Future Generation Computer Systems 150 (2024) 90–102

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Proactive resource management for cloud of services environments

Gonçalo Marques [a], Carlos Senna [a,*], Susana Sargento [a,b], Luís Carvalho [c], Luís Pereira [c], Ricardo Matos [c]

[a] Instituto de Telecomunicações, Campus Universitário de Santiago, Aveiro, 3810-193, Portugal
[b] DETI, University of Aveiro, Campus Universitário de Santiago, Aveiro, 3810-193, Portugal
[c] Veniam, Porto, 4000-098, Portugal

ARTICLE INFO

Article history:
Received 13 January 2023
Received in revised form 16 May 2023
Accepted 4 August 2023
Available online 22 August 2023

Keywords:
Cloud service management
Cloud monitoring
Service monitoring
Proactive management
Autonomic management
Machine learning

ABSTRACT

Microservices offer advantages such as better fault isolation, smaller and faster deployments, scalability, and speeding up the development of new applications through the composition of services. However, its large-scale use and specific requirements increase the challenges of monitoring and management. To meet these challenges, we propose a monitoring and management system for microservices, containers and container clusters that autonomously predicts load variations and resource scarcity, which is capable of making new resources available in order to ensure the continuity of the process without interruptions. Our solution's architecture allows for customizable service-specific metrics that are used by the load predictor to anticipate resource consumption peaks and proactively allocate them. In addition, our management system, by identifying/predicting low demand, frees up resources making the system more efficient. We evaluated our service management solution in the AWS environment, environment characterized by high mobility, dynamic topologies caused by disconnection, dropped packets and delay issues. Our results show that our solution improves the efficiency of escalation policies, and reduces response time by improving the QoS/QoE of the system.

1. Introduction

Lightweight services are purpose-built to perform a cohesive or objective business function. It is an evolution of the traditional style of service-oriented architecture, which is already being used by application developers and service providers such as Amazon, Netflix or eBay [1].

The growing adoption of microservices in the cloud is motivated by the easiness of deploying and updating the software, as well as the provisioned loose coupling provided by dynamic service discovery and binding. Structuring the software to be deployed in the cloud as a collection of microservices allows cloud service providers to offer higher scalability through more efficient consumption of resources, and quickly restructure software to accommodate growing demand [2]. Moreover, microservice applications increase the number and diversity of infrastructure resources that must be continuously monitored and managed at runtime. Finally, services can be deployed across multiple regions and availability zones, which adds to the challenge of gathering up-to-date information [3].

Within the scope of smart cities, the advantages of adopting services and their compositions are notorious, as they allow agile development to meet the enormous demand. In this context, Vehicular ad hoc Networks (VANETs) play an important role as a communication infrastructure, since people spend much of their time inside vehicles, especially in big cities. For these reasons, we chose a VANET and its applications to evaluate the performance of our service monitoring and management solution. VANETs are based on a set of communication systems that provide vehicles with access to many different applications. The pervasive roadside infrastructure offers high-speed internet access by advanced wireless communication technologies (3/4/5 G, ITS-5G or C-V2X), and brings innovative and divergent benefits, but also some challenges [4], such as instability of resources which make the management of vehicular networks complicated.

Due to the high mobility and dynamic topologies of VANETs, monitoring and managing microservices becomes an extremely complex issue that can incur high costs unless the services are managed properly. This requires a monitoring system capable of efficiently managing services to optimize overall data, ensure delivery of resources to customers, and detect resource shortages or overspends to keep costs to a minimum. This is usually achieved with monitoring solutions that collect metrics from each observed resource, which are later analyzed by applications to detect problems in the system and ensure that there is no over/underprovisioning of resources [5].

Monitoring is an essential tool for resource management. In cloud environments, resource management is very complex, as

* Corresponding author.
E-mail address: cr.senna@av.it.pt (C. Senna).

the large scale of the infrastructure makes it difficult to obtain accurate information about the global state of the system, and the large population of interacting users makes it very difficult to effectively predict the type and intensity of workloads in the system [6]. Although there are many studies on cloud resource management [7,8], there are not many studies on microservices management, despite the gradual trend of applications moving towards this paradigm. The dynamic lifecycle of microservices puts a burden on managing cloud resources. Furthermore, containerization (lifecycle, scaling, autoscaling, migration, etc.) requires more comprehensive management strategies [9].

To address these challenges, we propose an architecture for accurate monitoring of custom metrics for service-based cloud applications, which will be used for automatic provisioning and scaling of compute resources in cloud environments, while addressing services in very challenging scenarios such as VANETs. Our predictive approach, designed for the AWS/EKS environment, is capable of efficiently provisioning disparate services to users in VANETs. We implement several services with different requirements to evaluate the performance and versatility of our monitoring system. In our evaluation, we analyze data collected on a real vehicular network that was used to train our machine learning algorithm and produce predictions that are sent to the resource provisioning system.

The proposed architecture led to the following contributions:

- design and implementation of a generic monitoring architecture able to collect metric data from various services with an open source framework, enabling horizontal scaling;
- analysis and implementation of a network prediction model using statistical and machine learning techniques;
- analysis of the most common services provided and their different requirements;
- design and implementation of a cloud management system that proactively provides resources needed services in a dynamic environment.

The remainder of this article is organized as follows. Section 2 presents the related work about monitoring systems and service provisioning. Section 3 describes our service monitoring system with proactive resource management. Section 4 discusses the tests made to validate the proposed monitoring system. Finally, Section 5 presents the conclusion and directions for future work.

2. Basic concepts and related work

Regarding the proposed architecture, there are two relevant aspects: monitoring tools & frameworks for clouds and proactive cloud management.

There are specific tools for monitoring resources, applications and services for the cloud. Prometheus[1] is an open-source system monitoring and alerting toolkit that collects and stores its metrics as time series data alongside optional key–value pairs called labels. Prometheus extracts metrics, either directly or through a gateway, storing them locally. It also offers the option to run rules for aggregation and generate alerts. Stored metrics can be viewed through dashboards like Graphana or consumed through Application Programming Interfaces (APIs). Jaeger[2] is a open source distributed tracing system used for monitoring and troubleshooting microservices-based distributed systems. The Jaeger project is primarily a tracing backend that receives tracing telemetry data and provides processing, aggregation, data mining, and visualizations of that data. Open Telemetry[3] resulted from a merger between the OpenTracing and OpenCensus projects; it provides a unified set of instrumentation libraries and specifications for observability telemetry. The project is a set of APIs, a software development kit, tooling and integrations designed for the creation and management of telemetry data such as traces, metrics, and logs. The project provides a vendor-agnostic implementation that can be configured to send telemetry data to any backend. It supports a variety of popular open-source projects including Jaeger and Prometheus. In the testbed that we set up to evaluate our solution, we chose to use the Prometheus monitoring system, as it is the most complete solution, with ease of customization and the ability to produce metrics without affecting the application's performance.

Cloud resource management is a traditional topic with a large scope of related work [10]. Big cloud providers, such as Google, Microsoft, Amazon between others, have monitoring and auto scaling options. AWS offers the EC2 Auto Scaling (EC2-AS) service.[4] EC2-AS works very well for automatically managing cloud resources. To use services published in containers, the recommendation is to use Amazon Elastic Container Service (ECS)[5] but which has restrictions in relation to EC2-AS, and the best option in this case is AWS Fargate.[6] Fargate is a serverless compute engine for containers that works with Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS). However, Fargate aims to optimize the cost of consuming cloud resources while our solution aims to proactively optimize resource consumption while maintaining the performance of application services. Google Kubernetes Engine (GKE)[7] Cluster Autoscaler manages the number of nodes in a node pool, based on workload demand. If the demand is low, the cluster autoscaler scales down to a minimum size that is designated to control costs. Google also has an option, the Autopilot, an autoscaler used to improve the accuracy of threshold settings and optionally offers a sophisticated algorithm inspired by reinforcement learning that has allowed significant additional gains [11]. However, it is more focused on jobs than on services. The Microsoft Azure Kubernetes Service (AKS)[8] cluster autoscaler periodically checks the Metrics API server to determine if the number of nodes in the AKS cluster will increase or decrease accordingly to the usage of the PODs. However, both autoscalers do not mention proactive reaction when a new POD may be required.

Next-generation scaling approaches attempt to move beyond the limitations of such reactive systems, by instead attempting to predict the near-future workloads [12,13]. In such systems, the forecasting component can be considered the most important element, and the differentiating factor when comparing elastic systems [14]. Simic et al. [15] proposed an intelligent decision support mechanism based on artificial neural networks and metaheuristics that provides metrics of optimization duration and cost of resource consumption to support decision making between faster delivery or lower infrastructure costs. It is an interesting solution but it does not meet the specific requirements of service-oriented applications. In service oriented clouds, the forecasting problem is more complex since it involves the collection, processing and analysis of different data sets, which can be traces of resource usage, such as CPU, memory, network bandwidth or metrics related to provided applications/services, such as the number of requests that are being served and application architecture. Lee et al. [16] proposed a cloud-native workload

[1] https://prometheus.io/
[2] https://www.jaegertracing.io/
[3] https://opentelemetry.io/

[4] https://aws.amazon.com/ec2/autoscaling/
[5] https://aws.amazon.com/ecs/
[6] https://aws.amazon.com/fargate/
[7] https://cloud.google.com/kubernetes-engine
[8] https://azure.microsoft.com/en-us/products/kubernetes-service

profiling system with Kubernetesorchestrated multi-cluster configuration. The solution monitors the resource usage when deploying service cases in multiple clouds and identifies the correlation between services and resource usage. Srirama et al. [17] proposed a new container-aware application scheduling strategy with an autoscaling policy that deploys requested applications in lightweight containers for minimal deployment time. However, both works do not provide a predictive and proactive resource management.

