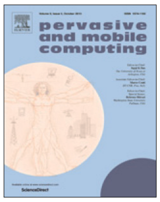
Pervasive and Mobile Computing 87 (2022) 101722

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc



Online machine learning for auto-scaling in the edge computing[☆]

Thiago Pereira da Silva [a,1,*], Aluizio Rocha Neto [c], Thais Vasconcelos Batista [a], Flávia C. Delicato [b], Paulo F. Pires [b], Frederico Lopes [c]

[a] Federal University of Rio Grande do Norte (UFRN), Natal, Rio Grande do Norte, Brazil
[b] Fluminense Federal University (UFF), Niterói, Rio de Janeiro, Brazil
[c] Instituto Metrópole Digital (IMD), Natal, Rio Grande do Norte, Brazil

A R T I C L E   I N F O

Article history:
Received 1 July 2022
Received in revised form 20 October 2022
Accepted 31 October 2022
Available online 5 November 2022

Keywords:
Edge computing
Auto-scaling
Autonomic computing
Intelligence service
Online machine learning

A B S T R A C T

The evolution of edge computing devices has enabled machine intelligence techniques to process data close to its producers (the sensors) and end-users. Although edge devices are usually resource-constrained, the distribution of processing services among several nodes enables a processing capacity similar to cloud environments. However, the edge computing environment is highly dynamic, impacting the availability of nodes in the distributed system. In addition, the processing workload for each node can change constantly. Thus, the scaling of processing services needs to be rapidly adjusted, avoiding bottlenecks or wasted resources while meeting the applications' QoS requirements. This paper presents an auto-scaling subsystem for container-based processing services using online machine learning. The auto-scaling follows the MAPE-K control loop to dynamically adjust the number of containers in response to workload changes. We designed the approach for scenarios where the number of processing requests is unknown beforehand. We developed a hybrid auto-scaling mechanism that behaves reactively while a prediction online machine learning model is continuously trained. When the prediction model reaches a desirable performance, the auto-scaling acts proactively, using predictions to anticipate scaling actions. An experimental evaluation has demonstrated the feasibility of the architecture. Our solution achieved fewer service level agreement (SLA) violations and scaling operations to meet demand than purely reactive and no scaling approaches using an actual application workload. Also, our solution wasted fewer resources compared to the other techniques.

1. Introduction

Recently, most smart sensor applications based on Internet of Things (IoT) devices rely on edge computing to reduce communication latency in processing data generated by such sensors [1]. Edge devices have computational limitations, but the distribution of processing services among several nodes enables a processing capacity close to cloud environments.

[☆] This research was partially funded by CNPq, Brazil grant 308274/2016-4, FAPESP, Brazil grant 2015/24144-7, and by FAPERJ, Brazil grant 200.972/2022. Professors Thais Batista, Flavia Delicato and Paulo Pires are CNPq Fellows.
[*] Corresponding author.
E-mail address: thiagosilva@ufmt.br (T.P.d. Silva).
[1] Researcher.

https://doi.org/10.1016/j.pmcj.2022.101722
1574-1192/© 2022 Elsevier B.V. All rights reserved.

However, the edge computing environment is highly heterogeneous and dynamic, making the distribution of processing tasks a non-trivial issue. Furthermore, the workload for each edge node can constantly change according to the occurrence of the events of interest. A well-known example of an event of interest is finding a missing person in public spaces through face recognition. The processing task in charge of face recognition must run rapidly, since the location of the recognized person is a piece of perishable information.

In a previous work [2], we developed a distributed system for video analytics designed to leverage edge computing capabilities. Such a system creates a microservice-based architecture called *Multilevel Information Distributed Processing Architecture* (MELINDA) [3]. MELINDA breaks the burdensome processing of large-scale video streams into smaller tasks as a data processing workflow deployed in edge nodes near the sensors (cameras). The load balance approach adopted by MELINDA uses the producer–consumer pattern (*producer* → *worker* → *consumer* of message queues). Each message in the workflow carries an image containing the application object of interest. When the first node in the workflow detects a frame from the camera having the object of interest, it produces a data message containing this frame and its contextual information. A worker node will then process this message to extract the object's features, and the consumer node will interpret the event of interest produced by that object. In MELINDA architecture, an *Intelligence Service* denotes a processing task shared by multiple smart applications to process an event of interest. Face recognition is an example of a task aimed at identifying people in different application domains. The microservice-based approach for sharing processing tasks adopted by MELINDA improves the system's processing throughput by using the idle capacity of nodes running the intelligence service.

In [4], we present an algorithm to solve the optimization problem of allocating a set of worker nodes with sufficient processing capacity to execute the workflow while minimizing the operational cost related to latency and energy consumption and maximizing service availability. However, a question arose with MELINDA's resource allocation model regarding the limit for sharing an intelligence service. A worker node extracts the object's features from the image of interest sent by the producer. The arrival of such an image to the worker node depends on an event on the camera that is the workflow's data source. The frequency of events is very dynamic with monitoring cameras, ranging from a few seconds to several minutes or even hours. Thus, the idle capacity of a worker node running the intelligence service might be relatively high. The worker node can exploit such idleness by accepting and running new workflows while still meeting the QoS requirements of the applications. To tackle this issue, we incorporate the elasticity property in the MELINDA architecture.

The elasticity of a system concerns its ability to adapt to workload fluctuations by provisioning and de-provisioning resources. In this sense, to achieve elasticity in the MELINDA architecture and avoid the idleness of nodes, an auto-scaling strategy is required to scale worker nodes to meet the workload demand and optimize resources usage without human intervention. However, the auto-scaling approach must address two main challenges:

(i) meeting the stringent time requirements of video analytics applications;
(ii) coping with the high dynamism of the edge computing environment.

The stringent time requirements of video analytics applications mean that scaling operations cannot harm the system in a way that violates the specified Service Level Agreement (SLA). Hence, the scaling approach must optimize the use of resources and avoid wastage to enable services to be shared by more workflows. In this way, the main goal of auto-scaling at the network edge is to meet the applications' latency requirements and optimize the resource usage of the edge devices. For example, in a video analytics scenario, instead of sending the images to the network core to be processed in the Cloud, auto-scaling enables the processing of the pictures at the edge, decreasing the response time and meeting the stringent latency requirements. There are several edge computing scenarios that can benefit from auto-scaling [5].

Different techniques have been employed in the development of auto-scaling. Machine learning-based techniques for workload prediction have been explored recently and have shown promising results compared to rule-based approaches [6]. However, batch or traditional machine learning-based techniques require datasets representing the data's behavior to create the prediction models. Such datasets are not always available, limiting the use of these techniques in various contexts. Moreover, even when datasets are available to create the model, the data's behavior can change over time. Thus, the model becomes outdated. Such an issue is a research challenge that requires different approaches than the traditional ones to ensure scalability in the edge computing environment.

A promising technique to deal with highly dynamic data (i.e., the occurrence of events) is Online Machine Learning (OML) [7,8]. In online learning, data becomes available in sequential order. Such a model updates the best predictor for future data at each time window instead of batch learning, which generates the best predictor by learning on the entire training dataset. Therefore, auto-scaling can employ an online machine learning model to operate in an environment without prior knowledge of workload behavior.

This paper presents an auto-scaling subsystem for container-based processing services in MELINDA using Online Machine Learning. The auto-scaling follows the MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) autonomic control loop [9] to dynamically adjust the number of worker nodes in response to workload changes. The approach considers scenarios where the number of processing requests is unknown beforehand. We designed a hybrid auto-scaling mechanism that behaves reactively when the online machine learning prediction accuracy is under a given threshold. The online machine learning model is continuously trained as new observation arrives. When the model reaches a desirable performance, the auto-scaling behaves proactively by using predictions to anticipate scaling actions. Online machine learning models are suited for handling sudden changes in data behavior (concept drift) and learning from new data in near real-time. Recent online machine learning algorithms can train and predict in a short time (milliseconds). Also, such algorithms are lightweight and consume few computational resources [8], so they are well suited to edge computing. Therefore, the main contributions of our work are:

- We extend our previous work [10], specifying a software subsystem for MELINDA that incorporates the auto-scaling approach;
- We develop an auto-scaling based on online machine learning that does not require a dataset in advance;
- The developed auto-scaling applies a hybrid approach, changing its behavior according to the accuracy of the online machine learning prediction model;
- The auto-scaling follows the MAPE-K autonomic control loop to endow the auto-scaling approach with the autonomic self-optimizing property;
- We present a new performance evaluation using a scenario of an intelligent video analytics application for smart cities. We use real and synthetic application workloads and new metrics to demonstrate the auto-scaling subsystem's viability.

The organization of this paper is as follows. Section 2 reviews the main concepts used in this paper. Section 3 describes the proposed auto-scaling subsystem, while Section 4 introduces a running example to demonstrate the proposal better and evaluates its performance using some setups of processing nodes. Section 5 discusses related work. Finally, Section 6 provides the conclusions and future directions.

2. Background

This section presents the background to the study, placing the research problem in a proper context discussed in this paper. First, Section 2.1 describes the MELINDA architecture, and Section 2.2 briefly presents the foundations of auto-scaling. Section 2.3 introduces the basic concepts of online machine learning. Finally, Section 2.4 presents the fundamental concepts of autonomic computing and introduces the MAPE-K architectural blueprint for autonomic systems.

2.1. MELINDA architecture

Motivated by the recent advances in edge computing and microservice architectures for IoT [11], the MELINDA architecture proposes a strategy to distribute video analytics tasks in edge computing. MELINDA leverages the potential of new generation edge devices tailored for running Machine Intelligence (MI) techniques near the data sources. MELINDA is a three-tier architecture (Fig. 1) that provides software components to break down the intensive processing duty of transforming raw data streams into high-abstraction events of interest. The MELINDA system model splits the video stream into a pipeline with three types of task: (i) the *measurement level task* (MLT) filters the raw video stream selecting only those frames (images) that have the object of interest; (ii) the *feature task* (FLT), extracts the object features in these images; and (iii) the *decision level task* (DLT) interprets the event held by the object. Each processing layer has its task – MLT, FLT, and DLT, respectively – running on different fixed node devices and interconnected via a wired LAN or MAN network.

It is not the purpose of this article to present all the elements of MELINDA. Instead, we focused on the auto-scaling subsystem, and a detailed description of the architecture is in [2,3].

2.2. Auto-scaling

Auto-scaling aims to optimize the use of resources without human intervention and improve the performance of the services offered to end-users, meeting the application QoS requirements without wasting resources and avoiding over/underprovisioning. According to [12], auto-scaling is the primary technique to ensure elasticity to cloud providers so that resources can be provisioned and released to scale, outward and inward, depending on demand.

Underprovisioning occurs when the allocated computational resources are insufficient to meet the demand and are saturated. Oppositely, an overprovisioned system has more computational resources than is needed to meet the demand. In this way, an auto-scaling approach ensures the system's elasticity to dynamically adapt to workload fluctuations by provisioning and de-provisioning resources autonomously. Thus, the provisioned resources must match the current demand. However, resources cannot exceed the need to avoid waste.

Auto-scaling approaches are classified mainly according to (i) the technique used to achieve scalability and (ii) the operational behavior [6]. Regarding the scalability techniques, *vertical scaling* is characterized by resizing (scaling up/down) computational resources such as memory, CPU, and storage of running service instances. On the other hand, *horizontal scaling* approaches create or remove replicas (scale in/out) of a given service to share the workload between the various resources.

Regarding the operational behavior, a *proactive* approach predicts the computational resource usage by analyzing performance metrics (e.g., latency, CPU, and memory usage). The continuous monitoring of such performance metrics produces historical data, which are processed by Machine Learning (ML) and Statistical algorithms to predict future

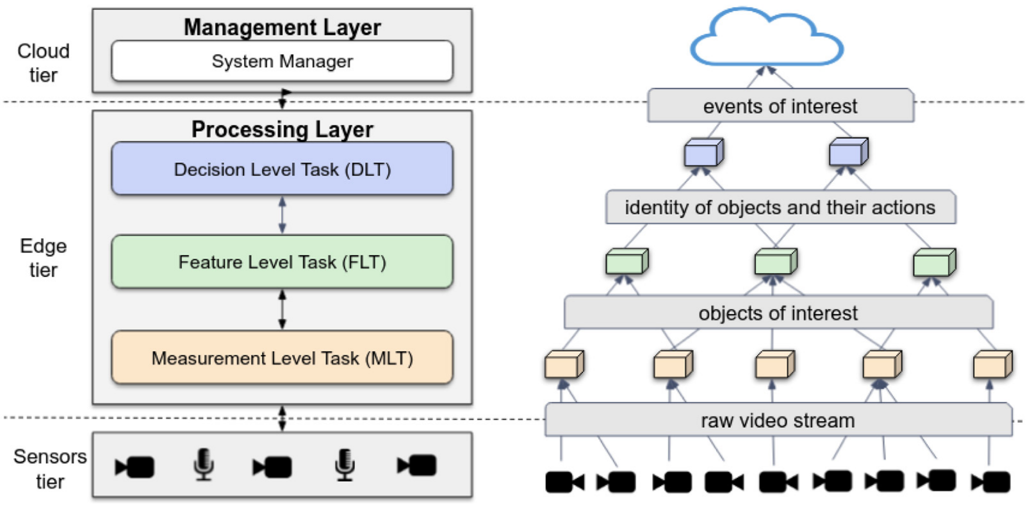


Fig. 1. MELINDA three-tier architecture for distributing video analytics tasks in edge computing.

workload. The auto-scaling can adjust the number of resources to meet the QoS requirements given the expected workload. Such algorithms require large datasets of historical data to observe the data behavior and create models that make good predictions. In general, training a model with large datasets takes a long time; therefore, once a model is created, it is usually no longer updated, or if it is, it is updated infrequently [13]. This way, the model becomes less accurate and does not reflect the actual reality of the data.

