Contents lists available at ScienceDirect

Computer Communications

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

Attribute-based policies through microservices in a smart home scenario

Alessandra Rizzardi, Sabrina Sicari *, Alberto Coen-Porisini

Università degli Studi dell'Insubria, Dipartimento di Scienze Teoriche e Applicate, via O. Rossi 9, 21100, Varese, Italy

ARTICLE INFO

Keywords:
Microservices
Kubernetes
Attribute-based policies
Smart home
Internet of Things

ABSTRACT

Application containerization allows for efficient resource utilization and improved performance when compared to traditional virtualization techniques. However, managing multiple containers and providing services such as load balancing, fault tolerance and security represent challenging tasks in the emerging microservices architectures. In this context, Kubernetes platform allows to build resilient distributed containers. Besides its efficiency in terms of configuration and architectural resiliency, it must also guarantee the access control to the managed resources. In fact, information must be protected throughout the different microservices which compose an application. To cope with such an issue, this paper proposes the definition of attribute-based policies able to regulate data disclosure within a Kubernetes-based microservices network. Simulations are carried out in a local Minikube environment, considering a smart residence scenario. The investigated metrics include response time, required memory, CPU load, and disk usage.

1. Introduction

In recent years, network traffic has been enormously grown. Such a trend forced many companies and industries to adopt a new kind of system architecture able to meet new efficiency requirements, including reducing response times. Hence, most of the monolithic systems, usually composed of a complex non-replicated component that runs a heavy application full of modules and external dependencies, have been replaced with microservices-based systems. Monolithic applications are also complex to maintain, since they require a long time to deploy new features, the system modules are strongly coupled and do not scale efficiently with the increase of the workload. To solve such limitations, many companies migrated their aged systems to the microservices-based architecture, and researchers analyzed the related costs and benefits [1–4].

The main scope of the microservices-based architecture is to split monolithic application responsibilities into many separated and lightweight modules, in order to reduce the size of each component, simplify the maintenance activities and accelerate the deployment phase. The advantages introduced by the system modularization lead to the improvement of system's scalability. Even if the new microservices-based architecture overcomes most of the monolithic system limitations, it is still complex to manage [5]. In fact, the lightweight modules are executed on physical or virtual machines that require ad-hoc configuration in order to run correctly. Moreover, creating a new module replica implies every time a setup phase that requires configuring a new machine, creating the required network rules, deploying the application, and verifying that it is working correctly. Such time-consuming tasks should be manually performed; for such a reason, companies and researchers were driven to find a mechanism able to automatically take care of all those aspects. A solution is represented by the Kubernetes[1] open-source platform, also known as $K8S$.

Kubernetes introduced a new automatic and faster way to define cluster architecture based on container technology, able to rapidly deploy a new cluster or upgrade an existing one. Today Kubernetes is adopted by many large companies like $Spotify$, *The New York Times*, and $Google$. Besides the advantages in terms of configuration and performance, another crucial aspect to consider is related to security, since protection of information managed inside and among microservices is fundamental in order to preserve unauthorized accesses. Kubernetes provides some security mechanisms to improve the overall network and cluster management security, allowing to define a set of cluster level permissions. However, Kubernetes is not able to provide application-level security policies, required to prevent users to access restricted information. For this reason, an application-level security mechanism is required, mainly in certain heterogeneous scenarios, such as those related to Internet of Things (IoT). Kubernetes has proven to be a valid solution for all environments with scalability requirements. This aspect, combined with the rapid rise in IoT network traffic, has driven researchers to analyze and implement scalable cluster systems based on Kubernetes. For this reason, different works have been carried out to analyze Kubernetes clusters in IoT environments like smart cities, smart

* Corresponding author.
E-mail address: sabrina.sicari@uninsubria.it (S. Sicari).
[1] https://kubernetes.io/.

https://doi.org/10.1016/j.comcom.2024.108039
Received 7 August 2024; Received in revised form 17 November 2024; Accepted 15 December 2024
Available online 20 December 2024

homes, and edge computing [6–8]. Such solutions do not consider the presence of many security risks, as we do in this work. In fact, the disclosure of information among decoupled microservices must be regulated as well as the interaction with users. Native integration to such an issue is not available, hence we propose a feasible application-level solution.

