 7 Experimental results for the ARIMA-based scaling policy (left: CPU usage; right: client-side response time). Note that the sub-pictures on the right side use a log-scale *Y* axis for the delay.

behavior of the policy appears significantly more noisy. On the other hand, when the input size is set to 5 min (see Fig. 6a) the cluster reaches 100% CPU for 3 min (with client-side response time going up to 100 ms), before the scaling action takes effect (around the 159th minute).

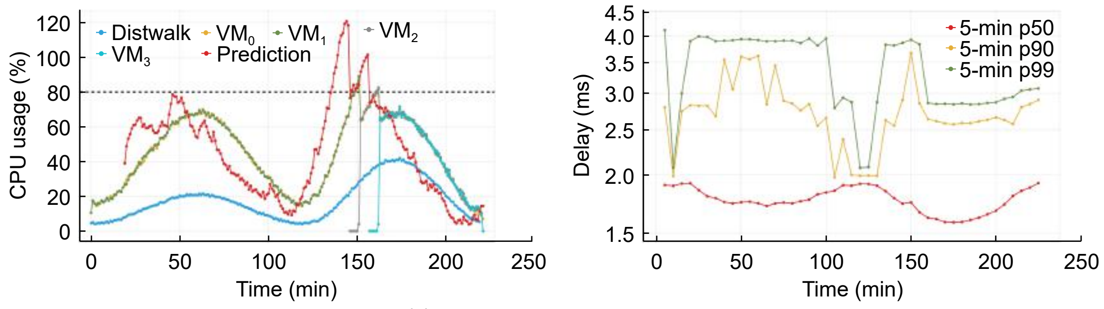
Figures 8 and 9 report the results of applying the MLP- and RNN-based policies, respectively. In contrast to the LR-based one, it is straightforward that the larger the input size, the better the overall performance. Setting the input size to 5 min (see Figs. 8a and 9a) results in obtaining a policy that is equivalent to the static one (see Fig. 5). As it is the case for the static policy, the inability to anticipate the second peak leads to a saturation of the system capacity that persists for 10 min, with client-side response time growing up to 1 s. On the other hand, both MLP and RNN behave nicely with input size set to either 10 or 20 min. When the input size is set to 10 min (see Figs. 8b and 9b), both policies scale the cluster just in time to prevent saturation, as the actions take effect around the 155th minute, when the average CPU usage is at 99%. However, there is no sign of performance degradation from the client perspective, as the response time stays below 4 ms. Also, both policies scale the cluster earlier when the input size is set to 20 min (see Figs. 8c and 9c), even though the predictions appear more noisy (that could lead to unexpected behaviors in other circumstances). In both cases, the scaling action takes effect around the 151st minute and the client-side response time stay below 3 ms. However, for the MLP-based policy the scaling action takes effect when the average CPU usage is at 89%, while for the RNN-based one the same happens at 80%. Such difference is likely the result of random fluctuations in the measured load. Remarkably, the RNN-based policy triggers an