In most cases, traditional ML or statistical methods cannot correctly detect and cope with sudden changes in data behavior (*Data Drift*). In a data drift, the data (independent variables) presented for the models have never been seen before and do not exhibit the same behavior as the data used in the learning process [14]. Thus, data drift is a significant threat that can harm the quality of the predictions of ML models.

Reactive approaches are based on threshold limits and are not able to forecast the future workload. Such an approach performs scaling actions when it detects changes that exceed the limits (thresholds). However, reactive strategies tend to degrade the overall performance of a system [15]. Moreover, defining thresholds is not trivial and influences the application performance at runtime.

In the literature, [6,16,17], proposals dealing with auto-scaling mainly use traditional ML models (or batch models), where all datasets are available in advance, and multiple passes over the static datasets can create prediction models. However, in scenarios with no dataset nor prior knowledge about the behavior of the data, batch learning models are not suitable. Furthermore, batch models become outdated since their parameters do not change. Thus, such models must be retrained and redeployed constantly to fit new data behaviors. Some models (e.g., deep neural networks) require many computational resources for training and predictions, often unavailable at the network edge. Retraining the model involves collecting data over time, detecting changes in data patterns, retraining the model with new data, and redeploying the model. Applying such a process is complex in highly dynamic contexts like edge computing.

2.3. *Online machine learning*

Online Machine Learning is an embodiment of machine learning that deals with learning environments that change over time, e.g., an abrupt changing service request requires the system quickly adapts to meet the workload. OML algorithms continuously analyze the data stream, taking each observation in turn and improving itself as it updates its parameters, i.e., the model learns with each new statement. Thus, such models are best suited for handling sudden changes in data behavior (data drift) and learning from new data in near real-time. The model achieves satisfactory performance only after several data observations have gone through the training process, and the model adjusts its parameters based on these several observations. Due to this characteristic, the initial predictions yielded by the online machine learning predictor are not accurate. Although, over time, accuracy tends to improve and become comparable to traditional ML models [18]. In recent years, researchers have proposed several online regression algorithms [7,19].

In OML, observations arrive from a data stream in sequential order. Each observation is denoted as (X_t, y_t^P), where X_t is a set of features or independent variables, y_t^P is the predicted value, and t is the timestamp. However, the predicted value y_t^P may not represent the correct value because it is only a prediction, and such value may be different from the *ground truth*. So, the model is trained when the ground truth y_t^T is available, and the metric that represents the performance of the models is updated using the predicted values and their corresponding ground truths.

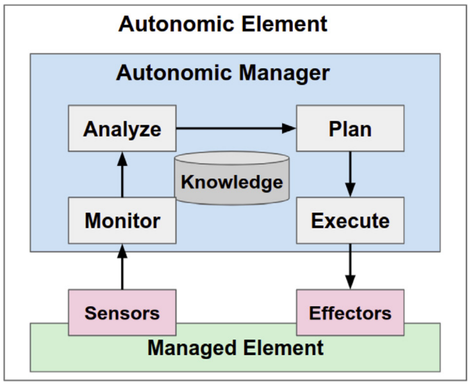


Fig. 2. MAPE-K control loop composed of four phases. Based on [22].

To evaluate the performance of the OML model, we used the *Delayed Progressive Validation Method* proposed by [20]. Such a method is an extension of the Prequential Evaluation Method [21] for scenarios where it cannot use the interleaved test-then-train method. The Delayed Progressive Validation Method requires temporary storage of each prediction y_t^P for future checking with the ground truth y_t^T. Considering a delay $\Delta > 0$, we can define a quadruplet as $(t+\Delta, X_t, y_t^P, y_t^T)$ and add it to a sorted list Q. At the time $t + \Delta$, the model is given access to the ground truth and can, therefore, be updated using X_t and y_t^T. The metric that represents the performance of the model (e.g., R2 score (R^2), Mean Squared Error (MSE), and Mean Absolute Error (MAE)) will also be updated with y_t^T and y_t^P.

An essential aspect of the evaluation method concerns their ability to detect concept drift quickly. The ground truth must be available almost immediately after a prediction. Thus, the ground truth y_t^T of an observation X_t must be already available when the following observation X_{t+1} arrives; otherwise, the ability to detect drifts will be negatively affected [19].

2.4. Autonomic computing

Autonomic computing is a computer's ability to automatically manage itself through adaptive technologies to change according to business policies and objectives. The concept of autonomic computing relies on the observed autonomic systems found in nature, such as the autonomic nervous system of humans. IBM initiated autonomic computing in the early 2001s [22]. IBM envisioned that a computer system with self-management capacity could overcome the rapidly growing complexity of management and reduce the complexity posed to further growth.

A self-management autonomic capability of a system enables the system to perform adaptations based on situations sensed in the environment. The transformations performed automatically by the system at runtime aim to fulfill the assigned function for the system. The function of any autonomic capability is a control loop that collects details from the system and acts accordingly [23]. The self-management system components of an autonomic computing system have an embedded control loop functionality. According to [22], there are four broad embedded control loop categories: Self-configuring; Self-healing; Self-optimization; and Self-protection. Such categories are the objectives or attributes of the system components.

An auto-scaling needs to ensure the self-optimizing capacity, since the tuning actions performed by the auto-scaling aim to adapt the system by increasing or decreasing resources to meet the workload fluctuations. This way, an auto-scaling improves the overall utilization and ensures meeting the service requests promptly. Therefore, auto-scaling is a self-optimizing autonomic manager that automates IT professionals' tasks to optimize the system.

An autonomic architecture comprises the attributes that allow self-management by involving control loops. The MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) feedback loop [22] is the most influential reference model for autonomic computing. Proposed by IBM, its primary goal is to allow systems to be mutable whenever the environment changes. MAPE-K encompasses a control loop of four phases/steps. Such steps are namely *Monitoring*, *Analysis*, *Planning*, and *Execution*. All the steps access a shared *knowledge* base that is fundamental to decision-making. In addition, the model defines the entities *Autonomic Element*, *Managed Element*, and *Autonomic Manager*, as illustrated in Fig. 2.

As illustrated in Fig. 2, an **Autonomic Element** (AE) is an executable software unit that exhibits autonomic properties (e.g., self-optimizing property in case of auto-scaling), implements a control loop and performs the four functions previously defined. It regularly perceives the environment through sensors, reasons about the current situation, and triggers adaptations through actuators/effectors when and where necessary. An AE consists of two distinct modules: managed resources/element and autonomic manager.

The **Autonomic Manager** implements the MAPE-K control loop and enhances the autonomic loop functionality as it absorbs and creates knowledge. The autonomic manager is in charge of the runtime administration of the managed

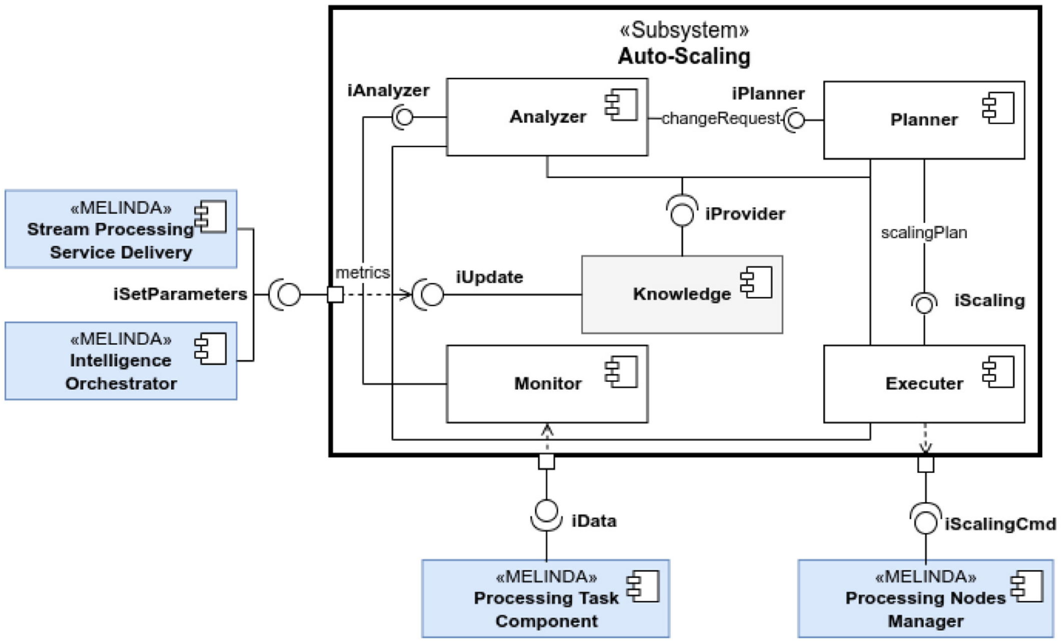


Fig. 3. Architecture of the MELINDA auto-scaling subsystem based on MAPE-K.

resource/element. It perceives the current situation, internal state, and external context determines desired management actions and governs their execution. The **Managed Resource** – also called managed element or artifacts – is an entity in the runtime environment of the autonomic element with autonomous behavior and can be driven through coupling with an autonomic manager. Such a managed resource/element must be adapted at runtime as a function of internal or external change that impacts its goals. The managed resource/element provides specific interfaces for monitoring and adaptation called touchpoints or control points. The two types of control points are sensors and actuators/effectors, and they enable the autonomic manager's work by transmitting events, information, or properties.

Sensors collect information about the managed resource/element, such as the state or their current performance (performance metrics). At the same time, **actuators/effectors** are responsible for changing the managed resource/element, such as increasing or decreasing the number of virtual CPUs of a VNF instance. The touchpoints employ mechanisms such as log files, events, commands, application programming interfaces (APIs), and configuration files. These mechanisms provide various ways to gather details about and change the behavior of the managed resources.

3. Auto-scaling subsystem

Our proposed auto-scaling subsystem has a hybrid behavior, proactively predicting the workload to anticipate scaling actions. However, it behaves reactively whenever the prediction model does not meet the desired quality. We employ an Online Machine Learning predictor to predict the workload of services running at the network edge. In terms of architecture, the proposed subsystem is based on the MAPE-K autonomic control loop. We employed MAPE-K to endow the auto-scaling approach with the autonomic self-optimizing property. The adaptations performed by auto-scaling increase and decrease the number of services being executed (creating/removing replicas of services), aiming to adapt the system to the changes in workload. This way, the auto-scaling improves the overall resource utilization and ensures service requests are properly met, minimizing application QoS requirements violations.

The proposed auto-scaling subsystem acts as an autonomic manager that implements the MAPE-K control loop and is in charge of runtime administration of the managed resource (services) [9]. It regularly perceives the environment, reasons about the current execution context and the internal state of the managed resources (the set of services), and triggers adaptations (scaling operations) when necessary. The internal state denotes a set of properties that define the current state of manageable services, such as the number of resources associated with the service, incoming requests, resource usage, etc. The triggered adaptations modify the behavior and internal structures of the system to satisfy a set of requirements, typically QoS requirements.

The managed resources are the set of services managed by MELINDA. Each service at the runtime environment has an autonomous behavior and can be managed by the auto-scaling subsystem (autonomic manager). In this way, the services can be adapted at runtime. The services provide specific interfaces for monitoring and adaptation. The auto-scaling subsystem can collect data about the state and current performance of services (performance data) and change the

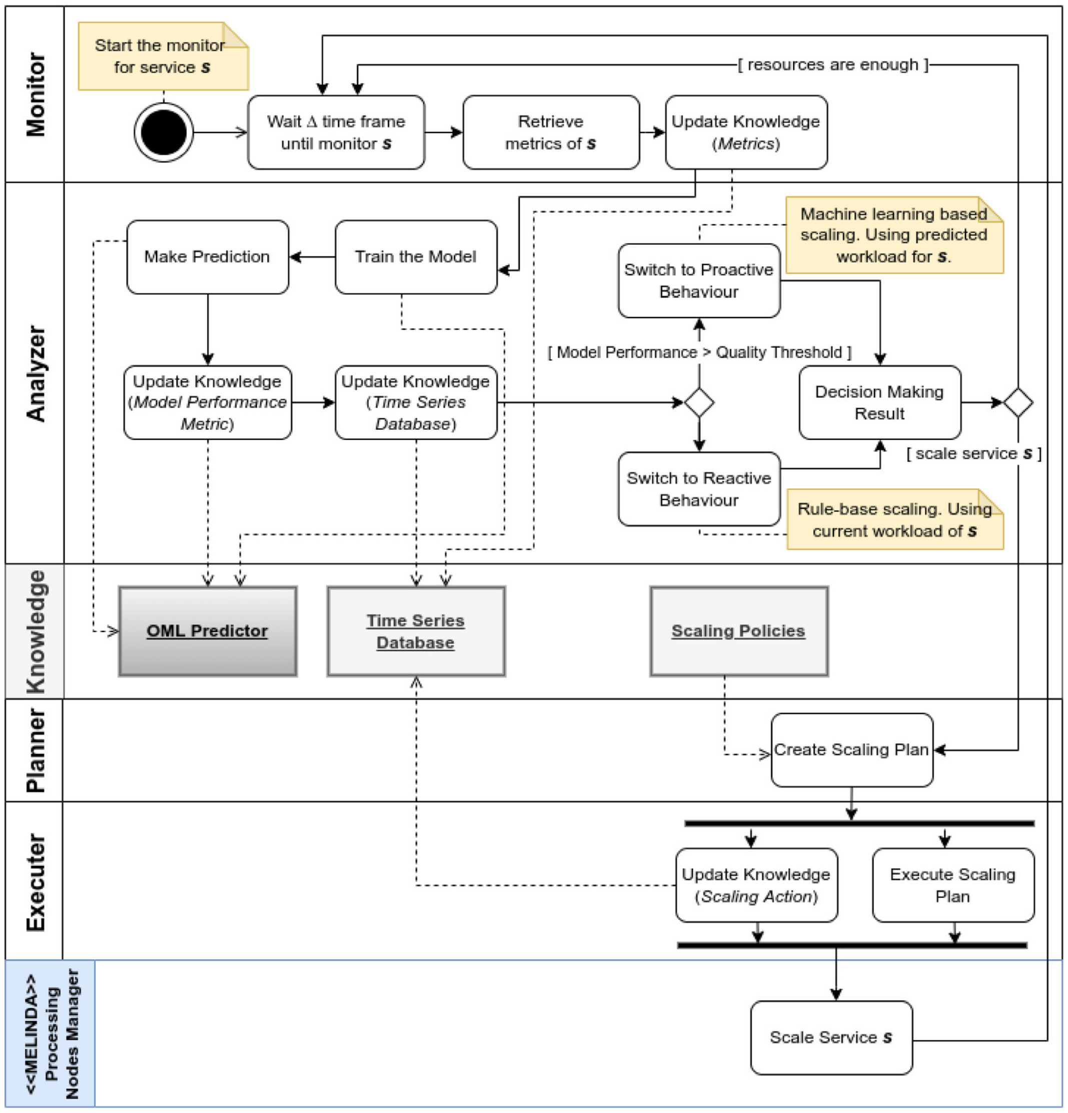

Monitor
Start the monitor for service s
Wait Δ time frame until monitor s
Retrieve metrics of s
Update Knowledge (Metrics)
[resources are enough]
Analyzer
Make Prediction
Train the Model
Machine learning based scaling. Using predicted workload for s.
Switch to Proactive Behaviour
[Model Performance > Quality Threshold]
Update Knowledge (Model Performance Metric)
Update Knowledge (Time Series Database)
Decision Making Result
[scale service s]
Switch to Reactive Behaviour
Rule-base scaling. Using current workload of s
Knowledge
OML Predictor
Time Series Database
Scaling Policies
Planner
Create Scaling Plan
Executer
Update Knowledge (Scaling Action)
Execute Scaling Plan
<<MELINDA>> Processing Nodes Manager
Scale Service s