Wang et al. [18] proposed a self-adapting resource management framework for cloud software services. For a given service, an iterative QoS model is trained based on historical data, which is capable of predicting a QoS value of a management operation using the information about the current running workload, allocated resources, actual value of QoS in a resource allocation operation. A runtime decision algorithm based on Particle Swarm Optimization (PSO) together with the predicted QoS value are employed to determine future resource allocation operations.

Chen et al. [19] proposed a microservice-based deployment problem based on the heterogeneous and dynamic characteristics in the edge-cloud hybrid environment, including heterogeneity of edge server capacities, dynamic geographical information of IoT devices, and changing device preference for applications and complex application structures. Their algorithm leverages reinforcement learning and neural networks to learn a deployment strategy without any human instruction. This model is to minimize the waiting time of the Internet of Things (IoT) devices, while our proposed model focuses on minimizing the delay of scaling operations by predictively allocating resources to cloud services based on the incoming workload.

Zhao et al. [20] introduced a predictive auto-scaler for Kubernetes clusters to improve the efficiency of autoscaling containers. They proposed a monitoring module that obtains the indicator data of the pod which is used by the prediction module to estimate the number of pods to the auto-scaler module for scaling services. While the authors choose to use the original responsive strategy of Horizontal Pod Autoscaler (HPA), we perform scale down operations based on the values calculated by the predictor, reducing the over provisioning of resources.

Zhou et al. [21] proposed an Ensemble Forecasting Approach for highly-dynamic cloud workload based on Variational Mode Decomposition (VMD) and R-Transformer. To decrease the non-stationarity and high randomness of highly-dynamic cloud workload sequences, workloads are decomposed into multiple Intrinsic Mode Functionss (IMFs) by VMD. The IMFs are then imported into the ensemble forecasting module, based on R-Transformer and Autoregressive models, in order to capture long-term dependencies and local non-linear relationship of workload sequences. However, the authors do not present solutions on how to use a non-decomposition to reduce the instability and randomness of highly-dynamic workload data, which would be advantageous for reducing the training cost of deep neural networks.

Liu et al. [22] presented a workload migration model to minimize migration times and improve the node processing performance. To forecast the workload more accurately, a model that combines Autoregressive Moving Average (ARMA) with the Elman Neural Network (ENN) are proposed. In order to meet cost constraints and deadline constraints, an elastic resource management model based on cost and deadline constraints is proposed. Both the cost and the deadline of tasks are considered. Workload migration can be used to balance the workload of each resource node, making the cluster's response time faster. However, combining on-demand resource provision method with mobile edge computing is worth studying, since in a smart city environment, different areas require different resources depending on the amount of data processing, so the on-demand resource provision can reduce service expenditure while meeting user needs.

Zhu et al. [23] proposed a model using the Long Short-term Memory (LSTM) encoder–decoder network with attention mechanism to improve the workload prediction accuracy. The model uses an encoder to map the historical workload sequence to a fixed-length vector according to the weight of each time step supported by the attention module, and uses a decoder to map the context vectors back to a sequence. Finally, the output layer transforms the sequence into the final output. Moreover, the authors propose a scroll prediction method to reduce the error occurring during long-term prediction, which splits a long-term prediction task into several small tasks.

Kubernetes[9] provides an excellent platform for hosting services and their compositions, offering facilities for managing dynamic workloads, making available and leveraging parallel and distributed architectures. With the growing trend of migration from monolithic applications to the service-based paradigm, Kubernetes is gaining importance. While Kubernetes supports automatic scaling of pods, neither automated scaling of the cluster itself nor proactive actions are available in the current release. Furthermore, there are few related works that explore this gap [24]. Khaleq et al. [25] proposed a generic autoscaling algorithm to identify resource demand of microservices in cloud applications that works in conjunction with an intelligent autoscaling module that identifies threshold values for autoscaling microservices based on resource demand and QoS constraint. The work in [26] proposed the Microscaler that identifies the scaling-needed services and scale them to meet the Service Level Agreement (SLA) with an optimal cost for microservice applications. Microscaler determines under-provisioning or over-provisioning service instances, and combines an online learning with a step-by-step heuristic approach to reach the optimal service scale meeting the SLA requirements. Despite improving the accuracy of the limits, the works in [25,26] do not act proactively in relation to the management of the PODs.

The presented works mostly focus on developing a predictor which is optimized for their specific use case. However, in environments with multiple distinct service applications, a single prediction algorithm might not present the best results for all applications. In our solution, we enable the integration and usage of different predictors for each service. This is a very useful feature, because it allows the usage of the optimal predictors for each case instead of a generic predictor, which may not have the best performance for some applications. Moreover, with the rapid advances in this field, the ability to easily integrate new predictors as they appear is another important characteristic of the system.

3. Proactive cloud resource management

The overall process of proactively managing services in cloud environments includes monitoring resource/service work load metrics, analyzing monitored data and, considering what may occur in the immediate future (proactively), deciding on suitable sizing methods [27]. It is essential that a monitoring system is able to accurately monitor cloud resources (hardware and software) and make automated decisions. There are several options that monitor hardware resources, others that monitor the software involved, whether monolithic applications, services or service compositions. However, most of them react linearly to the increase in demand, which can cause the loss of QoS, mainly due to the time required for new resources to be ready for use.

9 https://kubernetes.io/

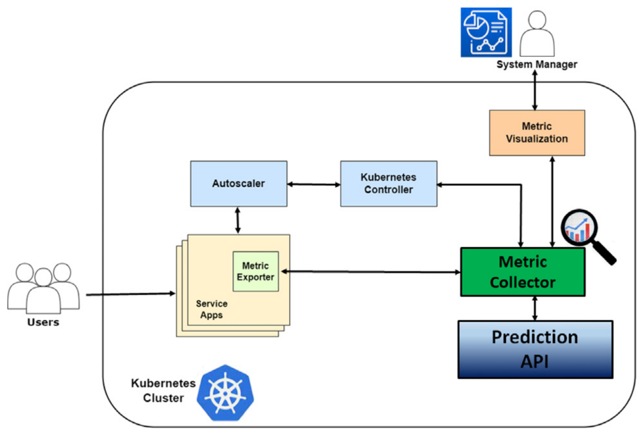


Fig. 1. Main components of the proposed conceptual monitoring architecture.

To ensure that the monitoring system is able to allocate resources while maintaining QoS, we propose an architecture that allows managers to easily define which custom metrics they want to monitor for each service or service composition using custom metrics exporters. In addition, our solution has a forecasting service that identifies consumption peaks (up and down) and proactively makes scale adjustments, in order to guarantee QoS (up), and at the same time optimize resource consumption of the cloud, lowering the operational cost (down).

As shown in Fig. 1, our conceptual architecture exposes the desired metrics to the Metric Collector that can be integrated with the Autoscaler, to make scaling decisions based on metrics other than Central Process Unit (CPU), memory and communication.

The Metric Exporters (ME) collect the metrics from the services and expose them to the Metrics Collector (MC) which will aggregate the data from all services. The Prediction API (PAPI) queries the MC to obtain historical data from the services to update the prediction models, and then exposes its updated predictions as metrics in an HTTP endpoint that is periodically scraped by the MC. The MC gathers data from the services themselves and the PAPI, and exposes the metrics in two ways. First, MC sends metrics to the System Manager (dashboard), allowing the manager to observe and analyze the behavior of the services. Second, MC sends metrics to the Kubernetes Controller which will use the metrics to detect service failures and automatically adjust the resources.

Using our conceptual architecture as a basis, we designed our cloud services monitoring solution, the details of which are presented below. First, we describe the main components related to the monitoring itself, and then we present the proactive version where we detail our predictor service.

3.1. Full metrics monitoring pipeline

In Kubernetes, application monitoring does not depend on a single monitoring solution. The resource metrics pipeline provides a limited set of metrics related to cluster components such as the HPA controller. Fig. 2 shows a high level view of the updated monitoring system architecture with support for full metrics. Its main components are: *Service Apps*, *Metrics Server*, *Prometheus Monitor*, *Prometheus Kubernetes Adapter*, the *Kubernetes Metrics API* and the *HPA*.

To monitor additional metrics, a custom exporter is integrated in each service to publish metrics such as throughput and number of client sessions, streams or file transfers, depending on the service. Moreover, in any monitoring system, it is extremely important to observe the collected data, in order to understand the applications resource requirements. However, the metrics API does not allow the visualization of the historical data. To do that, we chose Prometheus, an open source monitoring and alerting platform. Prometheus collects data from the sources in a time-series format at a set interval, and stores it locally on a disk identified by the metric name and key/value pairs. Details about the main components of our architecture are presented below.