(a) Input sequence length=5 min

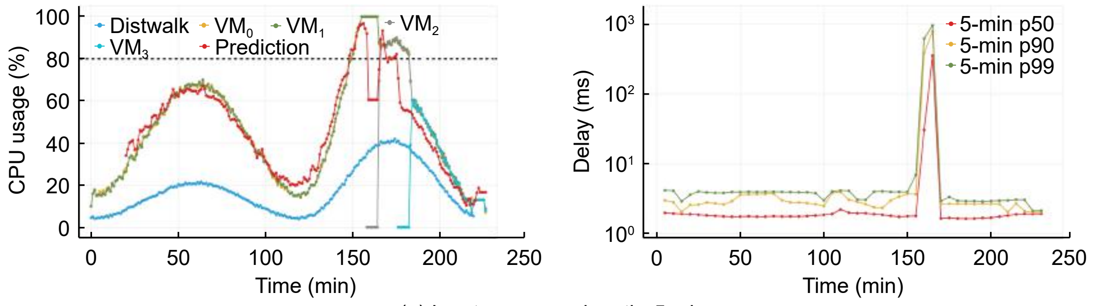


(b) Input sequence length=10 min

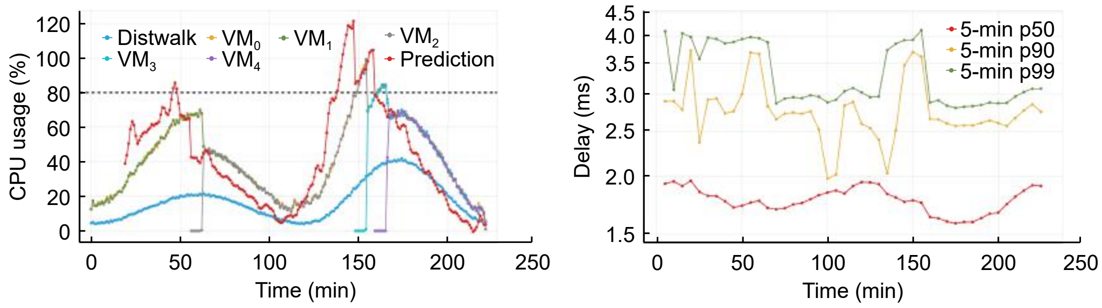


(c) Input sequence length=20 min

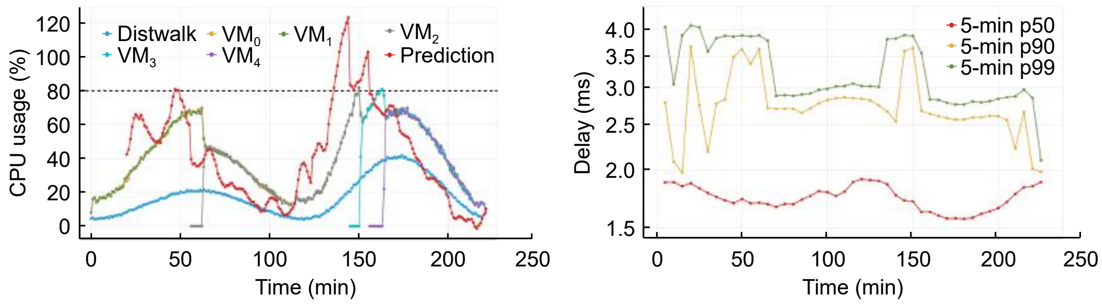
Fig. 8 Experimental results for the MLP-based scaling policy (left: CPU usage; right: client-side response time). Note that the sub-pictures on the right side use a log-scale *Y* axis for the delay.



(a) Input sequence length=5 min



(b) Input sequence length=10 min



(c) Input sequence length=20 min

Fig. 9 Experimental results for the RNN-based scaling policy (left: CPU usage; right: client-side response time). Note that the sub-pictures on the right side use a log-scale *Y* axis for the delay.

unnecessary scale-out before the first peak, as the predicted average CPU usage exceeds the threshold for the exact amount of time that is required. The same happens when using the ARIMA-based policy. Such behavior exposes the need for properly tuning, beside the specific forecasting model, also the other components of the scaling strategy. For instance, we could make the RNN-based policy more robust by increasing the number of breaches to the threshold required to trigger the action. Automatically adjusting a broader set of tunables (e.g., cooldown period, scaling adjustment, alarm thresholds, etc.) is among the engineering issues to be addressed in future extensions of the proposed architecture. Approaches based on neural networks are, in general, able to capture even fairly complex non-linear relations. However, in this case, an input size of 5 min is clearly not enough to provide such models with the information required to output a 15-min forecast.

Table 2 reports the mean average percentage error (MAPE) made by each predictor configuration during our runs. The MAPE was computed by considering the sum of the CPU usage of all VMs, as ground truth, and the predicted values multiplied by the number of active VMs at each specific point in time (i.e., the predicted sum of the CPU usage of all VMs). Such results further support our conclusions regarding which configuration is the best for each predictor. For instance, we can see that LR performs better when the input is set to 5 or 10 min (although the former leads to a very sensitive scaling policy). Conversely, in general, the bigger the input, the better the performance of the other predictors.

Tables 3 and 4 report the average and percentiles of

Table 2 Prediction errors (MAPE) observed for the considered runs.

Policy	MAPE		
	Input sequence length=5 min	Input sequence length=10 min	Input sequence length=20 min
LR	0.25	0.29	0.38
ARIMA	0.22	0.26	0.15
MLP	0.52	0.18	0.14
RNN	0.44	0.18	0.15

Table 3 Descriptive statistics of the client-side response time observed during the experimental runs, when the cluster is facing the first peak (0th–120th minutes) in CPU usage.

Policy	Response time (ms)						
	Avg	p50	p90	p95	p99	p99.5	p99.9
Static	2.12	1.81	3.18	3.73	3.96	4.08	8.12
LR (5)	3.30	1.79	2.76	3.56	3.92	4.03	9.41
LR (10)	2.02	1.79	2.81	3.57	3.90	3.99	6.59
LR (20)	1.95	1.75	2.72	2.95	3.82	3.90	5.89
ARIMA (5)	2.07	1.82	2.90	3.74	3.95	4.03	7.19
ARIMA (10)	2.03	1.80	2.78	3.16	3.92	4.17	7.79
ARIMA (20)	2.02	1.80	2.79	3.49	3.91	4.02	8.52
MLP (5)	2.03	1.80	2.82	3.68	3.90	3.97	6.88
MLP (10)	2.05	1.80	2.89	3.71	3.93	4.01	7.47
MLP (20)	2.04	1.80	2.81	3.69	3.91	4.00	7.65
RNN (5)	2.07	1.79	2.97	3.72	3.94	4.04	7.78
RNN (10)	2.01	1.79	2.79	3.57	3.90	4.00	7.80
RNN (20)	2.72	1.74	2.77	3.50	3.87	3.99	14.36

Table 4 Descriptive statistics of the client-side response time observed during the experimental runs, when the cluster is facing the second peak (121st–220th minutes) in CPU usage.