Fig. 4. UML activity diagram represents the scaling process performed by the proposed auto-scaling subsystem.

services, increasing or decreasing the number of service instances. It is important to note that other types of adaptation decisions are involved in an autonomic system, such as resource migration. However, our work is focused on scaling, which is just one example of adaptation. Other types of adaptation would give rise to other subsystems with the same interfaces but dealing with other types of adaptation decisions.

Fig. 3 illustrates the auto-scaling subsystem architecture. Each phase in the MAPE-K control loop contains different activities to be performed to realize the autonomous loop. All the phases access a shared knowledge base fundamental to decision-making. Thus, the components of our architecture are responsible for implementing the functions of the control loop. Fig. 4 illustrates a UML activity diagram that represents the dynamic aspects of the proposed auto-scaling subsystem to scale a service. The scaling process performed by the auto-scaling subsystem comprises some activities and various decision paths. Each component of the auto-scaling subsystem architecture is described in the following.

3.1. Monitor

The first phase of MAPE-K is performed by the *Monitor* component that constantly monitors the state of services maintained by MELINDA. For effective scaling decisions, an auto-scaling system must rely on a reliable and robust monitoring strategy that monitors the utilization of the edge node resources and the state of the services. Part of the state monitored by the *Monitor* corresponds to incoming requests to the service. The *Monitor* also collects the *resource* *metrics* from virtual resources used to run services, such as vCPU and vMemory utilization. Such data are used by the *Monitor* to calculate the *performance metric* of the manageable service. The purpose of collecting such metrics is to ensure the SLA between the service provider and end-users through QoS requirements. We deal only with the workload demand as the main scaling indicator in our work. We assume a direct mapping between the number of incoming requests and the workload of the service. In MELINDA, the *Processing Task* component is responsible for running the services and managing their life cycle. In this way, through the *iData* interface, *Monitor* retrieves the information related to the service and virtual resources used to run the services. All collected metrics are stored in *Knowledge* component as represented in Fig. 4.

At each time frame Δ, the *Analyzer* component is informed of the metrics obtained from the monitored services. The architecture is flexible enough to define different values for Δ, varying from milliseconds to hours, depending on the type of service. In this way, the entire control loop must execute in a time span shorter than Δ. For example, $\Delta = 1min$ implies measuring all metric data per minute for all monitored entities.

3.2. Analyzer

The *Analyzer* component is in charge of using the metrics (symptoms) gathered in the previous phase (performed by *Monitor*) as input for the decision-making processes. Such metrics are sent to the *Analyzer* via the *iAnalyzer* interface. This component handles the monitored data, filtering and grouping them to create time series representing the history of the measured data. The history of measured data represents the states of the system over time. Thus, the *Analyzer* component compares the desired state for the system (as defined by the stakeholders/managers and stored in the knowledge base) with the verified state and decides if a change is needed. A change is always aligned with the system's goals and policies (also stored in the knowledge base). Whenever it is necessary to make a change (adaptation), a change request is sent to the *Planner* component.

The reasoning process for decision-making is based on the current and predicted workload. We modeled the scaling as a regression predictive modeling problem [24], described as the mathematical problem of approximating a mapping function (f), called function approximation, from input variables X (observations) to a continuous output variable y (target value). Therefore, we employed the OML model to predict the future workload and required computational resources. Such required resources are specified in the scaling plan to serve as the basis for scaling operations performed by MELINDA.

The *Analyzer* decides the behavior (reactive or proactive) in which the auto-scaling acts to trigger a scaling operation. This aspect is essential to the scaling process and fundamental for "switching" from reactive to proactive behavior and vice versa. At each "training", the parameters are updated, and the metric that represents the model's performance is also updated, i.e., we recalculate such a metric. Thus, if the model makes inaccurate predictions, the metric representing the model's performance will be degraded every time such a metric is updated. Therefore, auto-scaling can switch its behavior to reactive until the model becomes "good" again.

The metrics for evaluating predictions made on regression problems can be used as the model performance. Such metrics involve calculating an error score to summarize the predictive skill of a model. Our proposed auto-scaling currently supports MAE, MSE, and R^2 as metrics representing the model's performance. The system operator chooses which metric is best to use depending on the problem domain. For example, using R^2 as the metric and setting it to 0.92 as the minimum acceptable value (*quality threshold*), at time t if the calculated R^2 is less than 0.92, the scaling process has the reactive behavior and only "looks" at the current workload. Otherwise, the scaling process uses the predicted (future) workload to take some scaling action. This strategy aims to reduce prediction errors when the prediction model is not accurate enough.

In Algorithm 1, at time t, the parameter $past\_workloadd\_predicted$ is the predicted workload at time $t - \Delta$ while $workload\_current$ is the current workload (real or ground truth) at time t. That is, $past\_workload\_predicted$ refers to the workload that the OML model predicted at time step $t - \Delta$ while $workload\_current$ represents the workload observed at time t. In an ideal scenario, where the OML is well-fitted, producing more accurate outcomes, $past\_workload\_predicted$ will equal $workload\_current$. Auto-scaling must always only store the last value of the workload. It is important to note that the OML predictor does not require storing all observations. In fact, this is one of the advantages of using such a method [25]. The parameters are used to train the OML predictor and update its performance metric. If the metric's value representing the predictor's performance is greater than the threshold (*quality threshold*), then the auto-scaling changes its behavior to proactive and sets the variable $workload\_demand$ to the predicted load. Otherwise, the auto-scaling uses the current workload as the workload demand.

In addition to creating time series and reasoning for decision-making, the *Analyzer* is also in charge of: (a) updating the value of the metric that represents the OML model performance whenever a new observation arrives; (b) training the OML model by updating the model parameters with a new observation; and (c) making predictions of future workload. These activities update the shared knowledge *Online Machine Learning Predictor* and *Time Series Database* maintained by *Knowledge*, as illustrated in Fig. 4.

3.3. Knowledge

The *Knowledge* is responsible for maintaining and sharing the system knowledge, denoting all information relevant to the adaptation process, such as configurations, metrics, symptoms, policies, scaling plans, prediction models, and their

```
Function CalculateWorkloadDemand(workload_current, past_workload_predicted)
    $workload\_demand \leftarrow \emptyset$
    $model.train(workload\_current, past\_workload\_predicted)$
    $metric.update(workload\_current, past\_workload\_predicted)$
    if $metric.current\_value() > metric.threshold()$ then
        $workload\_demand \leftarrow model.predict\_next()$
    else
        $workload\_demand \leftarrow workload\_current$
    end
    return $workload\_demand$
```

Algorithm 1: Calculating the workload demand and changing the behavior according to the Predictor's performance metrics.

corresponding performance metrics. Such knowledge is represented in Fig. 4 as artifacts shared between activities. This component provides the *iUpdate* and *iProvider* interfaces through which the components can update or retrieve shared knowledge during all stages of the control loop.

Fundamental knowledge stored and maintained in *Knowledge* are the policies, which consist of a set of behavioral constraints and preferences that influence the decisions the auto-scaling makes. Via the provided *iUpdate* interface, the *Knowledge* also obtains resource-specific knowledge from external sources and the parameters used during the auto-scaling process, e.g., thresholds values, the value of model performance metric, etc.

The metric that measures the prediction model's performance is a key element in the proposed auto-scaling because the decision on which approach (reactive or proactive) to use at any given time is based on the value of this metric. Suppose the model performance metric indicates that it does not meet a certain "***quality threshold***". In that case, the system will prioritize the reactive approach based on rules to perform scaling actions. In the meantime, the OML model continues to be trained, and the metric that indicates its performance is updated until they reach the desired quality threshold.

3.4. Planner

After the analysis phase performed by the *Analyzer* component, if any action needs to be taken to achieve the system's objectives/goals, the *Analyzer* sends a change request to the *Planner* component via the *iPlanner* interface. The *Planner* component structures the actions required to change the service, i.e., it creates a scaling plan. We are considering a direct mapping between the workload of a given service and the number of images (frames) it can process. Moreover, the processing capacity of edge nodes in terms of Frames per Second (FPS) is known in advance. FPS is a metric that measures the frame rate, i.e., the number of images consecutively processed per second by a device.

Algorithm 2 represents the process of creating the scaling plan considering the current capacity of a given service to deal with the workload and the current quality of the prediction model (i.e., the performance of the predictor model). This algorithm receives as input the current capacity of all replicas of a given service in terms of workload fulfillment ($system_{capacity}$), and the demanded workload ($workload\_demand$). It also receives, as inputs, the following shared knowledge defined by the system operator: the limits for triggering scaling operations ($THRESHOLD_{up}$ and $THRESHOLD_{down}$), the cool-down timer CDT, and the scaling ratio ($RATIO$).

```
Function CreateScalingPlan($system_{capacity}$, workload_demand, $THRESHOLD_{up}$, $THRESHOLD_{down}$, CDT, RATIO)
    $scaling\_plan \leftarrow \emptyset$
    if $system_{capacity} * THRESHOLD_{up} > workload\_demand \,\&\, timedOut(CDT)$ then
        $scaling\_plan \leftarrow ScaleIn(system_{capacity} - workload\_demand) * RATIO$
        $reset(CDT)$
    else if $system_{capacity} * THRESHOLD_{down} < workload\_demand$ then
        $scaling\_plan \leftarrow ScaleOut(workload\_demand - system_{capacity})$
        $reset(CDT)$
    return $scaling\_plan$
```

Algorithm 2: Creating a Scaling Plan according to the demanded workload.

We adopted a conservative strategy every time a scaling-down action is performed by waiting a fixed amount of time between each scaling-down process as proposed in [26]. The *CDT* parameter denotes this amount of time between scaling-down processes. The system administrator is in charge of setting the value for this parameter.

The cool-down timer aims to reduce the provision oscillations, a situation where repeated contradictory scaling operations are performed. In addition, the parameter decay factor (*RATIO*) defines the ratio of resources to be removed in case of scaling down operation. In the previous performance evaluation [10], these strategies (cool-down timer and decay factor) significantly reduced QoS degradation, mainly because several oscillations occur in a short period. If a given service's current capacity (in terms of FPS capacity) is greater than the demanded workload, the scaling down is triggered. This action is executed if the *CDT* has timed out, i.e., the time interval between two successively scaling processes is greater than *CDT*. Besides, $THRESHOLD_{up}$ and $THRESHOLD_{down}$ enable the system operator to set different limits for triggering the scaling process. Such parameters range from 1% to 100% and $THRESHOLD_{up}$ is always greater than $THRESHOLD_{down}$.

$THRESHOLD_{up}$ defines the upper limit for the current capacity (i.e., the current number of resources) concerning the demanded capacity, while $THRESHOLD_{down}$ defines the lower limit. To better understand how these thresholds work, let us give an example. Let us consider the processing capacity of a given service measured in FPS. If the system operator sets $THRESHOLD_{down}$ to 40% (0.4), the current capacity of the service is 300 FPS, and the demanded capacity ($workload\_demand$) is 100 FPS, then a scaling-down must be triggered since the current capacity is almost three times the demanded capacity (33.3%). Therefore, auto-scaling must adjust the number of resources to avoid waste. In this case, the amount of resources that will be deallocated from the service is given by a ratio (parameter *RATIO*) of the exceeding capacity of the service ($system_{capacity} - workload\_demand$). If *RATIO* is set to 70% (0.7), then 70% of exceeding capacity is deallocated. Considering another example where the $THRESHOLD_up$ is set to 90%, the current capacity of the service is 300 FPS, and the demanded capacity is 280 FPS, according to the second conditional statement, a scaling-up is required since 280 FPS represents 93% of the current capacity.

Functions *ScaleIn* and *ScaleOut* specify, in the scaling plan, that scaling in/out operations need to be performed to adjust the number of instances of a given service to meet the demand. Every time these functions are called, the CDT counter is reset. Finally, the scaling plan is sent to the *Executer* component via the *iScaling* interface and the *Knowledge* is updated. Information about the scaling performed (in or out) and the state of the service before and after the scaling operation is stored in component *Knowledge*.