3.1.1. Service apps

The *Service Apps* represent the generic services or compositions of services, that will be monitored and scaled according to the memory and CPU usage values gathered by the *Metrics Server*. The applications are exposed internally through a Kubernetes service, an abstraction which defines a logical set of Pods and a policy by which to access them; they are accessible to clients through a Network Load Balancer (NLB). The NLB selects a target (targets may be EC2 instances, containers or IP addresses) using a flow hash algorithm based on the protocol, source IP address, source port, destination IP address, destination port, and Transmission Control Protocol (TCP) sequence number. The TCP connections from a client have different source ports and sequence numbers, and can be routed to different targets. Each individual TCP connection is routed to a single target for the life of the connection. The *Service Apps* have been integrated with a custom exporter that enables the collection of additional metrics, the most relevant ones being downstream throughput and number of active user sessions. Although some applications may support Prometheus metrics out of the box, for the most part of them we need to develop a custom adapter (CA). The CA helps the application to monitor additional metrics other then memory and CPU usage, collected by the Kubernetes metrics-server by default. Moreover, we need an additional Exporter for the Prometheus server, which collects the values from the Prometheus monitoring server and publishes them in the Kubernetes metrics API where they become available to Autoscaler. Our custom exporters are GO servers that collect information from the applications at run time, and exposes them in an HTTP endpoint according to the Prometheus format, so they can be periodically scraped by the Prometheus Monitor.

3.1.2. Metrics server

The *Metrics Server* is a cluster-wide aggregator of resource usage data. Fig. 3 shows in detail the simultaneous implementation of the custom/external metrics pipeline and the resource metrics pipeline.

The Kubernetes Metrics Server collects and exposes metrics from applications. It discovers all nodes on the cluster, and queries each node's *kubelet* for CPU and memory usage. CPU is reported as the average usage, in CPU cores, over a period of time derived by taking a rate over a cumulative CPU counter provided by the kernel. Memory usage is reported as the working set, in bytes, at the instant the metric was collected. In an ideal world, the "working set" is the amount of memory in-use that cannot be freed under memory pressure. However, the calculation of the working set varies by host Operating System (OS), and generally makes heavy use of heuristics to produce an estimate. It includes all anonymous (non-file-backed) memory, since Kubernetes do not support the swap process. The metric typically includes also some cached (file-backed) memory, because the host OS cannot always reclaim such pages.

The monitoring pipeline fetches metrics from the *Kubelet* which is the primary agent running in every node that works in terms of *PodSpecs*, which are YAML or JSON objects that describe the containers running in that node. The *Kubelet* takes the *PodSpecs* provided by the API Server, and ensures that the containers described in them are running and healthy. The *Kubelet*

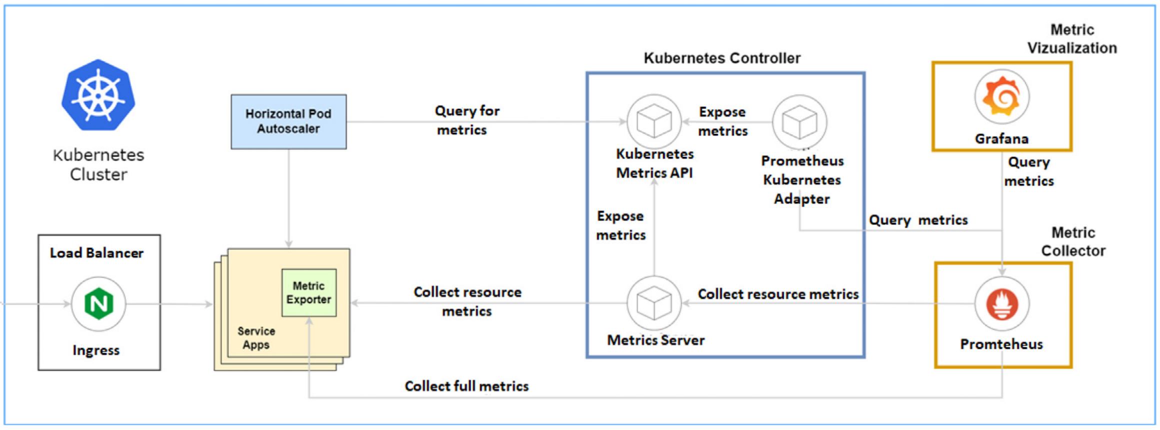


Fig. 2. High-level view of the custom/external metrics monitoring architecture.

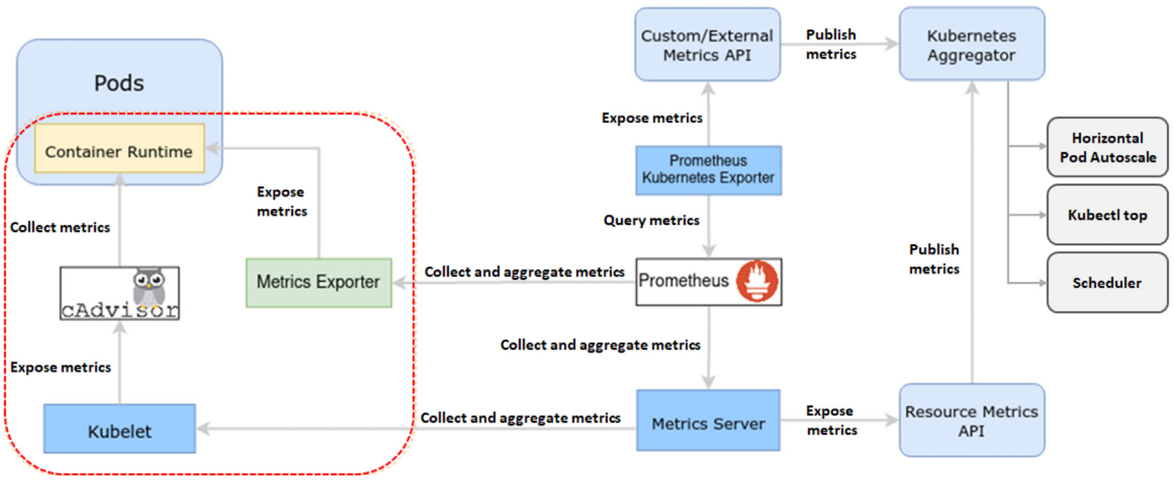


Fig. 3. Detailed view of the custom/external metrics monitoring architecture.

acts as a bridge between the Kubernetes master and the nodes, managing the pods and containers running on a machine. The *Kubelet* translates each pod into its constituent containers, and fetches individual container usage statistics from the container runtime through the container runtime interface. The *Kubelet* fetches this information from the integrated *cAdvisor*, a running *daemon* that collects real-time monitoring data from the containers for the legacy Docker integration. It then exposes the aggregated pod resource usage statistics through the Metrics Server API.

To collect custom metrics from deployed apps, we need to integrate a Metrics Exporter built specifically for each app. This exporter collects the desired metrics and exposes them to an Hypertext Transfer Protocol (HTTP) endpoint that is periodically scraped by Prometheus. The Prometheus monitor collects metrics directly from the application exporters, as well as the Metrics Server. The Prometheus Kubernetes Exporter queries the Prometheus monitor to collect specific metrics and publish them in the Custom/External Metrics API (CEMAPI). Finally, metrics can be used by the HPA to make scaling decisions.

3.1.3. *Prometheus Monitor*

The *Prometheus Monitor* queries the data sources (Metrics Exporters) at a specific polling frequency, at which time the exporters present the values of the metrics at the endpoint queried by Prometheus. The exporters are automatically discovered by creating a Kubernetes entity called *Service Monitor*, a custom Kubernetes resource that declaratively specifies how groups of services should be monitored and which endpoints contain the metrics. The Prometheus aggregates the data from the exporters into a local on-disk time series database that stores the data in a highly efficient format. Data is stored with its specific name and an arbitrary number of key=value pairs (Labels). Labels can include information on the data source and other application-specific breakdown information, such as the HTTP status code, query method (GET versus POST), endpoint, etc.

Supported by basic and custom metrics implemented, the *HPA* automatically scales the number of Pods in a replication controller, deployment, replica set or stateful set based on application-provided metrics instead of only CPU and memory usage metrics. Moreover, with custom metrics, we can detect malfunctions and resource shortages that would go unnoticed if we were just monitoring the CPU application and memory usage, making the metrics pipeline complete, a valuable addition and a great improvement to the tracking system. From the most basic perspective, the HPA controller operates on the ratio between desired metric value and current metric value, as formalized in Eq. (1) below:

$$\mathbf{desiredReplicas} = ceil[currentReplicas* (currentMetricValue/desiredMetricValue)] \tag{1}$$

For example, if the current metric value is 200 m, and the desired value is 100 m, the number of replicas will be doubled, since $200.0/100.0 = 2.0$. If the current value is instead 50 m, we will halve the number of replicas, since $50.0/100.0 = 0.5$.