Policy	Response time (ms)						
	Avg	p50	p90	p95	p99	p99.5	p99.9
Static	47.06	1.80	58.12	394.05	824.53	924.59	1050.76
LR (5)	1.89	1.67	2.63	2.78	3.79	3.92	7.82
LR (10)	1.88	1.67	2.60	2.75	3.77	3.91	7.69
LR (20)	4.90	1.66	2.69	3.67	84.76	245.82	381.07
ARIMA (5)	3.42	1.72	2.76	3.72	39.48	63.80	298.10
ARIMA (10)	1.96	1.70	2.68	2.81	3.80	4.48	9.10
ARIMA (20)	1.97	1.70	2.66	2.78	3.81	4.51	11.71
MLP (5)	63.63	1.84	198.39	522.05	953.45	1051.49	1278.05
MLP (10)	1.99	1.69	2.66	2.84	3.84	4.56	11.00
MLP (20)	1.94	1.67	2.64	2.77	3.73	3.89	9.53
RNN (5)	38.04	1.78	23.17	319.82	755.57	862.19	1024.15
RNN (10)	4.50	1.68	2.65	2.82	3.78	4.03	103.55
RNN (20)	1.90	1.65	2.62	2.72	3.69	3.81	6.12

the response-time observed by the client during the peaks of our load profile. From the client perspective, when facing the first peak, all the considered policies basically guarantee the same performance level. Remarkably, leveraging on an additional compute instance (e.g., as with the RNN-based policy) does not make any difference, thus it is just a waste of computing resources. In contrast, by looking at Table 4, we can appreciate significant performance degradation for some specific policies. For instance, bad values of the 99.9th percentile (p99.9) are obtained with the static policy, the LR-based policy with too long inputs, and both the ARIMA and the neural-based policies with too short inputs. On the other hand, sufficiently good results are obtained using the LR-based policy with short inputs and the ARIMA- and ANN-based models with sufficiently long inputs.

Table 5 reports the overhead imposed by the proposed forecasting component at each activation that, in our experiments, happens once per minute. The total time includes interacting with the Monasca APIs to fetch the data needed as input. Predictive policies based on LR, MLP, and RNN all exhibit quite similar overheads, in the range of additional 60 ms of processing time every minute, which seems acceptable in the considered scenario. On the other hand, ARIMA exhibits quite heavier overheads. Notice that RNN performance is measured in a pessimistic setting, i.e., assuming to unfold the network on the full sequence in order to obtain a prediction. The ability of RNN to keep a dynamic memory of the sequence history could be leveraged on to compute the prediction more efficiently. In that case, computing time would be roughly equivalent to those reported in Table 5, divided by the input sequence length.

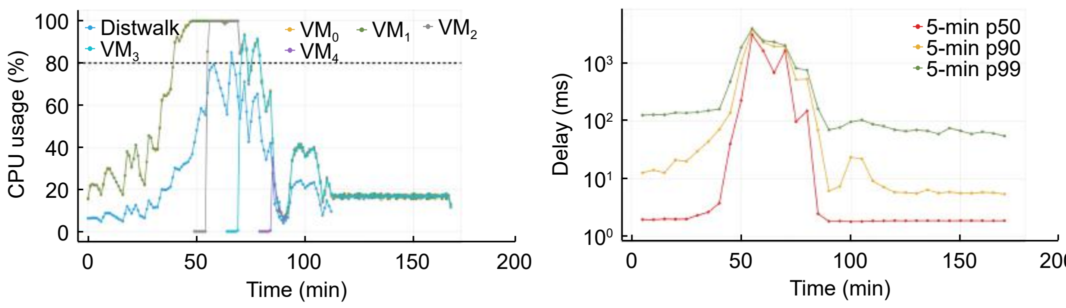
5.5 Validation on real workload traces

Our approach was also validate against a dataset exported from a real production environment. We consider the data provided by use-case D of the RECAP[12] EU project, reporting the requests handled by 3 nodes of a CDN managed by a British service provider (from 2016 to 2017). We convert a subset of these data to a 2 h long trace compatible with distwalk, such that we could generate similar traffic on our infrastructure. Similar to the previous runs, the client is set to spawn 6 threads, each one maintaining the rates specified in the trace for an interval of 1 min each. Each thread is also forced to break its work in 1000 sessions.