3.5. Executer

The *Executer* component is in charge of interacting with the MELINDA *Processing Nodes Manager* component, via the *iScalingCmd* interface, in order to send the scaling plan. The *Executer* component is in charge of performing the execution phase of the MAPE-K control loop. The scaling plan specifies the changes that must be executed to change the service's behavior (i.e., the number of service instances). The *Processing Nodes Manager* component is responsible for interacting with the MELINDA Orchestrator, sending commands to provision or de-provision edge nodes to run instances of the services. Part of the execution of the change plan involves updating the information maintained by the *Knowledge* component. In this way, as illustrated in Fig. 4, the state of a particular service is registered in the knowledge base before and after a given scaling process. This kind of information can become very useful in assessing the auto-scaling process. Many works deal with auto-scaling as a classification problem in ML [6,17,26–28], and we can use such information to improve our auto-scaling by using both the regression and the classification models. We intend to investigate how to combine these two types of modeling in machine learning to improve auto-scaling.

The logic for the MAPE-K components is represented in Fig. 3, and basically, each component runs a simple algorithm using only decision statements. The time complexity for such algorithms is quite simple because they do not involve any loop with significant data arrays. The number of elementary operations performed by the algorithm is low. The relevant computational time of our approach is related to the online machine learning algorithm to predict the number of requests in a time window, as demonstrated in the next section.

4. Performance evaluation

This section presents the performance evaluation of the proposed auto-scaling. We have extended the preliminary assessments performed in [10], introducing new evaluation metrics and a practical application scenario. This section is organized as follows. First, we present the application scenario and the workloads used during the experiments in Section 4.1. Section 4.2 presents the experimental setup, while the metrics used to assess the auto-scaling approach are present in Section 4.3. Finally, the results and analysis are discussed in Section 4.4. The source code of the experiment is available online.[2]

4.1. Application scenario and workloads

To evaluate the auto-scaling strategy in the MELINDA architecture, we created a scenario of an intelligent video analytics application for smart cities. The application is an Automatic License Plate Recognition (ALPR), which comprises a pipeline of three distinct tasks, as illustrated in Fig. 5. The first task (MLT) detects the application object of interest (vehicles) from multiple video streams, producing for each stream an Image of Interest Message (IIM). This message contains the vehicles' image as the data flow in the task pipeline. The second task (FLT) extracts the license plate image(s) from the original image. Finally, the third task (DLT) processes this/these plate image(s) to recognize the plate's characters and identify the car, yielding the application event of interest.

[2] Experiment source code - https://colab.research.google.com/drive/1dtYOjVEzi6dceb2NwSyy2Lf-kqWPVQgF?usp=sharing.

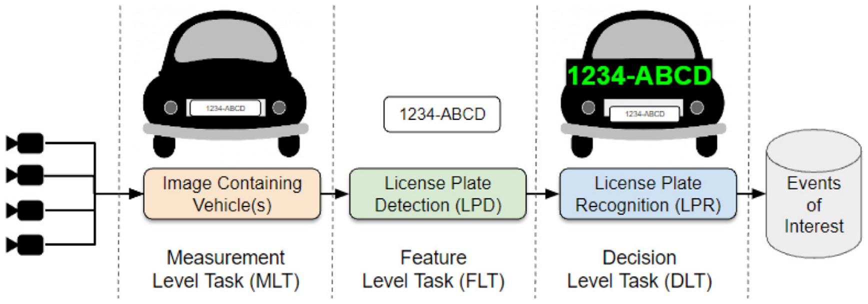

1234-ABCD
1234-ABCD
1234-ABCD
Image Containing Vehicle(s)
License Plate Detection (LPD)
License Plate Recognition (LPR)
Events of Interest
Measurement Level Task (MLT)
Feature Level Task (FLT)
Decision Level Task (DLT)

Fig. 5. Task pipeline for the evaluation application scenario.

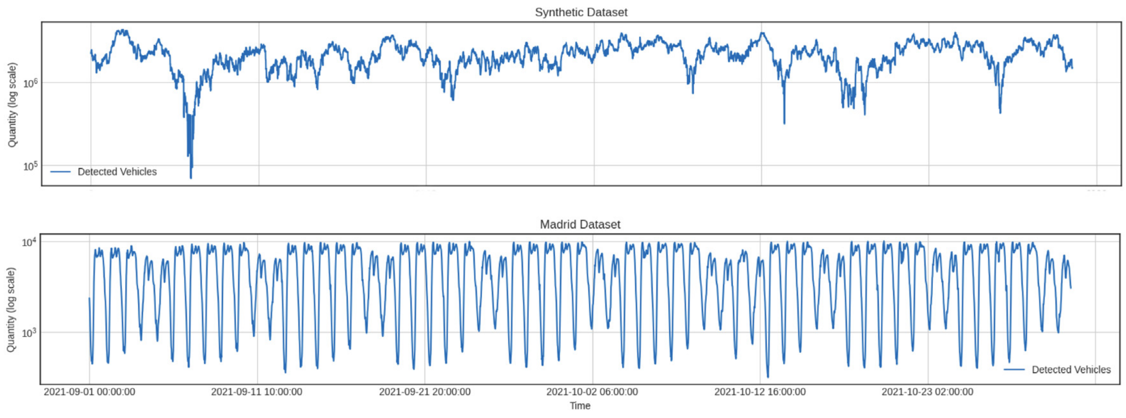


Fig. 6. Datasets used in the simulation representing the number of vehicles detected in September and October 2021.

Each task in the pipeline uses a Deep Learning (DL) model, which typically makes intensive use of resources. Thus, it is typical for a hardware-intensive job to make use exclusively of the entire resources of an edge node in this kind of task. The number of nodes to run the first task (Vehicle Detection) is pre-determined, since all cameras produce a known and fixed number of frames per second. The second and third tasks, running on workers and consumer nodes, respectively, will run only when the first task (on the producer) yields an IIM. In this case, the demand of worker and consumer nodes depends on the cameras' event occurrence. In such a scenario, an event is a vehicle passing by the camera. The number of events constantly varies according to the time of day (e.g., dawn and rush time) and the day of the week. In this context, we apply our auto-scaling approach to adjust the number of (worker) nodes according to the event's frequency and workload dynamics. The workers are in charge of running replicas of the last two tasks of the pipeline.

To simulate an actual workload of such a processing demand, we used the Madrid traffic dataset[3] and a synthetic dataset, as illustrated in Fig. 6. The Madrid dataset contains the historical data of the traffic measuring points in Madrid. Each measurement point provides the number of vehicles detected every 15 min. For the experiments, we used data from September and October 2021. Four thousand three hundred forty-two measurement points performed more than 30 million vehicle detections throughout the months. This dataset contains only the number of vehicles passed through the measurement points. We used this dataset to simulate the event's frequency, where an event is a vehicle detection performed by the node running the MLT task.

A maximum peak of about 3 million vehicles is observed in a 15-minute interval considering all 4342 detection points in the dataset. This way, on the peak, about 690 vehicles were detected by each detection point within 15 min, corresponding to 46 vehicles per minute. This large number is reasonable since several detection points are placed close to each other. Therefore, detection points on the same avenue may have detected the exact vehicle multiple times in the same interval. Considering all points, the average vehicle count is around 1.5 million vehicles, while the minimum observed value is 78 thousand vehicles per 15 min. Fig. 6 shows the distribution of the datasets over September and October 2021.

Fig. 7 illustrates the aggregation of the Madrid dataset, considering grouping by the day of the week and the hour of the day. Wednesday (2) and Thursday (3) are the days with the highest volume of detected vehicles, while Saturday (5) and Sunday (6) are the days with minor traffic. An initial analysis might suggest that a simplistic auto-scaling strategy

3 Madrid Open Data - https://datos.madrid.es/portal/site/egob.

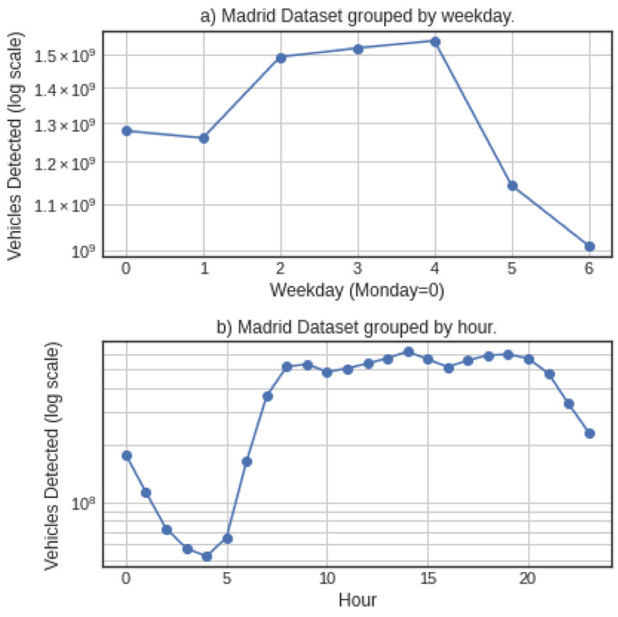


Fig. 7. Analysis of Madrid dataset.

can take advantage just by decreasing the number of resources on weekends in order to save resources. However, when analyzing the number of vehicles detected during the hours of the day, it is observed that around 5 a.m., the volume of detected vehicles is minimal. Therefore, auto-scaling must also consider the hours of day.

The Madrid dataset has visible behavior regarding the number of vehicles detected throughout the day and on weekdays. The machine learning model must understand such data behavior to make assertive predictions. We introduced a synthetic dataset with no visible behavior in the data to measure the ability of auto-scaling to handle non-visible patterns. The synthetic dataset was generated using TimeSynth,[4] an open-source library for synthetic time series generation. We applied the Continuous AutoRegressive (CAR) signal generation with red noise signal to create an irregular time series. We used 0.9 and 0.01, respectively, as the parameter for the AR(1) process and the average deviation of the signal. As illustrated in Fig. 6 the synthetic dataset contains data corresponding to the same interval (September to October 2021) as the Madrid dataset. A non-deterministic random process generated the time series values ranging from 70 thousand to 4 million. This range corresponds to the minimum and maximum values in the Madrid dataset. Such a dataset has no visible pattern in the independent variables, as presented in the Madrid dataset. Therefore, the purpose of using such a synthetic dataset is to evaluate the ability of the auto-scaling to adapt to a non-well-shaped workload pattern. The synthetic dataset generation process and the preprocessing of the datasets used in this work are available online.[5]

4.2. Experimental setup

The pipeline of the video analytics application encompasses a set of deep learning tasks. Such an end-to-end AI solution for license plate recognition applications requires processing nodes to execute these tasks. This way, we simulated a hierarchical edge computing environment, illustrated in Fig. 8, consisting of smart city monitoring cameras as video stream sources at the base of the hierarchy, heterogeneous single board computers as processing nodes in the middle, and the cloud as a data sink at the top.

Nowadays, many cameras have embedded AI hardware capable of running neural networks [29,30] for detecting objects of interest. Thus, executing the MLT task (Vehicle Detection) directly on such cameras is reasonable. We simulated heterogeneous edge nodes with different processing powers to perform the FLT task (License Plate Detection). This experiment simulates 17 NVIDIA edge nodes with various processing capacities in terms of FPS. We gathered the processing capacity in FPS of such nodes to run the deep learning LPR tasks from the NVIDIA hardware performance reported in [31] and presented it in Table 1. Using all the seventeen edge nodes to execute the second task (LPD) simultaneously, the system can process up to 3400 FPS. So, the auto-scaling approach scales the number of nodes to run the second task of the pipeline as a function of load variation to optimize the use of resources.

4 TimeSynth library - https://github.com/TimeSynth.

5 Datasets Preprocessing - https://colab.research.google.com/drive/1gzqScAjFlQtKQFkUABP33aAqQAavpG6Z?usp=sharing.

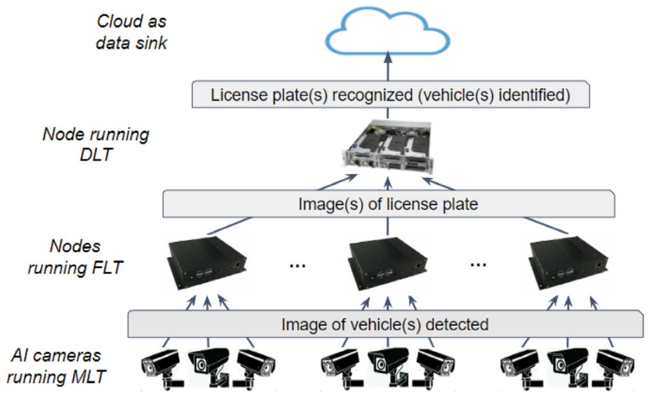

Cloud as data sink
License plate(s) recognized (vehicle(s) identified)
Node running DLT
Image(s) of license plate
Nodes running FLT
Image of vehicle(s) detected
AI cameras running MLT

Fig. 8. Hierarchical edge computing topology simulated in the experiment.

Table 1
Types of edge nodes and their capabilities in terms of FPS to perform LPD and LPR tasks.

Device	Task	FPS	Frames per 15 min	Quantity
Jetson Nano	LPD	66	59,400	10
Jetson NX	LPD	461	414,900	5
Jetson Xavier	LPD	913	821,700	2
T4 Accelerator	LPR	3821	3,438,900	1

The DLT task (LPR) runs on an edge server node with an Nvidia T4 GPU accelerator, with inference throughput up to 3821 FPS for this task. In 15 min, such a node can process up to 3,438,900 frames. A single node with a T4 accelerator can already handle all the output of the second pipeline task, considering the maximum simulated workload demand, as the maximum peak in the dataset is smaller than the node capacity. Although our scenario uses proprietary and GPU-equipped devices, the proposed auto-scaling can be employed in other systems with general-purpose edge nodes running workflows from different applications.

For this simulation, only the FLT task (License Plate Detection) is scalable considering the variable occurrence of vehicles detected. This way, only edge nodes performing FLT tasks (LPD) are scaled. A resource-intensive deep learning task like LPD uses all the node's resources; therefore, we assumed that there was only one task per node. This way, having the inference performance throughput in frames per second of each edge node, the auto-scaling strategy predicts the workload so that MELINDA Orchestrator can allocate and deallocate edge nodes to execute instances of the LPD task. We converted to frames per 15 min to attend Madri's dataset's time window.