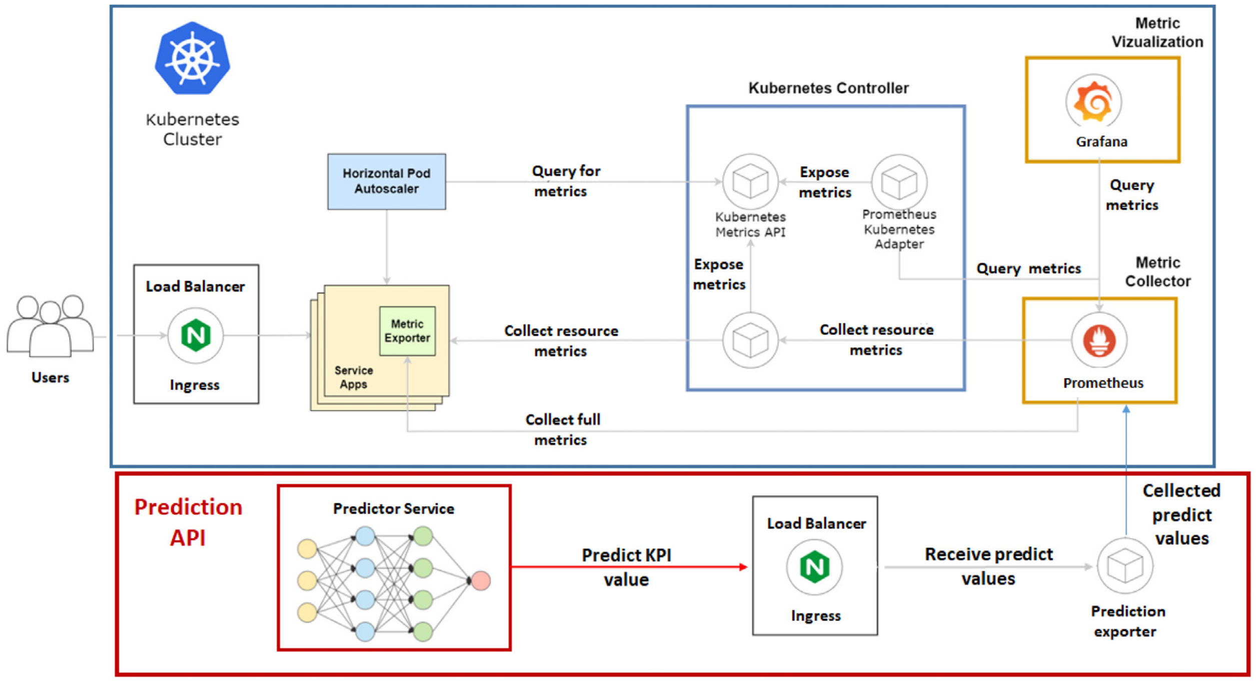


Fig. 4. High-level view of the Proactive Monitoring Framework.

This monitoring system is, however, still purely reactive and susceptible to the unpredictability of the workload characteristics of service cloud environments. To minimize the impact of these fluctuations, we have integrated a forecasting module capable of predicting resource usage and allowing the monitoring system to allocate resources in a preventive way, reducing system response time and wasted resources. Below, we present our complete solution that includes the Predictor Service.

3.2. Proactive monitoring for service provisioning

Our prediction module forecasts the network workload spikes based on machine learning algorithms, and sends predictions to monitoring system so it can proactively allocate resources. Fig. 4 contains a high-level depiction of our Proactive Monitoring Framework and its six main component: *Service Apps*, *Prometheus Monitor*, *Prometheus Kubernetes Adapter* (PKA), *Kubernetes Metrics API* (KMAPI), *HPA*, *Prediction Exporter* and *Predictor Service*, whose adaptations and differences from the previous version (Section 3.1) are below.

The *Service Apps* are now scaled according to the Key Performance Indicator (KPI) values calculated by the *Predictor Service*, which allows the applications to begin the scaling process before the workload spikes occur.

The *Prometheus Monitor* must be configured to add the *Prediction Exporter* to the scrape targets, so that it collects the predicted values from the exporter, and these metrics need to be added to the KPA in order for them to be available to the *HPA* through the KMAPI.

The *Prediction Exporter* is a standalone deployment based on a container running a GO server that acts as the liaison between the monitoring framework in the cloud and the *Predictor Service*. It queries the *Prometheus Monitor* to gather data from the metrics used for scaling the applications in a file, which is periodically sent to the *Predictor Service* as historical data to continuously train and consequently improve the accuracy of the predictor. The exporter is also a data source responsible for receiving the predicted KPI values from the *Predictor Service*, and exposing them in an HTTP endpoint which is scraped by the *Prometheus Monitor* to make the predicted values available to the *HPA*.

The Predictor Service uses a framework for distributed real-time time series forecasting [28] that makes predictions for various dynamic systems simultaneously, and provides straightforward horizontal scaling, increased modularity, high robustness and a simple interface for users.

Finally, the *Prediction Exporter* queries the *Prometheus Monitor* to collect the current value of the time series, and send it to the *Prediction API* along with their timestamp. The *Prediction API* will obtain the result of the predictor and return it to the client, along with the timestamp of the predicted point.

3.3. Main characteristics

Our Proactive Monitoring Framework is scalable, since a Prometheus instance can monitor several thousands of services. Moreover, it is versatile and can use other monitor engines, such as Thanos Prometheus,[10] to enable the aggregation of queries from several Prometheus instances and save the collected metrics in an external storage device, overcoming the scraping and storage limitations of a single replica. It is also flexible when it comes to add metrics, services or replace components, since it facilitates the addition of services and the collection of custom metrics specific to each service's requirements with the custom metrics exporters. Moreover, it can be used in any public, private or hybrid Kubernetes based cloud regardless of the provider. The Prediction Service modularity also makes it easy to replace with any other forecasting algorithm. The Prometheus monitor can be replaced by any other metric collector, although the alternatives such as Dynatrace[11] are often paid and harder to implement. The monitoring system components have the capability to monitor themselves and adjust and allocate additional resources as required, so as long as there are resources available, the system will scale and adjust to the number of active services.

10 https://thanos.io/
11 https://www.dynatrace.com/

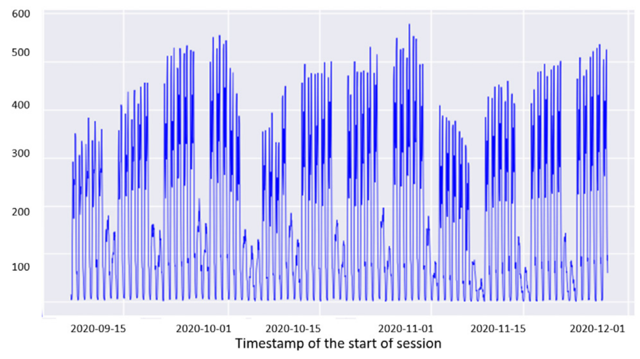


Fig. 5. Number of sessions per hour in the timeframe used for forecasting.

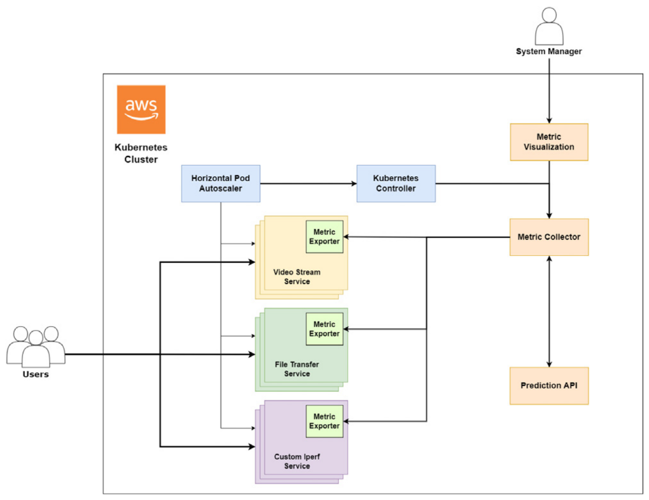


Fig. 6. High-level view of the tested services in the monitoring architecture.

4. Evaluation

The assessment of the proposed approach is performed in three steps. First, we set up a testbed with the currently most used monitoring tools in the cloud. We use this initial testbed to evaluate solutions and identify unmet challenges. Based on this assessment, we refine our testbed and perform a new assessment. The results show the efficiency of this refinement. Finally, we include a prediction service in our solution. Again, the results show the efficiency of this prediction approach.

Since this approach is tested in a cloud running services in a VANET environment, the tests use a real dataset provided by Veniam[12] from the network of vehicles in Oporto, Portugal, which contains the list of Internet sessions from the users in the vehicles, login and logout timestamps, session duration and number of bytes transferred from 2020-09-07 at 00:00:00 to 2021-04-27 at 23:00:00. Veniams' vehicular network infrastructure is a large-scale deployment of On Board Units (OBUs) in vehicles and infrastructural Road Side Units (RSUs) that enables Vehicle to Vehicle (V2V) and Vehicle to Infrastructure (V2I) communication, in the city of Porto, Portugal [29]. Currently, 600+ fleet vehicles are equipped with OBUs, pertaining to the public transportation authority (400+ buses) and waste disposal department (garbage-collection and road-cleaning trucks). Additionally, more than 50 RSUs have been deployed. The GPS traces and metadata of passenger Wi-Fi connections are collected by each OBU and stored in the backend infrastructure. The number of sessions per hour is shown in Fig. 5.

Two separate sets of tests are conducted to validate the monitoring and prediction model. The first set is run on a local small scale Kubernetes cluster to facilitate testing, and the second set is run on a larger private Kubernetes cluster owned by Veniam. The local testbed infrastructure runs on a desktop (Intel® Core™ i7-8750H CPU @ 2.20 GHz ×12, 8 GB RAM) using Ubuntu Linux, running a local Kubernetes cluster in a Minikube Virtual Machine (VM), which was used to simulate an environment where different services were being provided to users through a URL and monitored using Prometheus. The large scale tests run in an infrastructure based on the private cluster provided by Veniam running on Elastic Kubernetes Service (EKS), a managed service used to run Kubernetes on Amazon Web Services (AWS) without needing to install, operate, and maintain the Kubernetes control plane or nodes.

Amazon EKS runs and scales the Kubernetes control plane across multiple AWS Availability Zones to ensure high availability, automatically scales control plane instances based on load, detects and replaces unhealthy control plane instances, and provides automated version updates and patching for them. It is also integrated with many AWS services to provide scalability and security for your applications, including the following capabilities: Amazon Elastic Container Registry (ECR) for container images, Elastic Load Balancing for load distribution, Identity and Access Management (IAM) for authentication, and Amazon Virtual Private Cloud (VPC) for isolation. The Kubernetes cluster consists of 13 nodes, each one with a 110 pod capacity and using Amazon Linux 2.