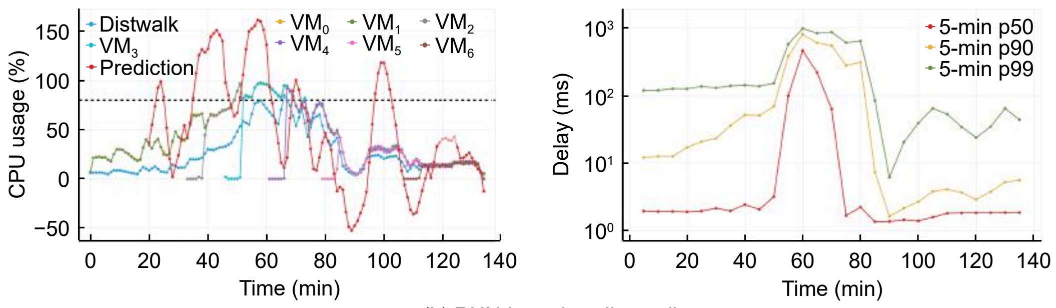
The purpose of validating our approach against a real workload is twofold: (1) we can assess the effectiveness of our approach in a production-like scenario, and (2) we can evaluate its ability to generalize workloads that do not quite resemble what the underlying models observed during training. Similar to the previous runs, we considered the static policy as a baseline and compared its performance with the predictive ones. For brevity, we only show the plots for the RNN-based policy, which exhibited the best overall performance. Figure 10a (left) shows that the static policy cannot prevent the system from saturating for 20 min (from the 48th minute to the 69th minute). The scaling decision is only triggered around the 42nd minute, which is too late, considering the artificial 10 min delay before new VMs start responding. Conversely, Fig. 10b shows the RNN-based policy acting just in time before the system completely saturates (e.g., around the 51st minute and the 58th minute). Also, the static policy run takes 30 min more to terminate because, during the saturation, the system was not able to respond to distwalk requests and caused a delay for the client to start the subsequent sessions. By comparing the plots reporting the client-side delays, we can see that, even though the RNN-based policy induces peaks of 1 s, it does not lead to a consistent increase of the delays, as the static policy does. This is also shown in Table 6, where we can see the p99.9 in the static policy case being 4 times greater than the RNN-based policy case. Despite the workload used for

Table 5 Average overhead imposed by the proposed forecasting component, for the considered predictors.

Policy	Input sequence length (min)	Total average overhead (ms)	Processing average overhead (ms)
LR	5	182.7	58.8
	10	224.5	58.4
	20	356.7	71.1
ARIMA	5	181.2	71.5
	10	245.2	102.9
	20	59.1	218.6
MLP	5	151.9	54.0
	10	213.1	60.5
	20	373.0	73.9
RNN	5	171.6	66.7
	10	256.6	72.8
	20	340.8	80.7



(a) Static scaling policy



(b) RNN-based scaling policy

Fig. 10 Experimental validation on real workload traces (left: CPU usage; right: client-side response time).

Table 6 Descriptive statistics of the client-side response time observed during the experimental validation on real workload traces (focus on the peak, 45th–90th minutes).

Policy	Response time (ms)						
	Avg	p50	p90	p95	p99	p99.5	p99.9
Static	879.01	426.26	2237.84	3019.86	3827.09	3940.71	4033.82
RNN (20)	163.64	20.62	549.91	687.74	883.46	943.71	1061.97

this additional validation does not quite resemble what the RNN observed during training, the model exhibited a reasonable generalization ability. However, to obtain a higher prediction accuracy, one should have trained the model also on some prior segment of such new data, something we omit here for brevity.

5.6 Reproducibility

For reproducibility, this work comes with a public companion repository[70] including: the Heat templates to set up the infrastructure; the configuration files for the different tools (e.g., the synthetic distwalk workload); the raw results produced by our experiments; the code to generate Figs. 5–9 and Tables 3–5; the synthetic dataset; and the code to train the forecasting models (pre-trained models are also included).

6 Conclusion and Future Work

In this paper, we proposed an architecture that enables predictive operations in cloud infrastructures. We prototyped our approach using OpenStack, extending Monasca to ingest predictive metrics that reflect the expected evolution of the monitored system in the near future. Such metrics can be seamlessly combined with the regular ones to build operations policies that go beyond standard reactive strategies. As a case-study, we realized a predictive elasticity controller for a cluster of VMs (managed by Senlin), which is able to anticipate workload changes that might not be easily handled by classical threshold-based rules. The approach was validated both on synthetic and on real workload traces from a production CDN. Remarkably, it proved to be particularly useful for services with non-negligible instance spawning time, a commonplace condition in production environments[3] (e.g., creating a new VMs and applying the required configurations may require tens of minutes).

Future work. We plan to better integrate our forecasting component within OpenStack. For instance, our idea is to provide a number of standard predictor implementations that can easily be deployed by an operator—which is not necessarily an ML expert—through the OpenStack CLI. Regarding the orchestration logics, we plan to explore additional ML techniques (e.g., RL) to experiment with alternative scaling policies, rather than only relying on threshold-based approach. However, for them to be dependable, learning-based policies cannot be delivered as black-boxes. Therefore, we also plan to provide human operators with the ability to query the models to get explanations about their outputs[71, 72], such that they can troubleshoot and fix possible erroneous behaviors. Regarding the model training, we plan to provide means for automatic periodic re-training on fresh data. Additionally, we will introduce concept drift detection to trigger model updates whenever the current version starts exhibiting performance drops, e.g., as the authors of Ref. [62] did. In this regard, our architecture would certainly benefit from integrating continual learning techniques[73]. We will also conduct a deeper validation of our architecture by considering additional datasets from real production workloads. In this way, we will address scalability issues that might arise in massive deployment scenarios, with thousands of predictive elasticity loops that control different services, possibly throughout a Cloud-Fog-Edge architecture[74]. In this regard, a promising possibility is to leverage on Monasca's scalable analytics processing architecture, which is based on Apache Storm.