Although we have chosen to employ auto-scaling in only one tier of the proposed hierarchical topology, we can incorporate auto-scaling in the other tiers when needed. For example, we can replace the edge server node running the DLT task (Nvidia T4 GPU accelerator) in the proposed scenario for some resource-constrained edge nodes. This way, auto-scaling will also ensure the elasticity property of edge nodes running DLT tasks.

In this experiment, the KNN Regressor algorithm (KNN) [32] was employed by auto-scaling as the online machine learning predictor. KNN is a non-parametric regression method that keeps the last training samples, and predictions are obtained by aggregating the values of the closest neighbors. We defined the features presented in Table 2 as the input to the online model. Such independent variables were defined based on the principles of feature engineering by using domain knowledge to select and transform the most relevant variables. There is a visible pattern in workload behavior over the hours of the day and the day of the week in the Madrid dataset. Therefore, the features chosen try to capture such a behavior. Thus, the selected features capture the temporal aspects so that the auto-correlation information is not lost. All features are treated as numerical, and the KNN algorithm was set to use a window of training samples with size 672, corresponding to seven days of observations. Also, according to the recommendations of [33], the number of nearest neighbors – parameter k – for KNN algorithm was set to five. We adopted the Euclidean distance function, and the mean was used to aggregate the target values of neighbors (weighting function). The time complexity of the k-NN algorithm, according to [33], is $O(ndk)$, where n is the number of training examples and d is the distance computation required for each training.

The windowing method [34,35] was employed so that the observations at prior time steps (lag observations) are input to the model to predict the observation at the current time step. Let $X(t+1)$ be the value in a time series at time $t+1$. The idea is to predict $X(t+1)$ using not only $X(t)$ but $X(t-1)$, $X(t-2)$ until $X(t-4)$. Therefore, the last five readings of the workload (variables w_0, w_1, w_2, w_3, and w_4) were also used as input for the model to pick up the trend and seasonal components of time series. As a model performance metric, we employed R^2 (R-squared) and set 0.70 (70%) as the *quality threshold*, *CDT* was set to 10, *RATIO* was set to 0.7 (80%), and $THRESHOLD_{up}$ and $THRESHOLD_{down}$ were set to 90% and 50%, respectively. The choices of these values were motivated by the experiments performed in [10].

Table 2
Independent variables for the online machine learning prediction model.

Features	Description
hour	Observation hour
day	Day of observation
weekday	Day of week
w_0	Workload at time t
w_1	Workload at time $t-1$
w_2	Workload at time $t-2$
w_3	Workload at time $t-3$
w_4	Workload at time $t-4$

A R^2 value above 0.8 indicates that the model can explain 80% of the variability in the outcome data. As the auto-scaling uses the same model to deal with unknown workloads, it is reasonable that such a model cannot account for at least some variability in the data. Using a high value for parameter *quality threshold* implies that proactive behavior is used less frequently. Using CDT equal to 10 means that scaling down operations will be performed successively only after 150 min (2.5 h) if there is no scaling up operation during this interval. Thus, CDT ensures that an edge node will be available once allocated to process the workload for at least 150 min. Parameters $THRESHOLD_{up}$ and $THRESHOLD_{down}$ define the load range of the allocated edge nodes, and the values for such parameters were chosen based on the values typically used by Cloud operators.

We preprocessed the dataset by applying feature normalization to scale numerical variables, putting all of them on the same scale to improve the algorithm's performance. Besides, we encode categorical features as a one-hot numeric array (*OneHotEncoder*). It is important to note that all the preprocessing performed on the dataset can also be performed in a real edge computing environment for data stream processing, as required by our architecture.

4.3. Auto-scaling performance metrics

In order to evaluate the auto-scaling, we used *Degraded QoS Metric* [27] as the cumulative time (minutes) a service spends in a state of QoS violation. Also, we used the metrics (*Provisioning Accuracy*, *Wrong Provision Time Share*, and *Elasticity Speedup*) proposed by [16]. Such metrics will be described in the following.

The experiment duration is defined as T, and an instant of time is represented as t, such that $t \in [1, T]$. The resource supply at time t is represented as s_t, and d_t is the demanded resource units at time t. In turn, Δt denotes the time interval between the last and current changes in demand d or supply s. The resource supply s_t is the monitored number of resources at a time t, corresponding to the system's current capacity.

4.3.1. Degraded QoS

The degraded QoS ψ is increased when the auto-scaling fails to allocate enough resources (nodes) to handle the demand. This metric represents the SLA violations and is calculated based on the number of times the demand is not met, as presented in Eq. (1). The range of this metric is the interval $[0, \infty)$, where 0 indicates that the auto-scaling has associated enough resources for all experiment duration.

$$\psi = \sum_{t=1}^{T} diff(d_t - s_t)\Delta t, \quad \text{where} \quad diff(n) = \begin{cases} 1, & \text{if } n > 0 \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Considering that the observations in the dataset comprise a 15-minute interval, the total time in minutes that the system did not meet the SLA corresponds to $\psi * 15$.

4.3.2. Provisioning accuracy

The provisioning accuracy θ_U and θ_O identify the number of underprovisioned and overprovisioned resources during the experiment. Such metrics are calculated as defined in Eq. (2). The range of these metrics is the interval $[0, \infty)$.

$$\theta_U = \frac{100}{T}\sum_{t=1}^{T}\frac{max(d_t - s_t, 0)}{d_t}\Delta t, \qquad \theta_O = \frac{100}{T}\sum_{t=1}^{T}\frac{max(s_t - d_t, 0)}{d_t}\Delta t, \tag{2}$$

The system is always expected to be overprovisioned. However, the amount of overprovisioned resources is expected to be as small as possible to meet the SLA and minimize wastage.

4.3.3. Wrong provisioning time share

τ_U and τ_O are metrics defined in Eq. (3) and represent how long the system was overprovisioned and underprovisioned, respectively. The range of these metrics is the interval [0, 100].

$$\tau_U[\%] = \frac{100}{T}\sum_{t=1}^{T} max(sgn(d_t - s_t), 0)\Delta t,$$

$$\tau_O[\%] = \frac{100}{T}\sum_{t=1}^{T} max(sgn(s_t - d_t), 0)\Delta t, \tag{3}$$

where

$$\tau_U + \tau_O = 100\% \text{ and } \mathrm{sgn(n)} = \begin{cases} -1, & \text{if } n < 0 \\ 0, & \text{if } x = 0 \\ 1, & \text{if } x > 0 \end{cases}$$

As stated earlier, the system is expected to be overprovisioned, and the amount of extra resources is minimal.

4.3.4. Elastic speedup

Finally, the elastic speedup represents the gain of an approach a compared to other n considering the geometrical mean of the metrics defined above. The approach a is better than n if the result in Eq. (4) is greater than 1.

$$\epsilon_n = \left(\frac{\theta_{U_n}}{\theta_{U_a}}.\frac{\theta_{O_n}}{\theta_{O_a}}.\frac{\tau_{U_n}}{\tau_{U_a}}.\frac{\tau_{O_n}}{\tau_{O_a}}\right)^{\frac{1}{4}} \tag{4}$$

4.4. Analysis

The proposed auto-scaling (henceforth called *Hybrid*) was evaluated by comparison with two baseline approaches (*noScale-Min* and *noScale-Max*) and a reactive approach that implements rule-based scaling. The *noScale-Min* and *noScale-Max* approaches did not employ any scaling method and used a fixed quantity of resources during the experiment, i.e., both approaches used a fixed number of edge nodes, which correspond directly to the frame processing capacity of the system to meet the demand. The *noScale-Min* used few edge nodes and the *noScale-Max* used all edge nodes.

Considering the application workloads, as illustrated in Fig. 6, the number of resources used by *noScale-Min* approach was relative to the average image processing demand, i.e., around 1.5 million vehicles per 15 min. Thus, we simulated only two Jetson Xavier edge nodes in such an approach, capable of processing 1,643,400 frames in 15 min. The *noScale-Max* approach used all edge nodes during the experiment. Thus, the total processing capacity in frames per 15 min is up to 4,311,900 frames, meaning that the resources are more than sufficient to meet the maximum demand of the application workloads. The reactive approach is rule-based and only performs scaling when the demand exceeds the system's capacity, considering the currently allocated resources.

4.4.1. Analysis of the auto-scaling approaches

The auto-scaling performance metrics were measured and summarized in Table 3. Considering the QoS degradation ψ, Fig. 9 illustrates that our approach (Hybrid) obtained less QoS degradation in Madrid and Synthetic datasets. Using the Madrid dataset as workload, the *Reactive* approach reached 237 QoS degradation. A decrease of 32 QoS degradation or a 13.5% improvement in auto-scaling was observed using *Hybrid* approach. Using the *noScale-Min* approach, 2813 QoS degradation was observed due to the lack of auto-scaling, which caused insufficient resources to meet the demanded workload. Although the *noScale-Max* approach does not cause any QoS degradation, many resources were wasted.

Using the synthetic dataset, according to Fig. 9, the *Hybrid* approach was not able to allocate enough resource to the system in 47 observations, which correspond to a 41.9% decrease in QoS violations compared to the *Reactive* approach (87 QoS degradation). The *Hybrid* approach could allocate edge nodes to meet the demand of 99.2% of the total simulation time. The *noScale-Min* reached 3644 QoS degradation, corresponding to 60.5% of the observations in the Synthetic dataset. In both workload patterns, the *Hybrid* approach performed better than the others approaches.

The amount of scaling operations (actions) performed by the auto-scaling approaches during the experiment is illustrated in Fig. 10. Considering the Madrid dataset, the *Hybrid* approach performed 650 actions against 770 actions performed by the *Reactive* approach. Considering the Synthetic dataset, the *Hybrid* approach performed 313 scaling operations (respectively 130 and 202 scaling up and scaling down actions). The *Reactive* approach required 626 scaling operations and was not able to achieve better performance than the *Hybrid* solution. It is important to note that each scaling operation has an associated computational cost — that we are not considering in the experiment. For example, the scaling operation may involve time-consuming tasks performed by the orchestrator or the virtualized infrastructure manager to provision or de-provision resources. In this way, the fewer scaling operations to meet the demanded workload, the better the auto-scaling will be.

The amount of scaling operations is directly related to the granularity of the auto-scaling activation interval, which we represent in our system model as Δ (time frame). Auto-scaling will be activated at spaced intervals if Δ is a large value,

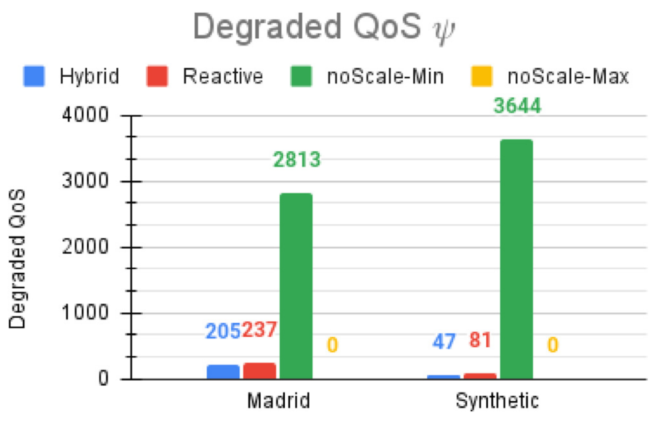


Fig. 9. Quantity of QoS degradation during the experiment.

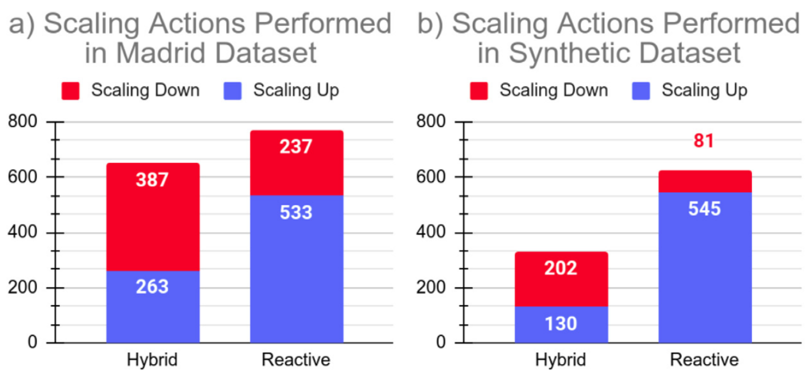


Fig. 10. Quantity of scaling operations performed by approaches during the experiments.

Table 3
Comparison of different approaches considering the Madrid and Synthetic datasets.

	Hybrid	Reactive	noScale-Min	noScale-Max
Madrid dataset				
Degraded QoS ψ	205	237	2813	0
Scaling operations	650	770	0	0
θ_U	0.27	0.27	11.27	0
θ_O	204.45	203.14	176.67	576.11
τ_U	3.50	4.05	48.04	0
τ_O	96.50	95.95	51.96	100
ϵ	4.06	3.92	1	–
Synthetic dataset				
Degraded QoS ψ	47	81	3644	0
Scaling operations	313	626	0	0
θ_U	0.04	0.06	16.06	0
θ_O	60.63	56.77	20.67	166.48
τ_U	0.92	1.38	62.21	0
τ_O	99.07	98.61	37.78	100
ϵ	7.47	6.37	1	–

leading to fewer scaling operations. However, during the interval, oscillations in the workload may have occurred, causing system degradation. On the other hand, using a small granularity for auto-scaling activation implies that auto-scaling will have the opportunity to control small fluctuations in the workload, leading to more scaling operations. Controlling the auto-scaling activation interval is a challenging task. Some work [6,36,37] uses dynamic window-based monitoring strategies to control auto-scaling activations. In this sense, the interval between each activation of auto-scaling decreases when oscillations or SLA violations occur. In contrast, the interval increases at times of workload stability.