The monitoring of CPU and memory usage, the two metrics supported by default by Kubernetes, may be inadequate for some services. Therefore, we will be running stress tests on three different services deployed in a Kubernetes environment, a *Video Stream Service*, a *File Transfer Service* and a *Custom Iperf Service*. Fig. 6 shows a high level architecture of the monitoring system and the deployed services.

The *Video Stream Service* is based on NGINX,[13] an open source software that provides a Real Time Media Protocol (RTMP) HTTP Live Streaming (HLS) module to be used in the streaming server. We use Fast Forward Motion Picture Experts Group (FFmpeg) to encode and publish in real-time a looped video file in H.264 format with a 1280 × 720 pixel resolution at a frame rate of 24 frames per second. The NGINX server allows clients to stream the video through an HTTP endpoint.

The *File Transfer Service* is an HTTP server written in GO language and uses the net/http package. The server receives HTTP requests from clients and responds with a randomly generated file. The clients can send requests to different endpoints in order to obtain files of different sizes.

The *Custom Iperf Service*[14] is a standard Iperf3 server, a tool for active measurements of the maximum achievable bandwidth on IP networks that supports tuning of various parameters related to timing, buffers and protocols (TCP, User Datagram Protocol (UDP), Stream Control Transmission Protocol (SCTP) with IPv4 and IPv6) written in C, and a mixer module, developed by Veniam, that creates multiple instances of the Iperf3 server, up to a preset limit, allowing a single instance of the service to receive multiple client sessions simultaneously.

Since it is impossible to differentiate the type of sessions recorded in the Veniam dataset, we will be making an estimate

12 https://veniam.com/

13 https://www.nginx.com/resources/glossary/nginx/

14 https://iperf.fr/

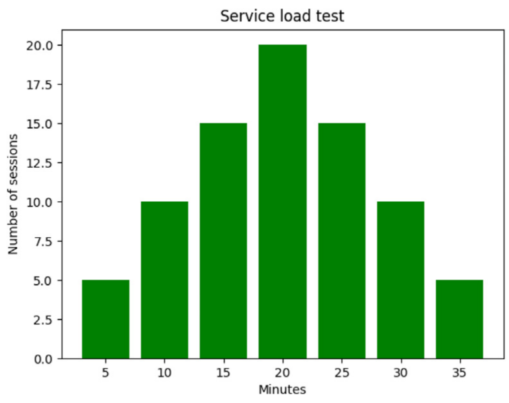


Fig. 7. Number of sessions in the load test.

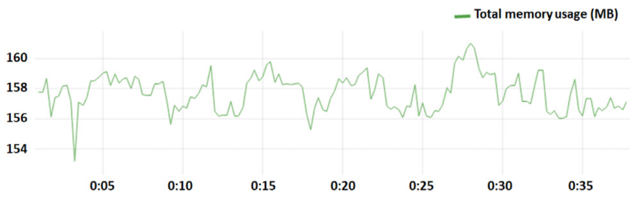


Fig. 8. Video Stream memory usage with resource metrics.

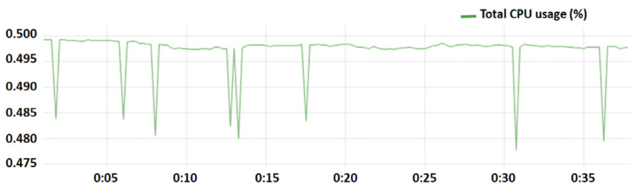


Fig. 9. Video Stream CPU usage with resource metrics.

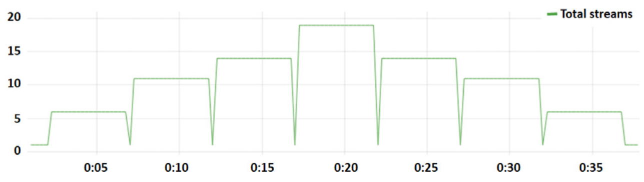


Fig. 10. Video Stream active streams with custom metrics.

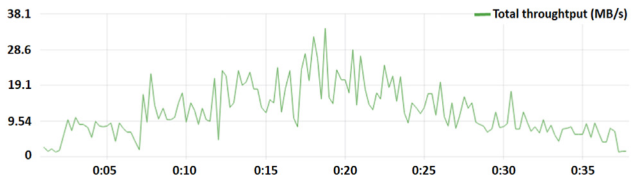


Fig. 11. Video Stream throughput with custom metrics.

based on the percentage values from Sandine's report [30]. According to the study presented in this report, 64% of the total sessions could be associated with video streams, 8% with file transfers and Iperf will be given 1% of the total sessions. Stress tests based on these values were created to establish a baseline for each service's memory, CPU and throughput requirements, allowing the discovery of bottlenecks for each specific service.

4.1. Basic monitoring system evaluation

To determine whether the metrics server and resource metrics pipeline were sufficient to accurately monitor the service applications, we developed a simple load test. First, there is a steadily increasing number of sessions, and afterwards the number of sessions decreases again on steady one minute intervals, as shown in Fig. 7.

The test consisted of simulating the number of sessions in each service and then analyzing the memory and CPU usage metrics to see if all services can be accurately profiled using these metrics.

Fig. 8 shows the memory usage, and Fig. 9 shows the CPU usage for the video stream service.

Neither of the metrics is able to accurately profile the status of the service and the incoming connections, since both metrics remain constant during the full test, with minimal variations that are not related to the incoming streams.

From the above tests, we can conclude that, exclusively monitoring memory and CPU usage is insufficient to have an accurate monitoring system. Even though there may exist services for which these two metrics are sufficient, such as the file transfer service, for video stream service and the custom Iperf service, these metrics fail to give an accurate representation of the service status and are unsuitable to be used as the basis for scaling decisions. This means that, in order to accurately monitor the services, additional metrics must be gathered from the services, which will be presented in the next section.

4.2. Custom metrics monitoring system evaluation

To determine whether additional metrics could be used to monitor the service applications, we repeated the load test shown in Fig. 7 to analyze the throughput and active users graphs to determine if they can be used to accurately profile the service load.

Fig. 10 shows the number of active video streams in the server at a given time. This metric is a strong candidate to be used in the autoscaler, since it shows the load of the service without too many fluctuations, making it easier to establish thresholds for the autoscaler. Fig. 11 shows the throughput out of the video stream service during the test. This metric is able to represent the status of the service, since throughput is directly related to the number of active streams. However, the rapid fluctuations in the values due to the accumulated changes in the throughput rate of each individual stream make it difficult to establish thresholds for autoscaler.

The custom exporters are a significant improvement, since they allow to configure any additional metric which may seem necessary for a specific service instead of relying solely on the metrics available from the metric server.

4.3. Service stress tests

Before implementing the HPA, we need to analyze each services' behavior and identify which resources are critical to its execution, and the amount of pods so that the service runs smoothly. The load tests were based on the values of the second to last day, since the last day corresponds to a holiday with anomalous sessions numbers (Fig. 12).

To evaluate the benefits of automatically scaling applications based on the collected metrics, we must first determine which metrics are most critical to the correct operation of the service. Fig. 13 shows that the memory usage reaches the maximum possible value regardless of the number of active streams.

This behavior makes this metric unsuitable to be used in the Autoscaler, and shows that the parameters monitored by default

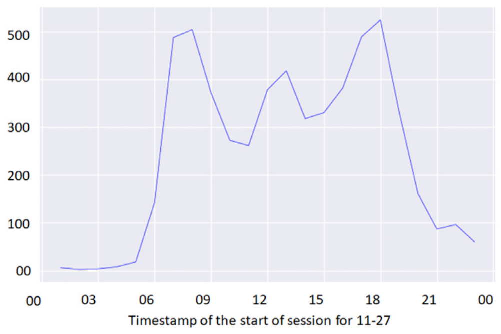


Fig. 12. Number of sessions per hour in a single day.

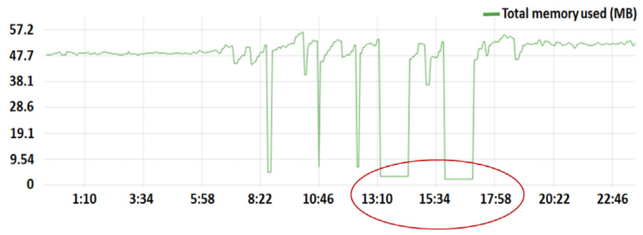


Fig. 13. Video Stream memory usage without HPA.

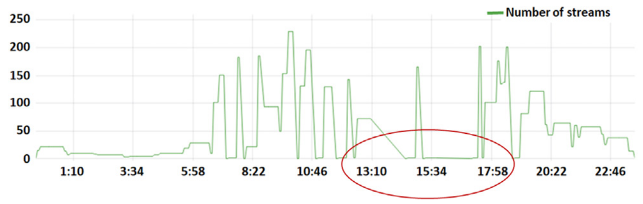


Fig. 14. Video Stream active streams without HPA.