In this paper, attribute-based policies have been designed and implemented within a Kubernetes-based microservices network at the application level, simulating a smart residence scenario using a dataset composed of sensor measurements. Policies will reflect the authorized exchanges among microservices and interactions with involved entities (e.g., users with different roles). The system has been executed in a local Minikube environment and extensive simulations have been executed; the performance of the system has then been evaluated to better understand the capabilities of the envisioned solution. To collect the required metrics, the cluster has been equipped with a monitoring console. The cluster definition is publicly available[2] along with the core logic and the procedures used to set up the system. Besides the envisioned approach is presented in a smart home scenario, it can be easily adapted in cross-domain applications.

The remainder of this paper is organized as follows. Section 2 investigates the actual state of the art solutions. Section 3 presents the technologies and tools adopted in this work, paying particular attention to the orchestration framework. Section 4 presents the system architecture and core logic implementation, while Section 5 outlines the experimental phase and provides a discussion about the proposed approach. Finally, Section 6 ends the paper, drawing some hints for future research directions.

2. Related works and motivations

Kubernetes is a well-known platform, which has been adopted by many companies and awakened the interest of many researchers. For such a reason, many related works have been carried out to deeply analyze the capabilities and limits of such a technology, paying particular attention to the Kubernetes cluster performance, the Kubernetes autoscaling feature, the fault-tolerance, and its application in IoT environments.

An example is the work proposed in [9], which analyzed if Kubernetes is able to grant the availability property; the simulations performed show that Kubernetes by default is able to repair from a failure state, anyway, this operation requires some time based on the cluster's complexity; this can be translated to a minimum downtime that leads to a decrease of the QoS (Quality of Service). A similar work has been carried out in [10] that analyzes the aging and fault tolerance of a microservices system based on Kubernetes. Even in this case, the result obtained shows that Kubernetes is not completely aging-resistant, in fact, the authors highlight the need for a faster and more accurate microservices fault detection system.

Other works compared the performance of Kubernetes with similar orchestration engines. More in detail, in [11], the authors compared Kubernetes with Docker Swarm,[3] Apache Mesos,[4] and Cattle.[5] Such a work performed different tests and the analysis highlighted that K8S is one of the most complete orchestrators and, at the same time, is one of the most complex. A similar work has been carried out in [12]. It compares Docker Compose, Docker Swarm, Fleet, Apache Mesos, and Kubernetes. Such engines have been tested showing that Kubernetes represents the best solution in large scale environments. In [13], the authors compared the performance obtained by stressing a Kubernetes cluster, a Docker Swarm cluster, and a plain Docker cluster.

The result obtained shows that Kubernetes recorded a higher overhead if compared with Docker Swarm and Docker; anyway, Kubernetes is preferable to Docker since it is able to more efficiently manage Docker containers cluster. Kubernetes also overcomes Docker Swarm because it provides an autoscaling functionality. Also in [14], the authors compared the feature provided by K8S, Docker Swarm, and OpenShift; the results show Kubernetes as the most feature-rich orchestrator.

One of the most particular Kubernetes features is replica autoscaling. This mechanism allows Kubernetes to automatically control the number of replicas of a specific resource, by increasing it when the system is overloaded, and by decreasing it when the system is overdimensioned for the workload. Moreover, autoscaling is useful to improve the scalability of the system and to prevent QoS degradation. Many researchers have analyzed the opportunities offered by the autoscaling feature, taking particular attention to the mechanism used to establish the replica threshold value. While most of the time the threshold value is fixed (and corresponds to the max amount of usable resources before the system could enter an overloaded state), such a mechanism may sometimes be limited. In fact, it could be more efficient to increase the number of replicas before the system reaches the overloaded state, as discussed in [15]. Hereby, the authors show how the custom-developed autoscaling controller is able to achieve better results in terms of the number of pods instantiated under different workloads compared to the default Kubernetes horizontal autoscaling controller. Similar solutions have been proposed in [16,17]. In the former, the authors proposed a new dynamic mode that automatically modifies the threshold values based on the system status. The results obtained showed an improvement in the system resource optimizations and QoS degradation. In the latter, the authors proposed the Resource Utilization Based Autoscaling System (RUBAS), a framework that overcomes the limitations of the already existing Kubernetes Vertical Pod Autoscaler (VPA) that requires killing the existing container and restart them with recalculated resources. The developed framework achieved better results than VPA, since it decreases the number of resource restart actions, the time required to perform a restart action, and the amount of used cluster resources.