Acknowledgment

This work was partly supported by the PNRR-M4C2-Investimento 1.3, Partenariato Esteso (No. PE0000 0013-FAIR).

References

[1] R. Buyya, J. Broberg, and A. M. Goscinski, *Cloud Computing: Principles and Paradigms*. Hoboken, NJ, USA: John Wiley & Sons, 2011.

[2] S. Brunner, M. Blöchlinger, G. Toffetti, J. Spillner, and T. M. Bohnert, Experimental evaluation of the cloud-native application design, in *Proc. 2015 IEEE/ACM 8th Int. Conf. Utility and Cloud Computing* (*UCC*), Limassol, Cyprus, 2015, pp. 488–493.

[3] Q. Rebjock, V. Flunkert, T. Januschowski, L. Callot, and J. Castellon, A simple and effective predictive resource scaling heuristic for large-scale cloud applications, presented at 2nd Int. Workshop on Applied AI for Database Systems and Applications, Online Event, Tokyo, Japan, 2020.

[4] W. Vogels, A new era of DevOps, powered by machine learning, https://www.allthingsdistributed.com/2021/05/devops-powered-by-machine-learning.html, 2021.

[5] S. Horsfield and A. Sethi, Introducing native support for predictive scaling with Amazon EC2 auto scaling, https://aws.amazon.com/blogs/compute/introducing-native-support-for-predictive-scaling-with-amazon-ec2-auto-scaling, 2021.

[6] A. Sethi, Using EC2 auto scaling predictive scaling policies with blue/green deployments, https://aws.amazon.com/blogs/compute/retaining-metrics-across-blue-green-deployment-for-predictive-scaling, 2021.

[7] S. Tuli, S. S. Gill, M. Xu, P. Garraghan, R. Bahsoon, S. Dustdar, R. Sakellariou, O. Rana, R. Buyya, G. Casale, et al., Hunter: AI based holistic resource management for sustainable cloud computing, *J. Systems and Software*, vol. 184, p. 111124, 2022.

[8] OpenStack, Welcome to Monasca's documentation, https://docs.openstack.org/monasca, 2022.

[9] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*: *Data Mining, Inference, and Prediction*. New York, NY, USA: Springer, 2009.

[10] G. E. Box, G. M. Jenkins, and G. C. Reinsel, *Time Series Analysis*: *Forecasting and Control*. Hoboken, NJ, USA: John Wiley & Sons, 2018.

[11] G. Lanciano, F. Galli, T. Cucinotta, D. Bacciu, and A. Passarella, Predictive auto-scaling with OpenStack monasca, in *Proc. 14th IEEE/ACM Int. Conf. Utility and Cloud Computing*, Leicester, UK, 2021, pp. 1–10.

[12] M. Leznik, R. G. Leiva, T. L. Duc, S. Svorobej, L. Närvä, M. N. Mariño, P. Willis, K. M. Giannoutakis, R. Loomba, H. Humanes, et al., RECAP artificial data traces, https://zenodo.org/record/3458559, 2019.

[13] NFV Industry Specification Group, Network functions virtualisation, Introductory White Paper, http://portal.etsi.org/NFV/NFVWhitePaper.pdf, 2012.

[14] N. Roy, A. Dubey, and A. Gokhale, Efficient autoscaling in the cloud using predictive models for workload forecasting, in *Proc. 2011 IEEE 4th Int. Conf. Cloud Computing*, Washington, DC, USA, 2011, pp. 500–507.

[15] S. Abdelwahed, J. Bai, R. Su, and N. Kandasamy, On the application of predictive control techniques for adaptive performance management of computing systems, *IEEE Trans. Netw. Serv. Manag.*, vol. 6, no. 4, pp. 212–225, 2009.

[16] V. A. F. Almeida, Capacity planning for web services techniques and methodology, in *Performance Evaluation of Complex Systems*: *Techniques and Tools*, M. C. Calzarossa and S. Tucci, eds. Berlin, Germany: Springer, 2002, pp. 142–157.

[17] S. Islam, J. Keung, K. Lee, and A. Liu, Empirical prediction models for adaptive resource provisioning in the cloud, *Future Gener. Comput. Syst.*, vol. 28, no. 1, pp. 155–162, 2012.

[18] TPC, TPC-W benchmark, http://www.tpc.org/tpcw/, 2021.

[19] M. Borkowski, S. Schulte, and C. Hochreiner, Predicting cloud resource utilization, in *Proc. 9th Int. Conf. Utility and Cloud Computing*, Shanghai, China, 2016, pp. 37–42.

[20] W. Iqbal, A. Erradi, M. Abdullah, and A. Mahmood, Predictive auto-scaling of multi-tier applications using performance varying cloud resources, *IEEE Trans. Cloud*

Comput., vol. 10, no. 1, pp. 595–607, 2022.

[21] P. Kang and P. Lama, Robust resource scaling of containerized microservices with probabilistic machine learning, in *Proc. 2020 IEEE/ACM 13th Int. Conf. Utility and Cloud Computing* (*UCC*), Leicester, UK, 2020, pp. 122–131.