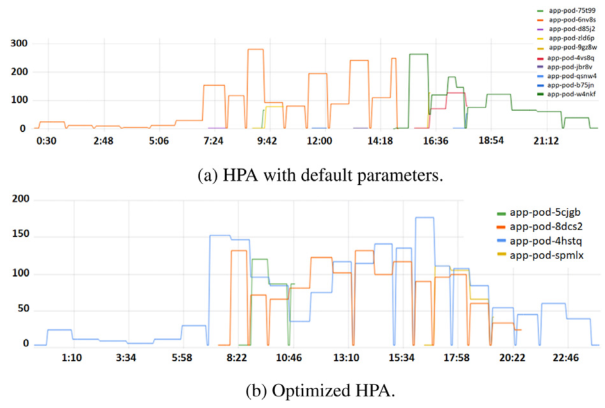


Fig. 15. Video Stream active streams per pod with HPA.

in Kubernetes are, in some cases, not capable of giving an accurate depiction of the status of the service. This indicates the need for additional specific metrics. In this way, the values of active streams shown in Fig. 14 are much more representative of the status of the service. This test was done without any automatic scaling (without HPA), therefore, no additional replicas were created to respond to the increase of users. As we can highlight by the circles in Figs. 13 and 14, if too many streams are placed on a single container at once, its performance starts to degrade up to the point where it completely stops responding. These intervals correspond to the peak of active sessions where a single instance of the service is insufficient to respond to the number of active users.

This situation shows the importance of allocating resources dynamically to prevent both scenarios where the services are unable to respond to the incoming workload and where resources are being wasted during periods of low workloads.

4.4. Stress tests with HPA

Once the bottlenecks for the services are identified, HPA can be implemented based on the most appropriate metric for each service in order to keep the bottleneck metric within a certain limit, so that it does not become harmful to the correct operation of the service. Fig. 15 shows the active sessions and the number of instantiated pods during our tests with the active HPA. First, we deploy HPA to each service with default settings to assess its impact on the service's performance (Fig. 15(a)). After that, to improve the quality of service and take full advantage of the autoscaler, we made changes to its configuration and reduced its reactivity by adding a stabilization period to the raise and lower operations, thus reducing the orchestrator's workload by creating and deleting pods quickly (Fig. 15(b)).

As can be observed in Fig. 15(a), it is notorious that the HPA is able to keep the metrics that are being tracked for each service within acceptable values, and is able to prevent the service from stopping to respond to higher workloads. However, to improve the quality of service and take full advantage of the Autoscaler, we must change its configuration to reduce its reactiveness by adding a stabilization period to both scaling up and down operations, reducing the workload placed on the orchestrator.

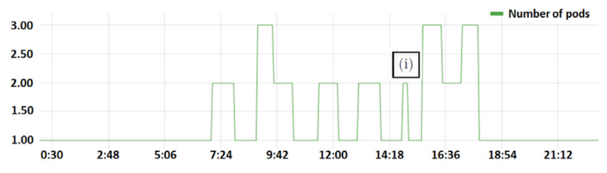
Fig. 15(b) shows the results obtained after the inclusion of a stabilization period. Choosing the stabilization period requires the value of a metric to be above or below the threshold for a configurable period of time before signaling a scale up or down. Additionally, it requires a policy to limit the rate at which pods are added or removed to 1 every 15 s. These conditions are added to the HPA configuration and are used to consider keeping the pods longer even if the metric value may be below the threshold, reducing the load on the orchestrator without leading to situations where we overrun the provision of resources for long periods of time.

The HPA's efficiency can also be observed through the number of instanced pods, as shown in Fig. 16. Fig. 16(a) shows that although the autoscaler has provided a general improvement in service performance, there is still room for optimization, as can be seen in (i), where an additional pod was deleted to be created again shortly thereafter. This behavior places additional workload on the orchestrator and prematurely terminates all active sessions running in the pod, which would appear to clients as a service failure.

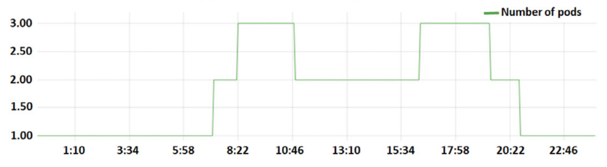
Similarly to the tests with the default configuration of the Autoscaler, the streams are split between the pods, preventing a single pod from being saturated and unable to respond to the clients. However, there are far less instances of pods being created or deleted during this test, which can be confirmed in Fig. 16(b). With default configurations, a total of 10 pods are used during the test. But, the addition of the stabilization period reduced the reactivity of the Autoscaler which made pod deletion less frequent, and so the total number of pods used for this test was only 4, resulting in a lesser load for the Orchestrator and a smaller number of dropped streams on pod deletions.

4.5. Stress tests with machine learning predictor

By itself, the Autoscaler with stabilization period is an effective way of responding to large spikes in client requests. However,



(a) HPA with default parameters.



(b) Optimized HPA.

Fig. 16. Video Stream number of pods with HPA.

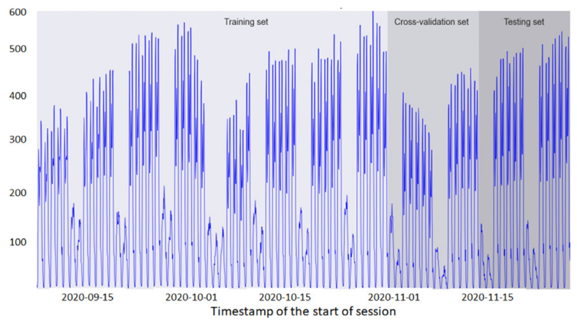


Fig. 17. Vehicular network dataset — Number of sessions per hour.

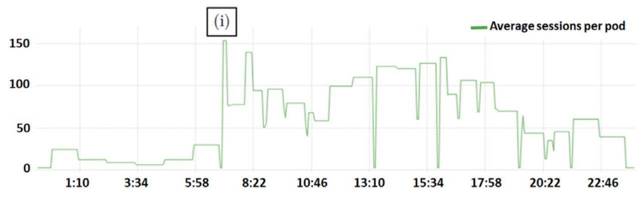


Fig. 18. Active streams per pod without the prediction module.

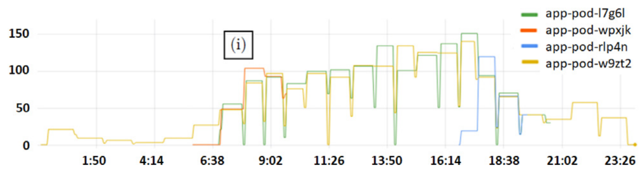


Fig. 19. Average active streams with prediction.

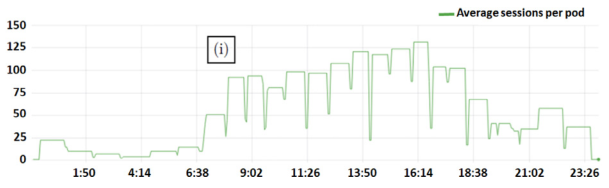


Fig. 20. Average sessions per pod using the prediction module.

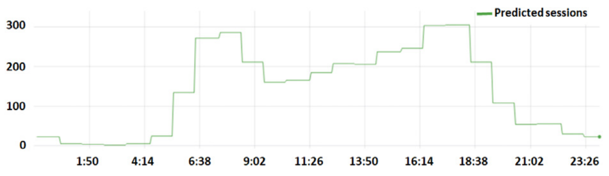


Fig. 21. Predictor — Forecasted number of sessions.

it is a reactive system, and there is a delay in the response to these spikes. By adding a predictive module, we can minimize this delay using machine learning algorithms to forecast the number of incoming sessions and thus ensuring the desired QoS.

First, we need to create a model to forecast the number of sessions in the vehicular network. The forecasting task consists of, given the past values of the number of sessions per hour in the vehicular network $y(0), y(1), \dots, y(t)$ and other possible additional features, forecast the value $y(t+1)$, where $y(t)$ represents the number of sessions in the network at hour t.

With the current dataset, two weeks will be used as cross-validation data (from the 31 of October to the 14 of November) to optimize the forecasting models, and the last two weeks (from the 14 of November to the 28 of November) will be used as test data, to perform the final tests and calculate the performance metrics. The previous weeks (from the 9 of September to the 31 of October) will be used as training data (Fig. 17).

In this work, we chose to use the predictor module developed by Ferreira et al. [28]. The performance metric used to evaluate the accuracy of the forecasts will be the Root Mean Square Error (RMSE), which is the square root of the average of squared differences between the forecasts and the actual observations. Predictors with lower RMSE values are more accurate than those with higher RMSE values.

Next, we discuss the performance of the monitoring and management system with the prediction service. Fig. 18 shows the distribution of sessions in pods using the optimized Autoscaler, without the prediction module. Since the Autoscaler needs to wait for session spikes to occur before it commands the Orchestrator to create an additional pod, the original pod will have to endure the user spike by itself, which can cause the degradation of service performance. The annotation (i) shows a period in which a spike of users occurred. During these periods, the average number of sessions in a single pod may far exceed its threshold and cause the pod to stop responding.

The addition of a forecasting module enables the Autoscaler to anticipate the client spikes and react preemptively, minimizing the time a pod exceeds the threshold of active sessions. Fig. 19 depicts the distribution of sessions per pod using the prediction module. In this case, the Autoscaler creates additional pods preemptively, allowing the pods to have time to start up before the spike of clients actually occurs in the period indicated by annotation (i).

Fig. 20 shows the distribution of streams per pod, and how the predictor enables the autoscaler to anticipate client spikes and create additional pods preemptively (i).

Fig. 21 shows the number of sessions forecasted by the prediction module.

The predicted values are very similar to the real ones shown in Fig. 22, demonstrating the accuracy of the predictor.