Some other researchers have proposed different solutions based on time series data analysis [18] and others based on machine learning techniques [19,20]. Both solutions try to predict potential future cluster resources' peak usage and vertically or horizontally scale up the cluster object resources beforehand, based on cluster resources history data; in this way, the system can safely execute all workloads by granting availability and QoS. Moreover, the proposed solutions can scale down the cluster object replica number and resources to avoid waste of resources.

Since Kubernetes has demonstrated excellent performance and scaling capabilities, recently many researchers have analyzed its adoption in IoT environments. In [8], the authors show that Kubernetes can be a valid solution for building highly available and fault-tolerant infrastructure, by using the built-in Horizontal Pod Autoscaling (HPA) mechanism. Their results demonstrated how the HPA is able to solve the scalability issue. However, their simulations have also highlighted some limitations related to the cluster node replication process, which required up to 8 min to create a new complete working node. Moreover, the *MaxPods* configuration value can easily become the limit of the cluster by blocking the creation of new pod if it scales rapidly; in this case, the value must be manually increased.

In [6] the authors try to overcome the orchestration limitations of EdgeX Foundry,[6] an open-source container-based platform that facilitates interoperability at the IoT edge, by using Kubernetes technology. The simulations performed by the researchers have measured the performances obtained by EdgeX over Kubernetes. The results demonstrates that EdgeX over Kubernetes is able to achieve better throughput

[2] https://github.com/alessandrarizzardi/projects/secure-microservices.
[3] https://docs.docker.com/engine/swarm/.
[4] https://mesos.apache.org/documentation/latest/.
[5] https://github.com/rancher/cattle.
[6] https://www.edgexfoundry.org/.

and QoS when configured with the HPA mechanism if compared with EdgeX Foundry. In [7] the authors proposed a new Traffic-aware HPA (THPA), that provides dynamic resource autoscaling that also takes into account the network traffic of each node in an edge computing scenario. The proposed solution is able to up-scale or down-scale the cluster node pods proportionally to the real-time network traffic received by the nodes. The work continues with the building of a test system and the execution of some simulations, in order to evaluate the system's performance when using the new THPA mechanism and to compare them with the default HPA autoscaling mechanism. The results show that the new THPA is able to increase the QoS by scaling the pods of the nodes according to the network traffic distribution. Moreover, it is able to achieve better performance and throughput compared to the default HPA.

The solution, presented in this paper, adopts HPA to guarantee a flexible scaling mechanisms and the final aim is to evaluate the performance of a Kubernetes cluster running an attribute-based policy framework. The need to build it from scratch is driven by the fact that Kubernetes does not include an application-layer security mechanism; in fact, since Kubernetes is a cluster orchestrator, it can only implement network-level security policies.