[22] A. Bashar, Autonomic scaling of cloud computing resources using BN-based prediction models, in *Proc. 2013 IEEE 2nd Int. Conf. Cloud Networking* (*CloudNet*), San Francisco, CA, USA, 2014, pp. 200–204.

[23] H. Tariq, H. Al-Sahaf, and I. Welch, Modelling and prediction of resource utilization of hadoop clusters: A machine learning approach, in *Proc. 12th IEEE/ACM Int. Conf. Utility and Cloud Computing*, Auckland, New Zealand, 2019, pp. 93–100.

[24] L. Ju, P. Singh, and S. Toor, Proactive autoscaling for edge computing systems with kubernetes, in *Proc. 14th IEEE/ACM Int. Conf. Utility and Cloud Computing Companion*, Leicester, UK, 2021, pp. 1–8.

[25] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms, in *Proc. 26th Symp. Operating Systems Principles*, Shanghai, China, 2017, pp. 153–167.

[26] J. Kumar, A. K. Singh, and R. Buyya, Self directed learning based workload forecasting model for cloud resource management, *Inf. Sci.*, vol. 543, pp. 345–366, 2021.

[27] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, Workload prediction using ARIMA model and its impact on cloud applications' QoS, *IEEE Trans. Cloud Comput.*, vol. 3, no. 4, pp. 449–458, 2015.

[28] I. K. Kim, W. Wang, Y. Qi, and M. Humphrey, CloudInsight: Utilizing a council of experts to predict future cloud application workloads, in *Proc. 2018 IEEE 11th Int. Conf. Cloud Computing* (*CLOUD*), San Francisco, CA, USA, 2018, pp. 41–48.

[29] V. K. Jayakumar, J. Lee, I. K. Kim, and W. Wang, A self-optimized generic workload prediction framework for cloud computing, in *Proc. 2020 IEEE Int. Parallel and Distributed Processing Symp.* (*IPDPS*), New Orleans, LA, USA, 2020, pp. 779–788.

[30] C. H. Z. Nicodemus, C. Boeres, and V. E. F. Rebello, Managing vertical memory elasticity in containers, in *Proc. 2020 IEEE/ACM 13th Int. Conf. Utility and Cloud Computing* (*UCC*), Leicester, UK, 2020, pp. 132–142.

[31] C. Makaya, D. Freimuth, D. Wood, and S. Calo, Policy-based NFV management and orchestration, in *Proc. 2015 IEEE Conf. Network Function Virtualization and Software Defined Network* (*NFV-SDN*), San Francisco, CA, USA, 2016, pp. 128–134.

[32] X. Fei, F. Liu, H. Xu, and H. Jin, Adaptive VNF scaling and flow routing with proactive demand prediction, in *Proc. IEEE Conf. Computer Communications*, Honolulu, HI, USA, 2018, pp. 486–494.

[33] S. Rahman, T. Ahmed, M. Huynh, M. Tornatore, and B. Mukherjee, Auto-scaling VNFs using machine learning to improve QoS and reduce cost, in *Proc. 2018 IEEE Int. Conf. Communications* (*ICC*), Kansas City, MO, USA, 2018, pp. 1–6.

[34] S. S. Rangapuram, M. Seeger, J. Gasthaus, L. Stella, Y. Wang, and T. Januschowski, Deep state space models for time series forecasting, in *Proc. 32nd Int. Conf. Neural Information Processing Systems*, Montreal, Canada, 2018, pp. 7796–7805.

[35] N. Laptev, J. Yosinski, L. E. Li, and S. Smyl, Time-series extreme event forecasting with neural networks at Uber, presented at ICML 2017 Time Series Workshop, Sydney, Australia, 2017.

[36] H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P. A. Muller, Deep learning for time series classification: A review, *Data Min. Knowl. Discov.*, vol. 33, no. 4, pp. 917–963, 2019.

[37] P. Malhotra, V. Tv, L. Vig, P. Agarwal, and G. Shroff, TimeNet: Pre-trained deep recurrent neural network for time series classification, presented at 25th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, Bruges, Belgium, 2017.

[38] Z. Zaman, S. Rahman, and M. Naznin, Novel approaches for VNF requirement prediction using DNN and LSTM, in *Proc. 2019 IEEE Global Communications Conference* (*GLOBECOM*), Waikoloa, HI, USA, 2020, pp. 1–6.

[39] I. Sutskever, O. Vinyals, and Q. V. Le, Sequence to sequence learning with neural networks, in *Proc. 27th Int. Conf. Neural Information Processing Systems*, Montreal, Canada, 2014, pp. 3104–3112.

[40] T. Cucinotta, G. Lanciano, A. Ritacco, F. Brau, F. Galli, V. Iannino, M. Vannucci, A. Artale, J. Barata, and E. Sposato, Forecasting operation metrics for virtualized network functions, in *Proc. 2021 IEEE/ACM 21st Int. Symp. on Cluster, Cloud and Internet Comp.*, Melbourne, Australia, 2021, pp. 596–605.