In the scenario we are dealing with, the worst situation occurs when the system allocates fewer resources than required. Underprovisioning harms the system because processing each Image of Interest (IIM) can generate an event

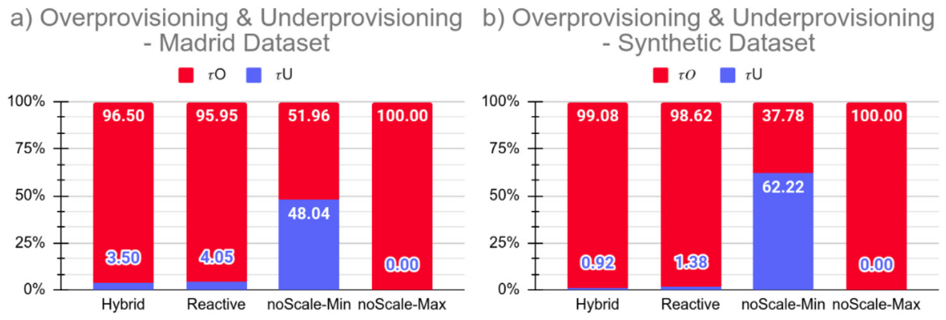


Fig. 11. Underprovisioning and overprovisioning during the experiments.

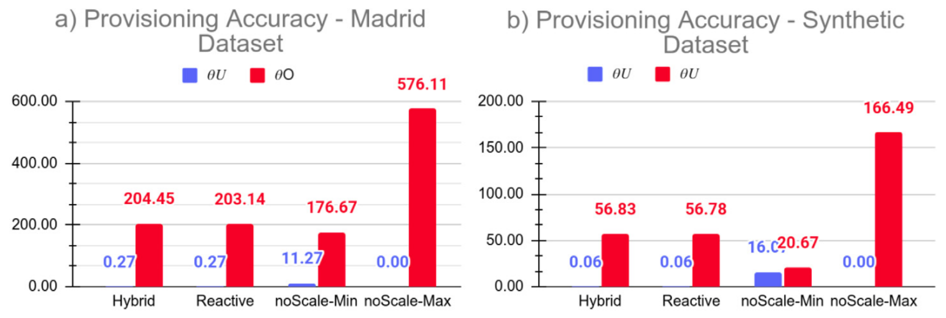


Fig. 12. Provisioning Accuracy of the approaches during the experiments.

that needs to be processed as quickly as possible. When the number of computational resources is insufficient to meet the demand, there is a delay in processing the images of interest. Oppositely, overprovisioning must also be avoided as much as possible. Allocations of more edge nodes than necessary to meet the demand cause wastage, leading to financial costs, for example.

Fig. 11 illustrates the amount of time – in percentage – that the system was underprovisioned and overprovisioned, i.e., the wrong provisioning time share τ_U and τ_O. When using the Madrid dataset as workload, *Hybrid* approach has caused the system to be under-provisioned 3.50% of the duration of the experiment. While using the *Reactive* approach, the system was underprovisioned for 4.05%. Using the baseline approach, *noScale-Min*, the system was underprovisioned for almost half the experiment time (48.04%). As expected, the baseline approach *noScale-Max* did not show any underprovisioning. Considering the Synthetic dataset, *Hybrid* auto-scaling was able to reduce under-provisioning to 0.92%, while *Reactive* auto-scaling reduced to 1.38%. The *noScale-Min* obtained the worst result, 62.22%.

Despite the *noScale-Max* approach avoiding underprovisioning in both datasets, the allocated resources during the experiments were higher than what was needed to meet the demand, and many resources were wasted, as illustrated in Fig. 12. Considering the Madrid and Synthetic datasets, the *Hybrid* and *Reactive* approaches had very similar results regarding provisioning accuracy θ_U and θ_O. However, when considering the amount of SLA (Degraded QoS) violations and scaling operations, *Hybrid* approach outperforms the others.

The Elastic Speedup ϵ metric calculates the geometrical mean of the metrics already discussed, representing the gain of a given approach compared to others. Table 3 shows that our approach outperformed the other auto-scaling approaches since the calculated ϵ values were higher than the others regarding both datasets. Since the calculated τ_U and θ_U are equal to 0 in *noScale-Max* approach, Eq. (4), which defines the ϵ metric, cannot be applied. However, if we consider τ_U and θ_U equal to 1 in *noScale-Max*, ϵ metric can be computed, and the resulting value is 3.05 and 2.62 for Madrid and Synthetic dataset.

Fig. 13 shows how many resources our proposed auto-scaling has allocated during the experiment, considering the two application workloads. It is possible to note that the number of resources always follows the workload fluctuation. The overprovisioning situations occurred when there was an abrupt change in the behavior of the data, known as concept drift. The statistical properties of the target variable, which the predictor model tries to predict, change over time in unforeseen ways. Since the online ML model constantly learns, concept drift effects are minimized as the model's parameters are updated with each new observation. In this way, the model "adapts" to the new data reality.

4.4.2. Analysis of the predictor algorithm

The predictor algorithm is a fundamental part of our auto-scaling approach. We employ an online machine learning predictor based on KNN Regressor. The predictor's performance is expected to improve as new observations arrive and

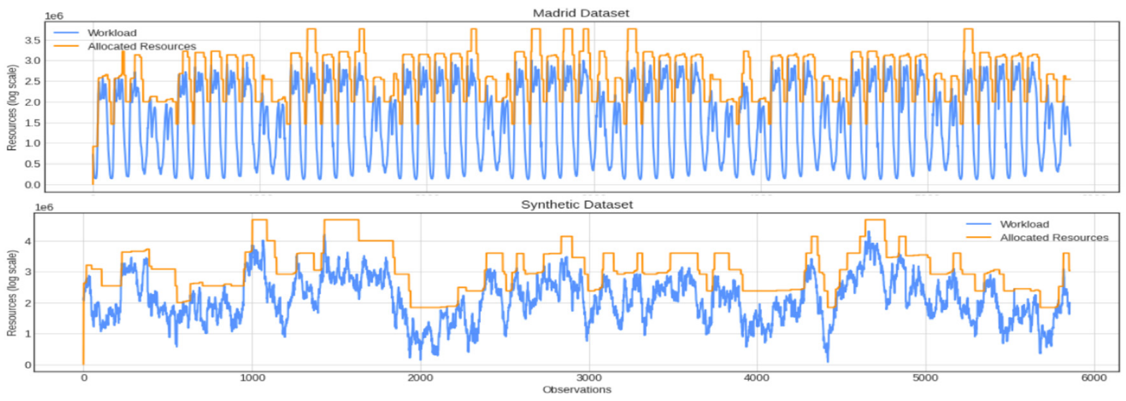


Fig. 13. Allocated resources during the experiments.

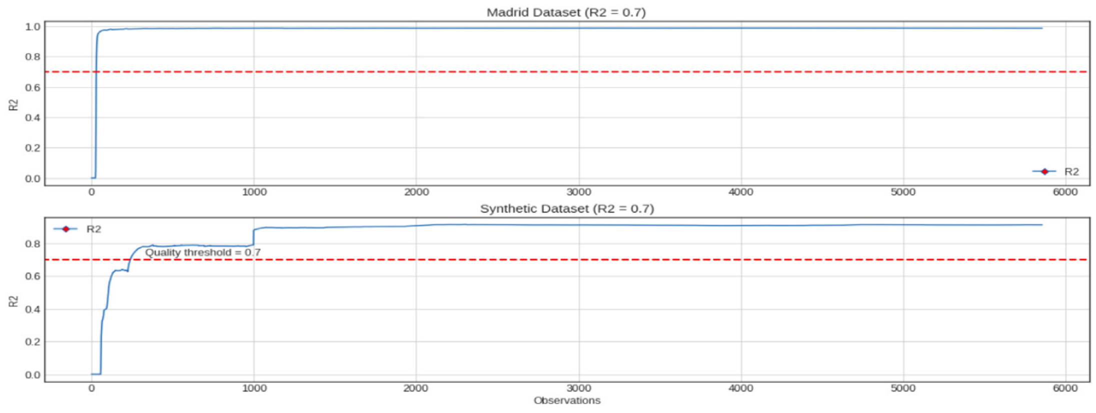


Fig. 14. The improvement of the performance metric during the experiments.

the model is continuously trained. Our approach ensures that auto-scaling will behave reactively if the predictor does not reach a desirable quality threshold. During the experiment, the predictions improved, so the auto-scaling accuracy also improved.

After around 1000 observations for both datasets, we observed that the difference between the predictions and the ground truth decreases following the model performance improvement. The model begins to "understand" the behavior of the data after learning from one-week observations.

Fig. 14 illustrates the improvement of the model performance metrics (R^2) during the experiments. The dashed line defines the quality threshold that was adopted (0.70). It can be seen that the point at which the quality metric crossed the dotted line in Fig. 14 corresponds to the point at which the predictor improved. From this point on (around 1000 observations for the Synthetic dataset and 50 observations for the Madrid dataset), auto-scaling behaves proactively.

Considering the time required by the online machine learning to train the model and make a prediction, the algorithm k-Nearest Neighbors regressor [32], implemented in Python language using the River Framework[6] and performed on a Raspberry Pi 3 (CPU Quad-core ARM Cortex-A7, 32Bit, 1 GB RAM), required on average less than two milliseconds to train the model and ten milliseconds to make a prediction.

The average time required to train the model and make a prediction on a resource-constrained device is a good indication of the viability of our solution in the context of video analytics. The maximum time observed for training was around 133 ms, while the maximum time required for prediction was 168. In addition, the average memory consumption of such a model is less than 10 MB. This way, it is possible to allocate the auto-scaling subsystem on a resource-constrained device on the network's edge.

5. Related work

Auto-scaling is widely used in the Cloud Computing environment to automatically adjust cloud services by increasing or decreasing the number of computational resources to ensure desired performance levels and reduce costs. Many

6 River Framework - https://riverml.xyz/latest/.

research papers in this area have already been published using different techniques [17,38] to ensure elasticity properties to cloud providers. However, such techniques cannot be directly employed in Edge Computing to prevent resource overprovisioning or underprovisioning due mainly to the heterogeneity and dynamic nature of the environment. To the best of our knowledge, this paper is the first to deal with auto-scaling in Edge Computing using a hybrid approach based on Online Machine Learning and MAPE-K. For a fair comparison, we compare our work with other auto-scaling approaches in the context of Edge Computing. The criteria used to select the studies included in the related work were: (i) year of publication, meaning we selected the more recent papers published from 2018 onward, (ii) the similarity to our proposal, and (iii) the context of the work must be focused on the edge environment. Next, we present each related work, and Table 4 presents a side-by-side comparison of the reviewed works in terms of their techniques applied, monitored indicators, scaling strategy, evaluation metrics, and the advantages and weaknesses of each approach.

Bali et al. [39] propose a reactive auto-scaling approach for lightweight virtualization technologies (containers). Such an approach adopts the cluster concept to define grouped edge devices, and it follows the MAPE-K loop to monitor the state of the cluster and the performance of services deployed on its devices. Auto-scaling adjusts the number of service instances (horizontal scaling), which corresponds to the service replicas, in response to workload fluctuations. Although this approach can be used in scenarios where the behavior of the data is not known a priori, reactive approaches tend to perform more scaling actions, which can degrade the system's performance. Unlike our approach, this rule-based approach neither anticipates scaling actions nor mitigates oscillations in workload demand. This approach uses response time, latency, and throughput as performance metrics. The authors simulated an IoT edge environment for experimentation and evaluation using Docker-Swarm. The services deployed as containers receive HTTP requests and record the performance metrics when the response is sent back to the client. Prometheus monitors the performance metrics of the deployed services, while AlertManager and Jenkins are used to handle the different alerts and adapt the services (scaling operations). The authors argue that auto-scaling has been proposed for the IoT domain.

In [26], a proactive auto-scaling architecture for containerized applications based on the MAPE control loop is presented. Such architecture provides horizontal scaling of containers tailored for the Web application domain. Their prediction model is based on Long Short-Term Memory (LSTM), an offline machine learning that requires a dataset for the training phase. However, the complex structure of LSTM models constrains their running speed and performance, so predictions take a few seconds to be performed [40]. Although this architecture mitigates oscillations in workload demand, differently from our approach, it cannot operate in scenarios where there is no previous dataset representing the behavior of the application workload. Furthermore, over time the model may become outdated as the application's workload pattern changes, and the model cannot predict with reasonable accuracy. Besides, the auto-scaling uses resource utilization, HTTP requests, and the number of queries to a database as the monitored indicators. The paper presents a simulated environment using the Worldcup98 dataset to evaluate and compare the proposed LSTM prediction model with Autoregressive Integrated Moving Average (ARIMA) and Artificial Neural Network (ANN) models in terms of prediction accuracy and speed. Besides, the authors proposed an evaluation scenario comprising a distributed database cluster implemented using Docker Swarm, in which replicas are created in response to the fluctuation of HTTP requests (Worldcup98). The manager node runs the auto-scaler container, and HTTP workloads are generated using Apache JMeter. The authors used under/overprovisioning and the elastic speedup [27] as the evaluation metrics to assess auto-scaling.

Tseng et al. [41] present a reactive fuzzy-based auto-scaling approach for fog computing environments using hypervisor and container virtualization technologies tailored to the industrial IoT applications domain. This approach controls the number of service instances (horizontal scaling) by considering CPU, memory, and network performance metrics. A set of thresholds for such metrics are previously defined, and "if-then" rules for fuzzy inference are used to determine the scaling actions to be performed. There is no mechanism to prevent scaling oscillations, and this work does not systematically organize the scaling process (e.g., MAPE-K control loop). Although it can operate in edge computing scenarios where the behavior of the data is not known a priori (no dataset), reactive approaches tend to degrade the system performance over time. The approach was evaluated using an HTTP Server and Apache JMeter as the workload generator (HTTP requests generator). According to the workload fluctuation, VMs and containers were scaled to meet the demand. Meanwhile, the consumption of CPU, memory, and network was measured.