An example is provided by the authors of [21], which analyzed the Kubernetes network security aspects and presented a new network security design based on the Zero Trust Model that, by default, considers all cluster traffic unsecured or untrusted [22]. The *Zero Trust Model* is implemented by using internal and external firewalls that verify all communication performed by the system. The performed simulation showed that the system is able to block unwanted internal and external traffic, by preventing attacks such as route hijacking and network scanning. Anyway, this work does not evaluate the overhead that the network traffic monitoring and logging can introduce. In [23] a similar work that aims to implement a *Zero Trust Model* in the Kubernetes ecosystem is envisioned. The Kubernetes cluster has been equipped with some add-ons such as: (i) Cilium, which provides load balancing and security; (ii) Istio, which introduces traffic encryption, micro-segmentation, and authorization policies enforcement; (iii) and Hubble,[7] which provide deep network traffic observability. The system has then been tested by deploying an application that performs data exchange among different pods; then, in order to verify that the intra-pods communication is encrypted, a traffic sniffer tool has been used. The results obtained by the authors show that the system is able to achieve the *Zero Trust network security Model*. In [24] the authors evaluated the performance of a Kubernetes cluster in which the Calico add-on[8] is installed to enforce network policies. The simulation aims to compare the cluster performance when enforcing the policies using Calico, with the performance without using Calico. The test system is composed of a maximum of 2000 policies, that check intra- and inter-node communication, and a maximum of 100 pods. The results obtained from the simulations demonstrated a low cluster overhead in terms of CPU, memory usage, and response time. The authors also highlighted some security aspects that a Kubernetes system must meet, such as configuring the policies in such a way that only the admitted connections are defined, avoiding policy misconfiguration by using a standard like Open Policy Agent (OPA)[9] and granting high isolation between microservices. In [25], the authors analyzed 104 internet artifacts deriving a set of best practices concerning security aspects. From the carried out analysis, the authors extracted 11 best practices that a cluster should meet, which can be summarized as following:(i) forcing a cluster-level authentication and authorization mechanism that prevent unknown users to access the Kubernetes API server; (ii) implementing Kubernetes network security policies to restrict intra-pod traffic; (iii) performing cluster-level vulnerability scanning; (iv) enabling application and cluster logging; (v) introducing namespace separation to improve the cluster components isolation; (vi) forcing access restriction and encryption mechanism for the Kubernetes etcd database (i.e., a local database used to store cluster information); (vii) introducing continuous updates for all the used components and technologies; (viii) enabling SSL/TLS for cluster level communications; (ix) forcing CPU and memory quota limit to prevent attacked pods to drain all cluster hardware resources; (x) running sensitive workloads in separated machines in order to improve system isolation; (xi) forcing secure metadata access to prevent sensitive information leak.

The authors in [26] proposed a cluster-level Intrusion Detection System (IDS), able to detect unauthorized accesses, named Snort.[10] Such a work aims to increase the overall system security by installing the IDS at the node level; in this way all the communications performed in the cluster can be intercepted and analyzed; anyway, it must be noted that if the cluster is composed of many nodes, Snort must be installed on each node and must be maintained separately. Once the system has been initialized with the required components and Snort has been configured with correct network policies, a port scan attack is executed in order to verify the introduced overhead and the correct policy enforcement. The outcome show that the introduced overhead is reasonable, without impacting the cluster performance, and the IDS system correctly detects the malicious packets.

The security impact of excessive permission attacks in real production environments is investigated in [27]. To mitigate this kind of risk, proper prevention mechanisms must be put in place, as we try to do with the solution presented in this paper. In fact, besides removing unnecessary permissions, the existing ones must be properly enforced. The work in [28] presents the use of attribute-based encryption to provide cloud resources with suitable confidentiality. A proof of concept has been conceived to validate the approach, while we go beyond the encryption to set up attribute-based policies.

Hence, in this paper, unlike the existing approach, an application-level attribute-based security policies mechanism is designed and developed, considering a smart home as a target scenario. The aim is to protect the resources managed inside the network architecture, which are acquired from IoT devices. The proposed solution exploits a Kubernetes cluster, which is composed of: (i) three separated databases, where the smart home dataset, the users, and the policies are, respectively, stored; (ii) a replicated Pod, running node server instances that implement the core logic and perform the policy enforcement; (iii) a monitoring system composed of Grafana and Prometheus used to collect and analyze the cluster node performance. Differently from the other solutions, this work aims also to analyze the cluster node performance with different workloads considering an application-level security mechanism running on a Kubernetes cluster, instead of evaluating the performance obtained by cluster-level security mechanisms.

3. Background on Kubernetes and involved technologies

Kubernetes is defined as an orchestrator, a component able to control, manipulate and manage a set of entities, named Pod, within a distributed system. By using this framework it is possible to define the structure and components of a cluster, in addition to intra-cluster interactions and communications.[11] The main services offered by Kubernetes are: load balancing, storage system, nodes' orchestration, roll-out management, and workload specification for each Pod.

When an administrator performs a deployment of some workload, Kubernetes create and allocate the required resources to the cluster execution. This results in the creation of two separated elements: (i)

[7] https://github.com/cilium/hubble.

[8] https://docs.tigera.io/calico/3.25/getting-started/kubernetes/managed-public-cloud/eks#install-eks-with-amazon-vpc-networking.

[9] https://www.openpolicyagent.org/.