4.6. Overview of experiments and discussion

The initial tests were performed in a local Kubernetes cluster with limited resources with the objective of determining if the tools provided by Kubernetes were sufficient to accurately monitor any generic service that may be deployed. Following, we tested the custom metrics pipeline using the Prometheus framework to complement the metrics server and enable the

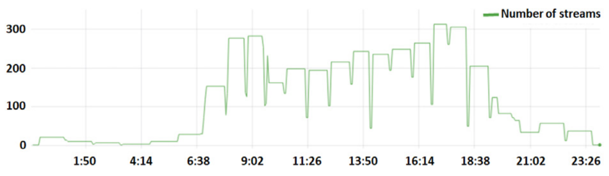


Fig. 22. Average active streams per pod.

collection of custom service-specific metrics. This model proved to be much more effective as it allows the operator to add new metrics for each service as needed, and was used as a basis for the tests below.

The next objective was to take advantage of the collected metrics to make automated scaling decisions based on them. The tests were done in the Veniam Kubernetes cluster, since the local deployment lacked the resources to simulate the required numbers of users. Initially, the tests were done without any scaling algorithms, which led to situations where the services crashed and/or stopped responding because they lacked the resources to respond to the amount of users being simulated, underlining the need for an automated scaling system capable of responding to workload spikes in real time. A scaling system was proposed using the Kubernetes HPA, and the custom metrics were collected from the services to trigger scaling operations whenever the selected metric surpassed certain thresholds. This addition was not enough to provide major benefits, since the Autoscaler was far too reactive to changes in the workloads which were themselves highly volatile. In fact, the Autoscaler was permanently scaling up and down, disconnecting users and placing a big workload in the orchestrator, so in order to make the system viable, the algorithm had to be reviewed.

The most effective way to reduce Autoscaler reactivity was to introduce a stabilization period on the Autoscaler, either by increasing or decreasing the scale. This modification greatly improved the system performance, services still reacted quickly to changes in workloads, preventing services from running out or wasting resources. While efficient, the monitoring system is still not fast enough to prevent any variation triggering a scaling operation, reducing the load placed on the orchestrator and user disconnection rates.

Besides, we evaluated the integration of a prediction service that allows the system to anticipate workloads variations and adjust proactively instead of reactively. We began by analysing the research done to select the most accurate predictor for our scenario, and used the vehicular dataset previously referred for training the predictor. The results of the load tests with the predictor module show reduced periods of threshold violations, since the orchestrator is able to proactively allocate resources, improving the response time of the system and preventing service degradation caused by insufficient resources.

In short, we managed to identify shortcomings of the default Kubernetes monitoring tools and design a monitoring system capable of overcoming them and adapting to the different requirements of any generic service while allocating resources as necessary rapidly and efficiently. Made available through the inclusion of the stabilization period, our solution improves cloud performance by providing an Autoscaler with less dynamism in terms of POD management. The Video Stream case presented illustrates the functionality of our solution, but it is important to emphasize that our service-based architecture is versatile and scalable, allowing greater granularity through the simultaneous use of multiple Predictor instances. Furthermore, our solution is customizable, allowing the use of different algorithms/machine learning models, which is not available in cloud player solutions where predictive intelligence is encapsulated. Finally, we used the configuration made for the AWS environment, but our solution is cloud agnostic and can be used in any environment, be it Azure or Google, among others. Also due to this characteristic, it can be used in a multi-cloud environment in public and private clouds.

5. Conclusion and future work

This article proposed a framework for a Kubernetes based cloud, focusing on collecting comprehensive metrics about agnostic services with different requirements, on automating scaling operations and predicting load surges using a forecasting module. The monitoring and management system was extended to provide autoscaling and to include a load Prediction Service that allows the system to act preventively in order to maintain the desired QoS while dealing with the efficient use of resources in peaks of low consumption. The Predictor Service publishes predicted user session values for the next time period as service metrics, which are collected by the monitoring system and used by the orchestrator to make predictive scheduling decisions. By using these service metrics, the monitoring system improved the response time on escalation responses and reduced service degradation. These improvements were made possible by variable workloads compared to reactive monitoring systems.

The proposed system is well suited for micro services environments, due to its ability to collect customized real time performance metrics from services. This allows system managers to overcome the limitations of the standalone Kubernetes monitoring tools, and have a complete view of status of the applications deployed, detect malfunctions and configure alerts. The system can easily integrate workload predictors that enable it to predict variations in service workloads, and ensures the provisioning of resources to the deployed services by automatically scaling applications according to workloads, minimizing the waste of resources.

As future work, there are some elements that can be enhanced: monitor and analyze additional pods, such as load balancers for an even more complete view of the system; automate the behavioral analysis of services to discover which of the collected metrics is critical for each service; and create service specific predictors instead of a central prediction module to reduce the load placed on the prediction module, and increase the modularity of the system.

CRediT authorship contribution statement

Gonçalo Marques: Conceptualization, Investigation, Validation, Writing – original draft. **Carlos Senna:** Conceptualization, Investigation, Validation, Formal analysis, Writing – original draft, Writing – review & editing, Supervision. **Susana Sargento:** Conceptualization, Investigation, Validation, Formal analysis, Writing – original draft, Writing – review & editing, Supervision. **Luís Carvalho:** Writing – original draft, Supervision. **Luís Pereira:** Writing – original draft, Supervision. **Ricardo Matos:** Writing – original draft, Supervision.

Data availability

The data that has been used is confidential.

Acknowledgment

This work is supported by the European Union/Next Generation EU, through Programa de Recuperação e Resiliência (PRR) Project Nr. 29: Route 25.

References

[1] W.H.C. Almeida, L. de Aguiar Monteiro, R.R. Hazin, A.C. de Lima, F.S. Ferraz, Survey on microservice architecture-security, privacy and standardization on cloud computing environment, in: ICSEA 2017, 2017, p. 210.

[2] C. Esposito, A. Castiglione, K.-K.R. Choo, Challenges in delivering software in the cloud as microservices, IEEE Cloud Comput. 3 (5) (2016) 10–14, http://dx.doi.org/10.1109/MCC.2016.105.

[3] P. Jamshidi, C. Pahl, N.C. Mendonça, J. Lewis, S. Tilkov, Microservices: The journey so far and challenges ahead, IEEE Softw. 35 (3) (2018) 24–35, http://dx.doi.org/10.1109/MS.2018.2141039.

[4] K. Naseer Qureshi, F. Bashir, S. Iqbal, Cloud computing model for vehicular ad hoc networks, in: 2018 IEEE 7th International Conference on Cloud Networking, CloudNet, 2018, pp. 1–3, http://dx.doi.org/10.1109/CloudNet.2018.8549536.

[5] C.B. Hauser, S. Wesner, Reviewing cloud monitoring: Towards cloud resource profiling, in: 2018 IEEE 11th International Conference on Cloud Computing, CLOUD, 2018, pp. 678–685, http://dx.doi.org/10.1109/CLOUD.2018.00093.

[6] D.C. Marinescu (Ed.), Chapter 9 - Cloud resource management and scheduling, in: Cloud Computing, second ed., Morgan Kaufmann, 2018, pp. 321–363, http://dx.doi.org/10.1016/B978-0-12-812810-7.00012-1.

[7] K. Braiki, H. Youssef, Resource management in cloud data centers: A survey, in: 2019 15th International Wireless Communications & Mobile Computing Conference, IWCMC, 2019, pp. 1007–1012, http://dx.doi.org/10.1109/IWCMC.2019.8766736.

[8] P. Kumar, R. Kumar, Issues and challenges of load balancing techniques in cloud computing: A survey, ACM Comput. Surv. 51 (6) (2019) http://dx.doi.org/10.1145/3281010.

[9] S.-Y. Huang, C.-Y. Chen, J.-Y. Chen, H.-C. Chao, A survey on resource management for cloud native mobile computing: Opportunities and challenges, Symmetry 15 (2) (2023) http://dx.doi.org/10.3390/sym15020538, URL https://www.mdpi.com/2073-8994/15/2/538.

[10] R. Jeyaraj, A. Balasubramaniam, M.A. Kumara, N. Guizani, A. Paul, Resource management in cloud and cloud-influenced technologies for internet of things applications, ACM Comput. Surv. 55 (12) (2023) http://dx.doi.org/10.1145/3571729.

[11] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, J. Wilkes, Autopilot: Workload autoscaling at google, in: Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys '20, Association for Computing Machinery, New York, NY, USA, 2020, http://dx.doi.org/10.1145/3342195.3387524.

[12] I.K. Kim, W. Wang, Y. Qi, M. Humphrey, Empirical evaluation of workload forecasting techniques for predictive cloud resource scaling, in: 2016 IEEE 9th International Conference on Cloud Computing, CLOUD, 2016, pp. 1–10, http://dx.doi.org/10.1109/CLOUD.2016.0011.

[13] Y. Garí, D.A. Monge, E. Pacini, C. Mateos, C. García Garino, Reinforcement learning-based application autoscaling in the cloud: A survey, Eng. Appl. Artif. Intell. 102 (2021) 104288, http://dx.doi.org/10.1016/j.engappai.2021.104288, URL https://www.sciencedirect.com/science/article/pii/-S0952197621001354.

[14] F.J. Baldan, S. Ramirez-Gallego, C. Bergmeir, F. Herrera, J.M. Benitez, A forecasting methodology for workload forecasting in cloud systems, IEEE Trans. Cloud Comput. 6 (4) (2018) 929–941, http://dx.doi.org/10.1109/TCC.2016.2586064.