In [42], Wang et al. presents a framework to manage edge nodes (ENORM), which provides mechanisms for provisioning and auto-scaling edge node resources tailored for the online game domain. Their proposed reactive auto-scaling approach uses application latency as the scaling indicator to increase or decrease the resources (vertical scaling) associated with the container running a given application. Although the framework does not systematically organize the scaling process, it mitigates oscillations by running the containers with different priorities. This way, the auto-scaling approach does not progressively scale down containers with lower priority. Instead, the container with the lowest priority is terminated until there are sufficient resources for the container with the highest priority to scale up. However, the auto-scaling only considers static priorities of containers (applications) set by the cloud manager instead of dynamic priorities, leading to QoS degradation when the workload increases rapidly. This approach uses response time as the performance metric. Besides, it was evaluated using an online game scenario (iPokeMon game) running on Amazon EC2 infrastructure, where Apache JMeter was used to stress the partitioned game servers on the edge nodes, and the response time, application latency, communication frequency and data transfer were measured.

In [36], Etemadi et al. propose an auto-scaling framework for resource scaling in the fog tier applying Recurrent Neural Network (RNN). To handle dynamic workloads in the fog environment, the *Resource Manager* (RM) module of the framework decides what to do (scale up/scale down/no action), according to the IoT device input request and the RNN method. The objective function minimizes the total cost (resource usage) and QoS violation (i.e., fog nodes fail to meet predetermined response latency). The metrics used are CPU utilization, delay violation, and network usage. When an IoT device request has a deadline (user QoS latency response) that exceeds a given threshold, this request is sent to the cloud nodes. Otherwise, RM must decide which fog node will process the request according to the "memory" (previous requests) associated with each one. With this aspect, RNN techniques are similar to the Reinforcement Learning methods, another broad area of research applying learning techniques without requiring a training dataset. This approach was evaluated using the simulator iFogSim toolkit and synthetic and real datasets to simulate incoming requests from IoT devices in an IoT domain.

Lee et al. [43] propose an auto-scaling method based on deep Q-networks (DQN), a reinforcement learning algorithm, to resize the number of instances (horizontal scaling) assigned to a service running in the Multi-access edge computing (MEC) environment. Such an approach was implemented as a module running in OpenStack and published as open-source software. In the MEC environment proposed by the author, a service consists of multiple components, which can be deployed independently in VMs or containers. The approach was evaluated using an OpenStack-based testbed with several servers connected through OpenFlow switches. Such a OpenStack environment consists of several nodes that host VMs and a controller node that manages the OpenStack environment. The evaluation employed a Network Service Chaining (SFC) composed of four network functions (VNFS): a firewall, flow monitor, deep packet inspection, and intrusion detection system. According to the authors, a VNF instance and SFC are considered microservice and services in the evaluation. The auto-scaling aims to scale the number of instances running the VNF process.

Etemadi et al. [44] present an autonomous resource provisioning framework based on the generic fog environment three-tier architecture and MAPE-K. The proposed framework is tailored to the IoT domain and implements an auto-scaling approach based on the bayesian learning technique and multiple time series predictors. The framework scales fog node resources according to the predicted workload of IoT services. The auto-scaling does not require any information regarding the environment. The authors performed simulations using the simulator iFogSim, and synthetic and real-world workload traces to validate the framework's effectiveness in reducing the total cost and delay violation compared with the other methods. The simulated fog computing infrastructure includes four fog nodes for deploying IoT services with different characteristics, including processing power, main memory, storage capacity, and bandwidth.

Ray et al. [45] present a horizontal auto-scaling for MEC using the Safe Reinforcement Learning-based auto-scaling policy agent that can efficiently adapt to traffic variations to ensure adherence to service-specific latency requirements. They use the Q-Learning algorithm to learn the rewards associated with the MEC environment. The MEC environment and the auto-scaling decision-making are modeled using the Markov Decision Process (MDP). The authors present experimental results in practical scenarios with real-world benchmark applications to show the effectiveness of the proposed approach. Three different application workloads from DeathStarBench [46], an open-source benchmark suite, were used during the simulations representing the applications scenarios in domains of social media, movie streaming, and object detection. This work does not propose any mechanism to mitigate scaling oscillations.

Considering the evaluation process presented in the related work, we observed that most proposals use simulations of edge environments to assess auto-scaling mechanisms. The main reason reported by the authors for choosing simulations lies in the fact that simulators for Edge and Fog environments have evolved recently. Moreover, in simulations, it is possible to monitor a wide range of metrics and variables of interest (e.g., edge node resources, network and infrastructure, etc.), control and minimize noise effects that may affect evaluations using real scenarios, as well as making experiments more easily reproducible. Two papers [36,44] use the IFogSim simulator to model and simulate a heterogeneous edge environment.

Docker Swarm and Docker were used by [26,39,41] to control and manage clusters of services, simulating a real deployment scenario on virtualized devices. The online game use case (iPokeMon) used by [42] uses the Amazon EC2 infrastructure to host and execute the services that make up the game and assess the auto-scaling. Unlike simulators, scenarios created using Docker Swarm, and real environment or tesbeds (used by [43]) are susceptible to interference from the network, operating system, and physical machine resources, making the results of sampling not repeatable in case the experiments were reproduced. Nevertheless, such simulations are closer representations of a deployment in a real environment, where various aspects can interfere with the auto-scaling behavior.

We realized that most auto-scaling approaches had been proposed for the container environment leveraging many benefits from container technologies such as being lightweight, very quick to launch, consuming minimum resources to run an application which reduces cost, and isolating applications. However, according to [26], designing and implementing general-purpose auto-scaling in virtualized technologies is still challenging despite the benefits of container technologies.

Finally, considering the application domains to which auto-scaling was proposed, we observed that the approaches consider only one domain. This is because auto-scaling solutions must "learn" the application workload behavior in a specific domain in order to scale the resources properly. Our auto-scaling approach was proposed to operate with IoT data-intensive applications, such as intelligent video analytics applications. Therefore, our proposal is domain-specific agnostic, i.e., we can use our solution in different domains that share the same characteristics.

Table 4
A side-by-side comparison of the related work auto-scaling approaches.

Work	Technique	Monitored indicators	Scaling method	Auto-scaling strategy	Evaluation metrics	Evaluation	Cons	Pros
[39]	Rule-based	-Response time	Horizontal	Reactive	-Response Time -Latency -Throughput	Simulated IoT edge environment using Docker-Swarm.	-Neither anticipates scaling actions nor mitigates scaling oscillations.	-It does not require an a priori dataset. -Reducing response time and delay. -Improving bandwidth utilization.
[26]	LSTM	-Resource utilization -HTTP requests rate -Quantity of queries	Horizontal	Proactive	CPU, memory and network usage	Simulated and real environment Docker Swarm.	-It requires a dataset in advanced for the training phase. -Predictions take a few seconds to be performed. -Real time implementation is not supported.	-Mitigate oscillations in workload demand. -Reducing the delay rate.
[41]	Fuzzy Theory	-CPU usage -Memory usage	Horizontal	Reactive	CPU, memory, and network usage	Simulated environment using Docker.	-Neither anticipates scaling actions nor mitigates scaling oscillations. -The system operator must define a complex set of thresholds.	-It does not require an a priori dataset. -Low overhead. -Autonomous management.
[42]	Rule-based	Response time	Vertical	Reactive	Response Time	Online game use case.	-Static priorities of containers (applications) lead to QoS degradation.	-It mitigates scaling oscillations. -It does not require an a priori dataset. -Low overhead. -Reducing service latency.
[36]	RNN + DL	-CPU usage -Network usage -Response time	Vertical	Proactive	-CPU and Network usage -Response time	Simulated environment using iFogSim.	-Low scalability. -It is not suitable to data intensive applications.	-It mitigates scaling oscillations. -It does not require an a priori dataset. -Minimize the resource usage.
[43]	DQN	-CPU usage, -Network usage -Response time	Horizontal	Proactive	Response time	Simulation using a OpenStack-based testbed.	-It only determines scaling actions based on the monitoring metrics observed in instances (VMs). -They did not validate the approach by using VNFs fully designed with microservices.	-Module running in OpenStack. -It mitigates scaling oscillations. -It does not require an a priori dataset.

(*continued on next page*)

6. Conclusions and future directions

This paper presented a MAPE-K-based auto-scaling subsystem that uses an Online Machine Learning (OML) [7] predictor to predict the workload of processing services running at the network edge. It considers scenarios where the

Table 4 (*continued*).

Work	Technique	Monitored indicators	Scaling method	Auto-scaling strategy	Evaluation metrics	Evaluation	Cons	Pros
[44]	Bayesian Technique	-CPU usage. -IoT requests (response time).	Horizontal	Proactive	-CPU Usage -Number of Fog Nodes -Response time Delay Violations	Simulated environment using iFogSim.	-High response time. -It does not mitigate scaling oscillations.	-Low cost. -It does not require an a priori dataset.
[45]	RL	-Service Latency -Quantity of containers	Horizontal	Proactive	-Average latency. -Number of service instances -SLA Violations. -Running time.	Simulated environment (NA)	-It does not mitigate scaling oscillations.	-Low cost. -It does not require an a priori dataset.
Our work	OML	Processing Requests	Horizontal	Hybrid	QoS Degradation	Simulation using a real and a synthetic dataset	-Shortcoming to run the approach using different workload patterns. -Lack of case studies in other scenarios.	-It does not require a dataset a priori. -Lightweight and fast prediction model. -Suitable to data intensive applications. -Different evaluation metrics. -Fully available for use and assessment.

number of processing requests is unknown beforehand, supporting proactive and reactive behavior to dynamically adjusts the number of worker nodes in response to workload changes.

This work advances the state-of-the-art auto-scaling strategies for an edge-based scenario by joining online machine learning and the autonomic computing MAPE-K control loop in a hybrid auto-scaling approach. The auto-scaling subsystem does not need to know the behavior of the application's workload a priori, and it can operate in resource-constrained edge devices (e.g., Raspberry Pi). Our experiments have shown that the prediction model is lightweight, consumes few computational resources, and the time for prediction and training is in the order of milliseconds.

We evaluated the auto-scaling strategy in a scenario of an intelligent video analytics application for smart cities — an Automatic License Plate Recognition application composed of a three-task pipeline. The simulation of the application workloads used a real and a synthetic traffic dataset. The experiments demonstrated the feasibility of our auto-scaling applied to a scenario where proactive auto-scaling based on offline machine learning cannot be applied due to the lack of a complete dataset representing the data's behavior. The proposed auto-scaling performed better than a purely reactive approach based on thresholds. Unfortunately, in the context of auto-scaling in the edge environment, most authors have not made available the source code of the proposed solutions and the datasets used in the evaluations. Thus, it is not easy to compare the solutions that have already been proposed with a new one. In this sense, our evaluation tries to be as comprehensive as possible, comparing our solution with three auto-scaling approaches (noScale-min, noScale-max, and reactive) tailored to the edge computing environment. In particular, the reactive method is quite a representative auto-scaling approach, as the related works [39,41,42] use the reactive method as a scaling strategy.

The central differential of our approach is that when using an online machine learning model, the dataset is not needed to break into test and training sets, as is mandatory in traditional machine learning. The online machine learning model learns from each observation it receives. Therefore, it has great applicability in contexts with little or no prior knowledge of the data's behavior. In this way, it is possible to use the proposed auto-scaling strategy in different scenarios at the network edge.

As future work, we intend to evaluate our auto-scaling subsystem with different application workload patterns and scenarios using real resource-constrained devices to measure various aspects such as CPU and memory usage, packet latency, and network usage. Other adaptation decisions are involved in a system, such as a resource migration. We also plan to propose a migration subsystem complementing the auto-scaling subsystem. Such a migration system will use the same interfaces but deal with other types of adaptation decisions.

We also intend to improve the overall performance of auto-scaling by employing an ensemble solution that combines single prediction models to strengthen the overall prediction, improving the performance and accuracy [19]. The ensemble is a promising method for auto-scaling because, in general, each predictor model presents different behaviors considering the bias and variance when experimented with a different dataset. Therefore, as ensemble method can alleviate the limitations of each model.

Finally, we aim to develop and evaluate the proposed auto-scaling approach in a real application benchmark such as Kubernetes, a container-orchestration system. The Kubernetes Vertical and Horizontal Pod auto-scaling are based on thresholds and cannot predict future workload. Defining such thresholds is a non-trivial task.