[10] https://www.snort.org/.

[11] https://kubernetes.io/.

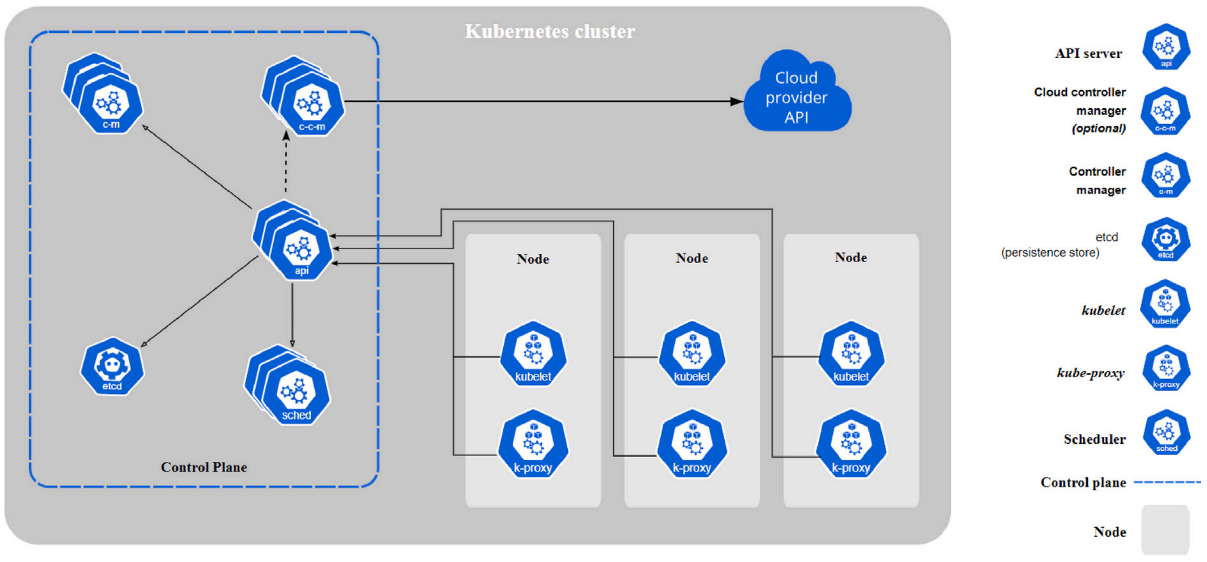

Fig. 1. The components of a Kubernetes cluster.

Control Plane Node that handles the *Worker Nodes*, which most of the times are replicated in order to grant the availability and service continuity properties to the system; (ii) *Worker Nodes* that execute the workloads defined by the developer. Fig. 1[12] sketches the architecture of a Kubernetes cluster.

3.1. Minikube

In order to run a Kubernetes cluster, it is necessary to set up an environment able to execute it. There are different ways and services to run the Kubernetes cluster, but to be also able to locally run the cluster (i.e., not only remotely), the choice of the execution environment is limited to $microk8s$, $k3s$ and $Minikube$. All of them are lightweight versions of Kubernetes, able to run a Kubernetes cluster locally. Since such three solutions are similar, the choice fell on the well-known Minikube [29]. Minikube can be defined as a lightweight Kubernetes distribution that can be executed on all operating systems, it is simple to configure and by default works with a single node configuration. Since it is not a production-ready solution, it is not equipped with all the Kubernetes functionalities; this is necessary to keep Minikube as lightweight as possible, anyway, to overcome this limitation, it is simply possible to add the required plugins.[13] Hence, in order to make Minikube platform-independent, it is executed on a virtualization layer that can be chosen between *VirtualBox, VMware, Docker*, and others. In this work, Minikube has been executed as a container with Docker.

3.2. Monitoring system

A key aspect of Minikube is that it provides a monitoring system by default. However, it is limited to specific metrics related to the cluster healthy and pod healthy, but it does not provide a detailed view of the resources used by the node running into the cluster. In order to obtain a more detailed view, Prometheus[14] and Grafana[15] have been installed into the Minikube node. Prometheus is a log system that collects information from Minikube nodes, stores them temporarily, and organizes them in order to perform efficient queries. It also provides a basic web page to visualize and query all the collected information. Although it does not provide a complex dashboard, due to the fact that querying and collecting all the information separately is a time-consuming activity, Grafana has been added as the primary dashboard. Grafana is a log visualization system that allows the user to perform powerful queries, aggregations, and data manipulations. It is equipped with so many predefined dashboards and it is perfectly integrated with Prometheus. The installation and integration of these two tools have not been done manually, instead, it has been done automatically with Helm,[16] a tool used to manage Kubernetes packages.