[41] R. Mijumbi, S. Hasija, S. Davy, A. Davy, B. Jennings, and R. Boutaba, Topology-aware prediction of virtual network function resource requirements, *IEEE Trans. Netw. Serv. Manag.*, vol. 14, no. 1, pp. 106–120, 2017.

[42] D. Bacciu, F. Errica, A. Micheli, and M. Podda, A gentle introduction to deep learning for graphs, *Neural Netw.*, vol. 129, pp. 203–221, 2020.

[43] G. A. Carella, M. Pauls, L. Grebe, and T. Magedanz, An extensible autoscaling engine (AE) for software-based network functions, in *Proc*. 2016 *IEEE Conf. Network Function Virtualization and Software Defined Networks* (*NFC-SDN*), Palo Alto, CA, USA, 2016, pp. 219–225.

[44] P. Tang, F. Li, W. Zhou, W. Hu, and L. Yang, Efficient auto-scaling approach in the telco cloud using self-learning algorithm, in *Proc. 2015 IEEE Global Communications Conference* (*GLOBECOM*), San Diego, CA, USA, 2015, pp. 1–6.

[45] Y. Xiao, Q. Zhang, F. Liu, J. Wang, M. Zhao, Z. Zhang, and J. Zhang, NFVdeep: Adaptive online service function chain deployment with deep reinforcement learning, in *Proc. 2019 IEEE/ACM 27th Int. Symp. Quality of Service* (*IWQoS*), Phoenix, AZ, USA, 2019, pp. 1–10.

[46] C. H. T. Arteaga, F. Rissoi, and O. M. C. Rendon, An adaptive scaling mechanism for managing performance

variations in network functions virtualization: A case study in an NFV-based EPC, in *Proc. 2017 13th Int. Conf. Network and Service Management*, Tokyo, Japan, 2017, pp. 1–7.

[47] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling, in *Proc. 2017 17th IEEE/ACM Int. Symp. Cluster, Cloud and Grid Computing*, Madrid, Spain, 2017, pp. 64–73.

[48] A. A. Khaleq and I. Ra, Development of QoS-aware agents with reinforcement learning for autoscaling of microservices on the cloud, in *Proc. 2021 IEEE Int. Conf. Autonomic Computing and Self-Organizing Systems Companion*, Washington, DC, USA, 2021, pp. 13–19.

[49] L. Schuler, S. Jamil, and N. Kühl, AI-based resource allocation: Reinforcement learning for adaptive autoscaling in serverless environments, in *Proc. 2021 IEEE/ACM 21st Int. Symp. Cluster, Cloud and Internet Comp.*, Melbourne, Australia, 2021, pp. 804–811.

[50] OpenStack, OpenStack compute (nova), https://docs.openstack.org/nova, 2022.

[51] OpenStack, OpenStack block storage (cinder) documentation, https://docs.openstack.org/cinder, 2022.

[52] OpenStack, Welcome to glance's documentation! https://docs.openstack.org/glance, 2022.

[53] OpenStack, Welcome to neutron's documentation! https://docs.openstack.org/neutron, 2022.

[54] OpenStack, Welcome to the Senlin documentation! https://docs.openstack.org/senlin, 2022.

[55] OpenStack, Welcome to the heat documentation! https://docs.openstack.org/heat, 2022.

[56] D. Truong and J. Cross, How blizzard entertainment uses autoscaling with overwatch, https://www.openstack.org/videos/summits/denver2019/how-blizzard-entertainment-uses-autoscaling-with-overwatch, 2022.

[57] OpenStack, Octavia documentation, https://docs.openstack.org/octavia, 2022.

[58] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[59] S. Hochreiter and J. Schmidhuber, Long short-term memory, *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[60] J. F. Kolen and S. C. Kremer, Gradient flow in recurrent nets: The difficulty of learning LongTerm dependencies, in *A Field Guide to Dynamical Recurrent Networks*, J. F. Kolen and S. C. Kremer, eds. New York, NY, USA: Wiley-IEEE Press, 2001, pp. 237–243.

[61] M. Turowski and A. Lenk, Vertical scaling capability of OpenStack, in *Service-Oriented Computing - ICSOC 2014 Workshops*, F. Toumani, B. Pernici, D. Grigori, D. Benslimane, J. Mendling, N. B. Hadj-Alouane, B. Blake, O. Perrin, I. S. Moustafa, and S. Bhiri, eds. Cham, Switzerland: Springer, 2015, pp. 351–362.

[62] L. Kidane, P. Townend, T. Metsch, and E. Elmroth, When and how to retrain machine learning-based cloud management systems, in *Proc. 2022 IEEE Int. Parallel and Distributed Processing Symp. Workshops* (*IPDPSW*), Lyon, France, 2022, pp. 688–698.

[63] G. Lanciano, Monasca-predictor, https://github.com/giacomolanciano/monasca-predictor, 2022.

[64] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., Scikit-learn: Machine learning in python, *J. Mach. Learn. Res.*, vol. 12, no. 85, pp. 2825–2830, 2011.

[65] S. Seabold and J. Perktold, Statsmodels: Econometric and statistical modeling with python, in *Proc. 9th Python in Science Conf.*, Austin, TX, USA, 2010, pp. 92–96.