CRediT authorship contribution statement

Thiago Pereira da Silva: Conceptualization, Methodology, Software, Resources, Validation, Data Curation, Investigation, Writing - Original Draft. **Aluizio Rocha Neto:** Conceptualization, Methodology, Investigation. **Thais Vasconcelos Batista:** Conceptualization, Funding acquisition, Writing - Review & Editing. **Flávia C. Delicato:** Conceptualization, Funding acquisition, Writing - Review & Editing. **Paulo F. Pires:** Conceptualization, Funding acquisition, Writing - Review & Editing. **Frederico Lopes:** Conceptualization, Writing - Review & Editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] E. Ramos, R. Morabito, J. Kainulainen, Distributing intelligence to the edge and beyond [research frontier], IEEE Comput. Intell. Mag. 14 (4) (2019) 65–92.
[2] A. Rocha Neto, T.P. Silva, T. Batista, F.C. Delicato, P.F. Pires, F. Lopes, Leveraging edge intelligence for video analytics in smart city applications, Information 12 (1) (2021).
[3] A.R. Neto, T.P. da Silva, T.V. Batista, F.C. Delicato, P.F. Pires, F. Lopes, An architecture for distributed video stream processing in IoMT systems, Open J. Internet Things 6 (2020) 89–104.
[4] A. Neto, T. Silva, T. Batista, F. Lopes, F. Delicato, P. Pires, Optimizing resource allocation in edge-distributed stream processing, in: Proceedings of the 17th International Conference on Web Information Systems and Technologies - WEBIST, SciTePress, INSTICC, 2021, pp. 156–166, http://dx.doi.org/10.5220/0010714700003058.
[5] R. OpenFog, OpenFog reference architecture for fog computing, 2017, URL: https://www.openfogconsortium.org/ra/.
[6] S. Taherizadeh, V. Stankovski, Auto-scaling applications in edge computing: Taxonomy and challenges, in: Proc. of the Int. Conf. on Big Data and Internet of Thing, ACM, New York, NY, USA, 2017, pp. 158–163, http://dx.doi.org/10.1145/3175684.3175709.
[7] A. Bifet, R. Gavaldà, G. Holmes, B. Pfahringer, Machine Learning for Data Streams with Practical Examples in MOA, MIT Press, Cambridge, MA, 2018, URL: https://moa.cms.waikato.ac.nz/book-html/.
[8] S.C. Hoi, D. Sahoo, J. Lu, P. Zhao, Online learning: A comprehensive survey, Neurocomputing 459 (2021) 249–289.
[9] P. Arcaini, E. Riccobene, P. Scandurra, Modeling and analyzing MAPE-k feedback loops for self-adaptation, in: 2015 IEEE/ACM 10th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2015, pp. 13–23, http://dx.doi.org/10.1109/SEAMS.2015.10.
[10] T.P. Silva, A.R. Neto, T. Batista, F. Lopes, F.C. Delicato, P.F. Pires, Horizontal auto-scaling in edge computing environment using online machine learning, in: Proc. of the 19th IEEE International Conference on Pervasive Intelligence and Computing (PICom 2021), IEEE Computer Society, Calgary, Alberta, Canada, 2021, pp. 161–168.
[11] A. Razzaq, A systematic review on software architectures for IoT systems and future direction to the adoption of microservices architecture, SN Comput. Sci. 1 (6) (2020) 1–30.
[12] P.M. Mell, T. Grance, SP 800-145. The NIST Definition of Cloud Computing, Technical Report, National Institute of Standards & Technology, Gaithersburg, MD, USA, 2011.
[13] A. Akbar, A. Khan, F. Carrez, K. Moessner, Predictive analytics for complex IoT data streams, IEEE Internet Things J. 4 (5) (2017) 1571–1582.
[14] A. Tsymbal, The Problem of Concept Drift: Definitions and Related Work, Technical Report, Trinity College Dublin, 2004.
[15] Y.S. Patel, D. Verma, R. Misra, Deep learning based resource allocation for auto-scaling VNFs, in: 2019 IEEE Int. Conf. on Advanced Networks and Telecommunications Systems (ANTS), 2019, pp. 1–6, http://dx.doi.org/10.1109/ANTS47819.2019.9118065.
[16] A. Bauer, J. Grohmann, N. Herbst, S. Kounev, On the value of service demand estimation for auto-scaling, in: MMB, Springer, 2018, pp. 142–156.
[17] C. Qu, R. Calheiros, R. Buyya, Auto-scaling web applications in clouds: A taxonomy and survey, ACM Comput. Surv. 51 (2016).
[18] D. Saad, On-Line Learning in Neural Networks, in: Publications of the Newton Institute, Cambridge University Press, 1999, http://dx.doi.org/10.1017/CBO9780511569920.
[19] H.M. Gomes, J. Read, A. Bifet, J.P. Barddal, J. Gama, Machine learning for streaming data: State of the art, challenges, and opportunities, SIGKDD Explor. Newsl. 21 (2) (2019) 6–22.
[20] M. Halford, The correct way to evaluate online machine learning models, 2020, https://maxhalford.github.io/blog/online-learning-evaluation/. Last accessed 19 May 2020.
[21] J. Gama, R. Sebastião, P.P. Rodrigues, On evaluating stream learning algorithms, Mach. Learn. 90 (3) (2013) 317–346.
[22] IBM, An Architectural Blueprint for Autonomic Computing, Technical Report, IBM, 2005.
[23] J. Kephart, D. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.
[24] J. Brownlee, Master Machine Learning Algorithms: Discover how They Work and Implement Them from Scratch, Jason Brownlee, 2016, URL: https://books.google.com.br/books?id=PCJnAQAACAAJ.
[25] S. Putatunda, Practical Machine Learning for Streaming Data with Python, first ed., Apress, Berkeley, CA, 2021, p. 136, http://dx.doi.org/10.1007/978-1-4842-6867-4.

[26] M. Imdoukh, I. Ahmad, M. Alfailakawi, Machine learning-based auto-scaling for containerized applications, Neural Comput. Appl. 32 (2019) 9745–9760.

[27] S. Rahman, T. Ahmed, M. Huynh, M. Tornatore, B. Mukherjee, Auto-scaling VNFs using machine learning to improve QoS and reduce cost, in: 2018 IEEE International Conference on Communications (ICC), 2018, pp. 1–6, http://dx.doi.org/10.1109/ICC.2018.8422788.

[28] P. Singh, P. Gupta, K. Jyoti, A. Nayyar, Research on auto-scaling of web applications in cloud: Survey, trends and future directions, Scalable Comput. Pract. Exp. 20 (2019) 399–432.

[29] S. Bodiwala, N. Nanavati, Efficient hardware implementations of deep neural networks: A survey, in: 2020 Fourth International Conference on Inventive Systems and Control (ICISC), 2020, pp. 31–36, http://dx.doi.org/10.1109/ICISC47916.2020.9171171.

[30] Y. Han, X. Wang, V.C.M. Leung, D.T. Niyato, X. Yan, X. Chen, Convergence of edge computing and deep learning: A comprehensive survey, IEEE Commun. Surv. Tutor. 22 (2020) 869–904.

[31] Y. Zhu, M. Huang, F. Chen, Creating a real-time license plate detection and recognition app, 2021, https://developer.nvidia.com/blog/creating-a-real-time-license-plate-detection-and-recognition-app/. Last accessed 20 Dec 2021.

[32] N.S. Altman, An introduction to kernel and nearest-neighbor nonparametric regression, Amer. Statist. 46 (3) (1992) 175–185.

[33] H.A. Abu Alfeilat, A.B.A. Hassanat, O. Lasassmeh, A.S. Tarawneh, M.B. Alhasanat, H.S. Eyal Salman, V.B.S. Prasath, Effects of distance measure choice on K-nearest neighbor classifier performance: A review, Big Data 7 (4) (2019) 221—248.

[34] R.I. Rasel, N. Sultana, P. Meesad, An efficient modelling approach for forecasting financial time series data using support vector regression and windowing operators, Int. J. Comput. Intell. Stud. 4 (2015) 134.

[35] N. Wagner, Z. Michalewicz, An analysis of adaptive windowing for time series forecasting in dynamic environments: Further tests of the DyFor GP model, in: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, 2008, pp. 1657–1664.

[36] M. Etemadi, M. Ghobaei-Arani, A. Shahidinejad, A cost-efficient auto-scaling mechanism for IoT applications in fog computing environment: a deep learning-based approach, Cluster Comput. 24 (4) (2021).

[37] L. Ju, P. Singh, S. Toor, Proactive autoscaling for edge computing systems with kubernetes, 2021, CoRR abs/2112.10127.

[38] T. Chen, R. Bahsoon, X. Yao, A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems, ACM Comput. Surv. 51 (3) (2018).

[39] A. Bali, M. Al-Osta, S.B. Dahsen, A. Gherbi, Rule based auto-scalability of IoT services for efficient edge device resource utilization, J. Ambient Intell. Humaniz. Comput. (2020) 1–18.

[40] Y. Zhang, X. Hao, Y. Liu, Simplifying long short-term memory for fast training and time series prediction, J. Phys. Conf. Ser. 1213 (2019) 042039.

[41] F.-H. Tseng, M.-S. Tsai, C.-W. Tseng, Y.-T. Yang, C.-C. Liu, L.-D. Chou, A lightweight autoscaling mechanism for fog computing in industrial applications, IEEE Trans. Ind. Inform. 14 (10) (2018) 4529–4537.

[42] N. Wang, B. Varghese, M. Matthaiou, D.S. Nikolopoulos, ENORM: A framework for edge node ResourceManagement, IEEE Trans. Serv. Comput. 13 (6) (2020) 1086–1099.

[43] D.-Y. Lee, S.-Y. Jeong, K.-C. Ko, J.-H. Yoo, J.W.-K. Hong, Deep Q-network-based auto scaling for service in a multi-access edge computing environment, Int. J. Netw. Manag. 31 (6) (2021).

[44] M. Etemadi, M. Ghobaei-Arani, A. Shahidinejad, Resource provisioning for IoT services in the fog computing environment: An autonomic approach, Comput. Commun. 161 (2020) 109–131.

[45] K. Ray, A. Banerjee, Horizontal auto-scaling for multi-access edge computing using safe reinforcement learning, ACM Trans. Embed. Comput. Syst. 20 (6) (2021).

[46] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, C. Delimitrou, An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, Association for Computing Machinery, New York, NY, USA, 2019 pp. 3–18, http://dx.doi.org/10.1145/3297858.3304013.

Thiago Pereira has been an assistant professor at the Federal University of Mato Grosso (UFMT) since 2013. He is a Ph.D. candidate in Computer Science at the Federal University of Rio Grande do Norte (Natal, Brazil). He holds a Master's degree in Systems and Computing from the Federal University of Rio Grande do Norte (2012) and a BS in Computer Science from the Federal University of Goiás. He has experience in Computer Science, programming, and distributed systems, working mainly on the following topics: Fog Computing, Middleware, Cloud Computing, Ubiquitous Computing, and the Internet of Things.

Aluizio Rocha is an assistant professor at the Instituto Metrópole Digital of the Federal University of Rio Grande do Norte (UFRN). He is a BS in Computer Science from UFRN (1994) and a Master's degree in Computer Science from UNICAMP (1997). He obtained his doctoral degree (2021) in Computer Science in the Department of Informatics and Applied Mathematics (DIMAp) at UFRN. He was a professor at the Centro Universitário do Rio Grande do Norte (UNI-RN) from 1999 to 2015. Currently, he develops research on the Internet of Things, Edge Computing, Computer Vision, Machine Learning and Deep Learning.

Thais Batista is a professor at the Federal University of Rio Grande do Norte (UFRN), which she joined in 1996. She has held the Graduate Program in Computer Systems and Computing (PPgSC) coordinator in the period 2006–2010. She holds a Bachelor's degree in Computer Science Federal University of Paraíba (1990), a Master's degree in Computer Science from Pontifícia Universidade Católica do Rio de Janeiro — PUC-Rio (1994), and a Ph.D. in Computer Science from Catholic University of Rio de Janeiro — PUC-Rio (2000). He did post-doctoral studies at the University of Lancaster — UK (2004–2005 and 2013–2014). He has experience in Computer Science, with emphasis on Software Architecture and Distributed Systems, working mainly on the following Cloud Computing, Middleware, the Internet of Things, and Architectural Description Language, among others. Member of the FIWARE Foundation since 2017 and FIWARE Evangelist since 2018. He is a CNPq level 1D productivity fellow.

Frederico Lopes is a Professor at the Federal University of Rio Grande do Norte since 2012. He has a degree in Computer Science from the Federal University of Rio Grande do Norte (2005). He holds a master's degree in Systems and Computing from the Federal University of Rio Grande do Norte (2008) and Ph.D. in Computer Science Computing Science from UFRN. Postdoctoral fellow (2016) at the University of British Columbia, Vancouver/Canada. In 2010, he was on a Ph.D. student internship (sandwich doctorate) at IST/UTL, Lisbon/Portugal. IST/UTL, Lisbon/Portugal. He has experience in Computer Science, with emphasis on Distributed Systems and Software Engineering, working mainly on the following topics: middleware, ubiquitous computing, pervasive computing, smart cities, publish/subscribe middleware, cloud computing, and context-aware. Member of the Fiware Foundation since 2017.

Flávia C. Delicat holds a master's degree in Computer Science (2000) and Ph.D. in Electrical Engineering (2005) from the Federal University of Rio de Janeiro and postdoctoral studies (2010 and 2016) at the University of Sydney, Australia. He has experience in Computer Science, with emphasis on Distributed Systems and Software Engineering, working mainly on the following topics: Internet of Things, wireless sensor networks middleware, adaptive systems, Fog/Edge Computing, and model-driven development. She is an associate professor at Fluminense Federal University, integrates the Centre for Distributed and High-Performance Computing at the University of Sydney, and since 2017 has been a member of the IEEE Technical Committee on Smart World (Technical Committee on Smart World — SWTC). He is a CNPq productivity level 1D of CNPq.

Paulo F. Pires is an associate professor of the Computer Institute at Universidade Federal Fluminense (UFF) and an external member of the Centre for Distributed and High-Performance Computing at the University of Sydney. He was a visiting researcher at the CLIP laboratory in Maryland (USA) (2000), at the University of Malaga, Spain (2009) (2009), and the University of Sydney, Australia (2010 and 2016). His main research interests lie in Software Engineering and Distributed Systems intersection. In recent years, his research efforts have focused on the application of Software Engineering techniques in the context of developing systems for the Internet of Things (IoT). Dr. Pires is co-author of two books on the subject of IoT: "Middleware Solutions for the Internet of Things" (Springer, 2013) and "Resource Management for the Internet of Things" (Springer, 2017). He has coordinated several cooperative industry-university and research projects in these areas with industry and several international and Brazilian governmental agencies. He has published over 200 articles in internationally renowned journals, conference papers, book chapters, and book chapters and holds three patents registered with the USPTO. Dr. Pires has served on program committees for several international conferences. He is currently a member of the IEEE TC on Cybermatics and the editorial boards of the Scalable Computing and Communications Journal (Springer) and International Journal of Computer Networks (CSC Journals). His recent awards include the "Comandante Vital de Oliveira" (2017) and (co-author) best paper award, IEEE NCA 2016. He has been a CNPq level 2 research productivity fellow since 2010 and is a member of the IEEE and the Brazilian Society of Computing (SBC).