3.3. Storage

Since the work involves the use of a persistent storage to store the dataset and the information about interacting users, MongoDB[17] has been adopted. MongoDB is an open document-oriented database that stores heterogeneous data in JSON format. This means that the information within a document can be very different to another document. Moreover, it allows us to create different collections that, in the analyzed scenario, have proven to be really helpful. In addition, since all the code is written in JavaScript, the integration of the data structures used with MongoDB does not need adaptation or manipulation, thereby reducing the effort required to develop the application. The combination of JavaScript and MongoDB has also reduced the query creation process, improving the reusability of complex and generic queries. However, the most important reason for which this type of database has been chosen is that the structure of the dataset used is heterogeneous; therefore, a NoSQL database is more suitable than an object-relational one.

3.4. Node

A core component of the envisioned approach is represented by the Node, which is often named Node server. Node[18] is an open-source, single-thread, cross-platform runtime environment that is able to run JavaScript code. It has been used with the Express package that allows a Node process to listen, reply and make HTTPS requests. Node server has been chosen in order to adopt a single programming language for the whole solution, because it has been used for all the components of the cluster.

[12] https://kubernetes.io/docs/concepts/overview/components/.
[13] https://minikube.sigs.k8s.io/docs/.
[14] https://prometheus.io/.
[15] https://grafana.com/.
[16] https://helm.sh/docs.
[17] https://www.mongodb.com/.
[18] https://nodejs.org/it/docs/.

4. Proposed solution

In this section, the system architecture, the analyzed scenario and the proposed attribute-based policy management framework will be detailed. Note that we make use of a running case study in order to better explain the envisioned approach, thus avoiding to be too generic.

4.1. System architecture

The developed system is shown in Fig. 2 and it is composed of the following entities:

- Three *PersistentVolumes*. Since the cluster has been designed to handle all data sources separately, three $PersistentVolume$ have been created, and are used by the MongoDB instances running on the cluster to store users, authorizations and dataset information, respectively
- Two *Jobs*, in charge of loading, respectively, the list of users interacting with the system, and the data gathered from the dataset (i.e., available in csv format)
- Two *Statefulset*, since the cluster has been designed to keep the dataset separated from users and authorizations, three instances of MongoDB have been deployed using the so-called Statefulset objects. They introduce a state within pods, where each pod has its own network address, persistent storage, and a pod name, which cannot be changed. Moreover, to keep the MongoDB configuration as simple as possible, both implementations share the same *Secret* and *ConfigMap* objects. The first is used to provide confidential data stored in base64 format. The ConfigMap, in contrast, supplies the container storage path where MongoDB stores data and a JavaScript script used to create the master database and the master user by reading Secret records. Both $user - statefulset$ and $datadb - statefulset$ components are configured to run a single, not replicated MongoDB version 4.0.8 container. Both require a $Service$ object to be accessible from other cluster elements.
- One *Deployment*, which exposes an API endpoint that can be used by users to access stored data. It is the only component able to read data from both $Statefulset$ elements, it is replicated, and it is responsible for checking user authorization and returning correct data. Like the previously explained $Statefulsets$, it requires a $Service$ object in order to be reached by users.

All such components and microservices make up the cluster executed with Minikube. Note that all of them are defined to be executed in a custom namespace; this helps to group them together and isolate them from the Kubernetes default pods. Moreover, deleting the namespace will remove all the related resources, with the exception of volumes, which require dedicated deletion actions.

4.2. Application scenario

The application scenario is based on a csv dataset[19] composed of a massive number of samples related to six homes, respectively *homeA, homeB, homeC, homeD, homeE,* and $homeF$. Each home has its own dataset that represents the installed and monitored sensors. The sampling interval can vary from one to fifteen seconds depending on the sensor set; while the home monitoring period starts from one to three years, respectively from 2014 to 2016. Moreover, this dataset is highly heterogeneous since each home is composed of a different sensor set; for such a reason, a NoSQL database (i.e., MongoDB) revealed to be the best solution. An example of adapted document is shown in listing 1.