[15] V. Simic, B. Stojanovic, M. Ivanovic, Optimizing the performance of optimization in the cloud environmentâ€"an intelligent auto-scaling approach, Future Gener. Comput. Syst. 101 (2019) 909–920, http://dx.doi.org/10.1016/j.future.2019.07.042, URL https://www.sciencedirect.com/science/article/pii/-S0167739X18321496.

[16] S. Lee, S. Son, J. Han, J. Kim, Refining micro services placement over multiple kubernetes-orchestrated clusters employing resource monitoring, in: 2020 IEEE 40th International Conference on Distributed Computing Systems, ICDCS, 2020, pp. 1328–1332, http://dx.doi.org/10.1109/ICDCS47774.2020.00173.

[17] S.N. Srirama, M. Adhikari, S. Paul, Application deployment using containers with auto-scaling for microservices in cloud environment, J. Netw. Comput. Appl. 160 (2020) 102629, http://dx.doi.org/10.1016/j.jnca.2020.102629, URL https://www.sciencedirect.com/science/article/pii/-S108480452030103X.

[18] H. Wang, Y. Ma, X. Zheng, X. Chen, L. Guo, Self-adaptive resource management framework for software services in cloud, in: 2019 IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking, ISPA/BDCloud/SocialCom/SustainCom, 2019, pp. 1528–1529, http://dx.doi.org/10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00223.

[19] L. Chen, Y. Xu, Z. Lu, J. Wu, K. Gai, P.C.K. Hung, M. Qiu, IoT microservice deployment in edge-cloud hybrid environment using reinforcement learning, IEEE Internet Things J. 8 (16) (2021) 12610–12622, http://dx.doi.org/10.1109/JIOT.2020.3014970.

[20] H. Zhao, H. Lim, M. Hanif, C. Lee, Predictive container auto-scaling for cloud-native applications, in: 2019 International Conference on Information and Communication Technology Convergence, ICTC, 2019, pp. 1280–1282, http://dx.doi.org/10.1109/ICTC46691.2019.8939932.

[21] S. Zhou, J. Li, K. Zhang, M. Wen, Q. Guan, An accurate ensemble forecasting approach for highly dynamic cloud workload with VMD and R-transformer, IEEE Access 8 (2020) 115992–116003, http://dx.doi.org/10.1109/ACCESS.2020.3004370.

[22] B. Liu, J. Guo, C. Li, Y. Luo, Workload forecasting based elastic resource management in edge cloud, Comput. Ind. Eng. 139 (2020) 106136, http://dx.doi.org/10.1016/j.cie.2019.106136.

[23] Y. Zhu, W. Zhang, Y. Chen, H. Gao, A novel approach to workload prediction using attention-based LSTM encoder-decoder network in cloud environment, EURASIP J. Wireless Commun. Networking 2019 (1) (2019) http://dx.doi.org/10.1186/s13638-019-1605-z.

[24] B. Thurgood, R.G. Lennon, Cloud computing with kubernetes cluster elastic scaling, in: Proceedings of the 3rd International Conference on Future Networks and Distributed Systems, ICFNDS '19, Association for Computing Machinery, New York, NY, USA, 2019, http://dx.doi.org/10.1145/3341325.3341995.

[25] A.A. Khaleq, I. Ra, Intelligent autoscaling of microservices in the cloud for real-time applications, IEEE Access 9 (2021) 35464–35476, http://dx.doi.org/10.1109/ACCESS.2021.3061890.

[26] G. Yu, P. Chen, Z. Zheng, Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach, IEEE Trans. Cloud Comput. 10 (2) (2022) 1100–1116, http://dx.doi.org/10.1109/TCC.2020.2985352.

[27] M.-N. Tran, D.-D. Vu, Y. Kim, A survey of autoscaling in kubernetes, in: 2022 Thirteenth International Conference on Ubiquitous and Future Networks, ICUFN, 2022, pp. 263–265, http://dx.doi.org/10.1109/ICUFN55119.2022.9829572.

[28] D. Ferreira, C. Senna, S. Sargento, Distributed real-time forecasting framework for IoT network and service management, in: NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, IEEE Press, 2020, pp. 1–4, http://dx.doi.org/10.1109/NOMS47738.2020.9110456.

[29] P.M. Santos, J.G.P. Rodrigues, S.B. Cruz, T. Lourenço, P.M. d'Orey, Y. Luis, C.l. Rocha, S. Sousa, S. Crisóstomo, C. Queirós, S. Sargento, A. Aguiar, J. Barros, PortoLivingLab: An IoT-based sensing platform for smart cities, IEEE Internet Things J. 5 (2) (2018) 523–532, http://dx.doi.org/10.1109/JIOT.2018.2791522.

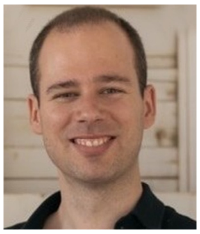
[30] Sandvine, Global internet phenomena report, Technical Report, Sandvine, 2012.

Gonçalo Marques received the title of M.Sc. degree in computer engineering and telematics from the University of Aveiro, Aveiro, Portugal, in 2021. He worked on different projects at the Instituto de Telecomunicações, Aveiro, including implementation and management of services and applications in cloud systems. Currently he works as a software engineer at Adobe Commerce, Barcelona, Spain.

Dr. Carlos Senna, is a researcher at Instituto de Telecomunicações (IT) in Aveiro, Portugal, received the Ph.D. degree s in Computer Science from the University of Campinas, Brazil in 2014. He is working in research related to ad-hoc and vehicular networking mechanisms and protocols, resource management, orchestration, and distributed AI for cloud/fog/edge based solutions. Over the last years he was involved in several research projects (Horizon Project: A New Horizon to The Internet, Support for Orchestration of Resilient and Reliable Services In the Fog (SORTS), and Europe - Brazil Collaboration of Big Data Scientific Research Through Cloud-Centric Applications (EUBRA BIGSEA)), S2MovingCity: Sensing and Serving a Moving City, P2020 SAICT- PAC/0011/2015, MobiWise: from Mobile Sensing to Mobility Advising, and 5G Mobilizer project "Components and services for 5G networks''. Currently, he is involved in projects such as, Efficient Information Centric Networks for IoT Infrastructure (InfoCent-IoT), SNOB5G, MH-SDVANET, and Aveiro STEAM City.

Susana Sargento is a Full Professor in the University of Aveiro and a senior researcher in the Institute of Telecommunications, where she is leading the Network Architectures and Protocols (NAP) group (https://www.it.pt/Groups/Index/62). She received her Ph.D. in 2003 in Electrical Engineering in the University of Aveiro, being a visiting student at Rice University in 2000 and 2001. Since 2002 she has been leading many national and international projects, and worked closely with telecom operators and OEMs. She has been involved in several FP7 projects (4WARD, Euro-NF, C-Cast, WIP, Daidalos, C-Mobile), EU Coordinated Support Action 2012-316296 "FUTURE-CITIES", EU Horizon 2020 5GinFire, EU Steam City, and CMU-Portugal projects (S2MovingCity, DRIVE-IN with the Carnegie Melon University) and MIT-Portugal Snob5G project. She has been TPC-Chair and organized several international conferences and workshops, such as ACM MobiCom, IEEE Globecom and IEEE ICC. She has also been a reviewer of numerous international conferences and journals, such as IEEE Wireless Communications, IEEE Networks, IEEE Communications. She was the founder of Veniam (www.veniam.com), which builds a seamless lowcost vehicle-based internet infrastructure, and she was the winner of the 2016 EU Prize for Women Innovators. Susana is also the cocoordinator of the national initiative of digital competences in the research axis (INCoDe.2030, http://www.incode2030.gov.pt/), belongs to the evaluation committee of the Fundo200M (www.200~m.pt) government co-investment and funding), and she is one of the Scientific Directors of CMU-Portugal Program (http://www.cmu-portugal.org/). Her main research interests are in the areas of selforganized networks, in ad-hoc and vehicular network mechanisms and protocols, such as routing, mobility, security and delay-tolerant mechanisms, resource management, and content distribution networks. She regularly acts as an Expert for European Research Programs.

Luís Carvalho, M.Sc., finished his integrated master in Electrical and Computers Engineering from the University of Porto in 2009. He started his career as a researcher in the areas of Personal Area Networks and Ambient Assisted Living. In 2016 Luís joined Veniam as a Systems Engineer and progressed to Lead Cloud Engineer, driving the technical aspects of Veniam's cloud solution. Since November 2022, Luís is a Senior Golang Developer at CryptoCompare.



Luís Pereira, M.Sc. in Communications Engineering (University of Minho – 2008) and M.Sc. in Business Administration (Porto Business School – 2024), has 15 years of IT experience and is an expert in the areas of information security and cloud computing. He worked as Veniam's head of cloud and security, and now leads cloud services and infrastructure projects always with an emphasis on the information security domain. He is currently visiting professor at ISPGAYA and Senior Cloud & Security Manager at Nexar Inc. Porto, Portugal.

Ricardo Matos (male) is currently Principal Product Manager at Nexar. He received his Ph.D. in Electrical Engineering by the University of Aveiro in January 2013. At the same time, he joined Veniam as System Engineer. In Veniam, he has been involved in the research of many networking aspects (mobility, data management, delay tolerant data, etc.), IP management, product management and definition (specifically aligned with connected car, autonomous vehicles and future mobility services), benchmarking and research activities, engineering management for 1 year, among others. On August 2022, he joined Nexar as Principal Product Manager. He has been involved in several national funding projects, as well as in the FP6 European Project WEIRD, FP7 European Project Euro-NF, and RETHINK BIG.