[66] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., Pytorch: An imperative style, high-performance deep learning library, in *Proc. 33rd Conf. Neural Information Processing Systems*, Vancouver, Canada, 2019, pp. 8024–8035.

[67] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al., TensorFlow: Large-scale machine learning on heterogeneous systems, arXiv preprint arXiv: 1603.04467, 2015.

[68] T. Cucinotta, Distwalk, https://github.com/tomcucinotta/distwalk, 2022.

[69] OpenStack, Welcome to Kolla's documentation! https://docs.openstack.org/kolla, 2022.

[70] G. Lanciano, F. Galli, T. Cucinotta, D. Bacciu, and A. Passarella, Companion repo of the paper extending OpenStack Monasca for predictive elasticity control, https://github.com/giacomolanciano/predictive-elasticity-monasca, 2022.

[71] M. Resta, A. Monreale, and D. Bacciu, Occlusion-based explanations in deep recurrent models for biomedical signals, *Entropy*, vol. 23, no. 8, p. 1064, 2021.

[72] B. Lim, S. Ö. Arik, N. Loeff, and T. Pfister, Temporal fusion transformers for interpretable multi-horizon time series forecasting, *Int. J. Forecast.*, vol. 37, no. 4, pp. 1748–1764, 2021.

[73] A. Cossu, A. Carta, V. Lomonaco, and D. Bacciu, Continual learning for recurrent neural networks: An empirical evaluation, *Neural Netw.*, vol. 143, pp. 607–627, 2021.

[74] G. Merlino, R. Dautov, S. Distefano, and D. Bruneo, Enabling workload engineering in edge, fog, and cloud computing through OpenStack-based middleware, *ACM Trans. Internet Technol.*, vol. 19, no. 2, pp. 1–22, 2019.

Giacomo Lanciano received the MSc degree in computer science from Sapienza University of Rome, Italy in 2017. He currently is pursuing the PhD degree in data science at Scuola Normale Superiore, Italy. His research interests lie at the intersection of cloud computing and data science, with a focus on data-driven methods for data center operations support. He was also a research intern at Nokia Bell Labs in Stuttgart, Germany, working on large language models for deployment code analysis.

Filippo Galli received the bachelor degree in electrical engineering and the master degree in mechatronics engineering from Politecnico di Torino, Italy in 2014 and 2018, respectively. He currently is pursuing the PhD degree in data science at Scuola Normale Superiore, Italy. His current research interests are in machine learning, privacy, and distributed computing.

Davide Bacciu received the PhD degree in computer science and engineering from IMT Lucca, Italy for which he received the 2009 E.R. Caianiello prize. He is an associate professor at University of Pisa, Italy, where he heads the pervasive AI lab. Previously, he was a visiting researcher at the Neural Computation Research Group, LJMU, and the Cognitive Robotic Systems Laboratory, Orebro University, Sweden. He has co-authored over 140 research works on neural networks, generative learning, Bayesian models, learning for graphs, continual learning, and distributed and embedded learning systems. He is the coordinator of the H2020 TEACHING project. He has been a secretary and board member of the Italian Association for AI, a senior member of the IEEE, and a member of the IEEE CIS Neural Networks Technical Committee. He is an associate editor of *IEEE Transactions on Neural Networks and Learning Systems*, and he chairs the IEEE CIS task force on learning for structured data.

Tommaso Cucinotta received the MSc degree in computer engineering from University of Pisa, Italy in 2000, and the PhD degree in computer engineering from Scuola Superiore Sant'Anna (SSSA), Italy in 2004, where he has been investigating on real-time scheduling for soft real-time and multimedia applications, and predictability in infrastructures for cloud computing and NFV. He has been the member of technical staff (MTS) in Bell Labs in Dublin, Ireland, investigating on security and real-time performance of cloud services. He has been a software engineer in Amazon Web Services in Dublin, Ireland, where he worked on improving the performance and scalability of DynamoDB. He is an associate professor at SSSA, Italy since 2016 and the head of the Real-Time Systems Laboratory (RETIS) since 2019. He has co-authored 120+ research papers on international conferences and journals, and 8 international patent grants. He is a senior IEEE member.

Andrea Passarella received the PhD degree from University of Pisa, Italy in 2005. He is a research director at the Institute for Informatics and Telematics (IIT) of the National Research Council of Italy (CNR). Prior to join IIT he was with the Computer Laboratory of University of Cambridge, UK. He has published 170+ papers on human-centric data management for self organising networks, decentralised AI, next generation Internet, online and mobile social networks, opportunistic, ad hoc, and sensor networks. He received four best paper awards, including IFIP Networking 2011 and IEEE WoWMoM 2013. He is the general chair of IEEE PerCom 2022. He is the founding associate editor-in-chief of the Elsevier journal *Online Social Networks and Media* (*OSNEM*). He is the co-author of the book *Online Social Networks*: *Human Cognitive Constraints in Facebook and Twitter Personal Graphs* (Elsevier, 2015). He is the principal investigator of the EU CHIST-ERA SAI (Social Explainable AI) project.