[19] https://traces.cs.umass.edu/index.php/Smart/Smart.

```
{
  'dateTime': '2016-12-31T23:00:00.000Z',
  'usedKw': 0,
  'generateKw': 0,
  'furnaceHRVKw': 0.195337778,
  'cellarOutletsKw': 0.083204444,
  'washingMachineKw': 0.005686111,
  'fridgeRangeKw': 0.006891667,
  'disposalDishwasherKw': 0.005568889,
  'kitchenLightsKw': 0.012153889,
  'bedroomOutletsKw': 0.020451667,
  'bedroomLightsKw': 0.004899444,
  'masterOutletsKw': 0,
  'masterLightsKw': 0.010393333,
  'ductHeaterHRVKw': 0.382681667,
  'electricRangeKw': 0.002882778,
  'dryerKw': 0.005231111,
  'garageMudroomLightsKw': 0.013155,
  'diningRoomOutletsKw': 0,
  'mudroomOutletsKw': 0,
  'masterBathOutletsKw': 0,
  'garageOutletsKw': 0,
  'basementOutdoorOutletsKw': 0,
  'kitchenDenLightsKw': 0,
  'masterBedBathLightsKw': 0,
  'denOutdoorLightsKw': 0,
  'denOutletsKw': 0,
  'rearBasementLightsKw': 0,
  'kitchenOutletsEastKw': 0,
  'kitchenOutletsSouthKw': 0,
  'dishwasherDisposalSinkLightKw': 0,
  'refrigeratorKw': 0,
  'microwaveKw': 0,
  'officeLightsKw': 0
}
```

Listing 1: Dataset record example

The dataset records are accessible to all the authorized users based on the corresponding authorizations, while the users are simulated using a dedicated program in order to perform a series of HTTPS requests. The information required by a user is served by a central, replicated node server that enforces the authorizations bound to each user. A key aspect of the scenario defined concerns the roles. They have been structured in such a manner that each one is specifically related to a home and specify the readable fields and optional time permission. It is worth to remark that the user time authorization defines the range from which the user has lived in a home, while the time period specified in a role defines the daily time range for which a user can enter a home and read related dataset records; note that the only role that has daily time restrictions is $CleaningCompanyEmployee$. During the role-designing phase, some critical aspects have been addressed. The most complex one is related to dataset heterogeneity. In fact, there exist some roles that must be able to read different fields belonging to different homes. In order to solve such a limitation, the roles have been defined in such a manner that they are uniquely identified by the field pair <*home, relatedRole*>; in this way, it is possible to separately define the readable fields related to a role for a specific home. Then, the resulting roles are the following:

- *HomeOwner*: is the owner of the home and has full access to recorded information
- *Tenant*: represents the user that lives in a home for a limited time period and has the same authorization set of the HomeOwner role
- *UnderageResident*: is supposed to be an underage user like a child, that has a limited authorization set
- *CleaningCompanyEmployee*: represents a cleaner that has limited authorizations in terms of readable fields and access time window. In fact, he/she can read records that match the daily time restriction defined

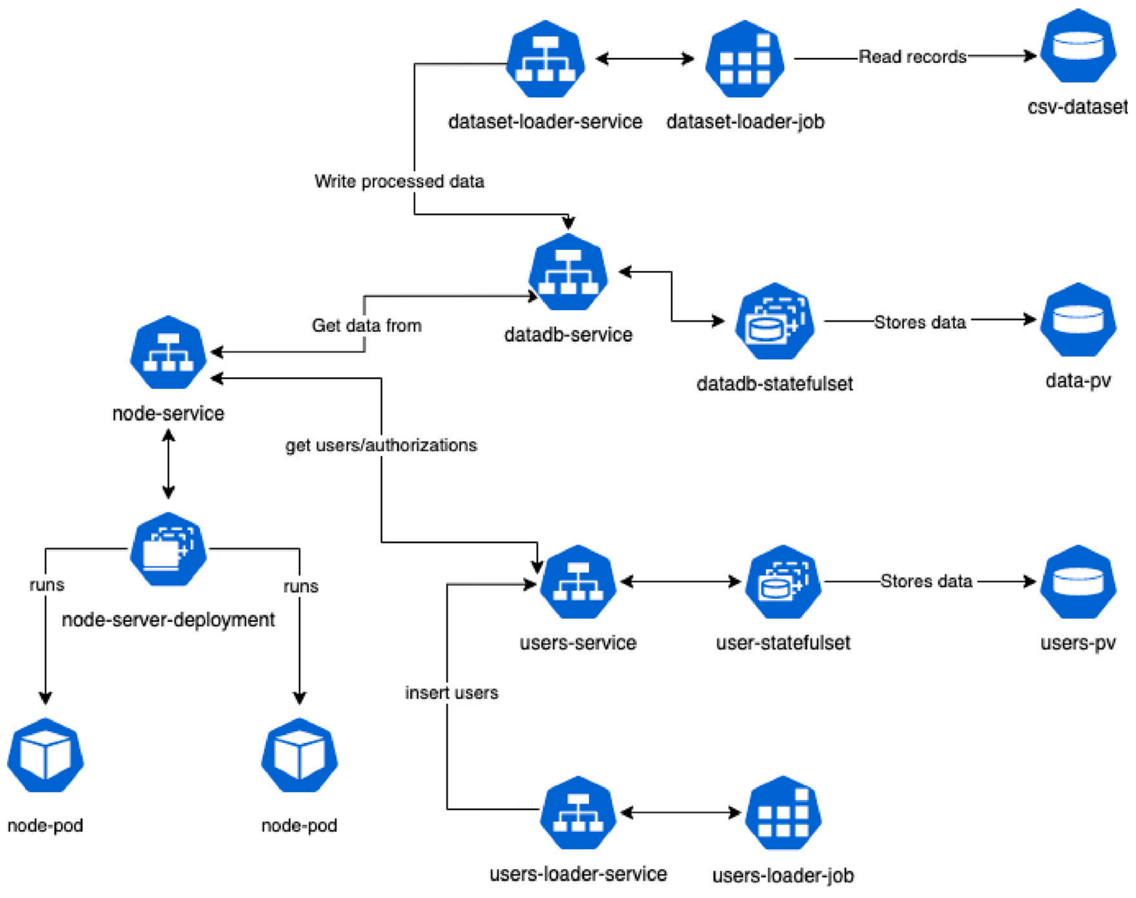

Fig. 2. System architecture.

- *ResidenceOwner*: is the owner of the residence and has limited access permission to each home record; in fact, he/she can see only aggregated information.

In listing 2 an example of stored authorization document is shown.

```
{
  'relatedRole': 'ResidenceOwner',
  'home': 'homeA',
  'fields': [
    'kitchenOutletsEastKw',
    'kitchenOutletsSouthKw',
    'refrigeratorKw',
    'kitchenDenLightsKw',
    'kitchenLightsKw',
    'fridgeRangeKw',
    'furnaceHRVKw',
    'microwaveKw',
    'dishwasherDisposalSinkLightKw',
    'disposalDishwasherKw',
    'dryerKw',
    'washingMachineKw'
  ]
}
```

Listing 2: Role authorizations example

The last fundamental entity is represented by the users. A user includes a set of information, such as:

- *userId*: a uniquely identifier field used to identify a single user. This field's value must be passed as query parameter every time a user performs an HTTPS request to retrieve the user information and related authorizations
- *home*: is an array field containing the homes for which the user has read authorizations
- *role*: is used to specify the role assigned to a user
- *lengthOfStay*: this field is used to define the time period for which a user has lived into the related home. It must be noted that it is an object containing the starting date that is mandatory, and an end date that can be optional. Both dates are required to retrieve the dataset information; in fact, a user cannot read dataset records collected in a time period not included in that defined for the user.

The users have been divided into two logical groups based on the role played: (i) home-related roles; (ii) residence-related roles. The first group includes all the users that can retrieve dataset records related to a specific home, having access permission to all the home API routes; the roles included are: $HomeOwner$, $Tenant$ and $UnderageResident$. The last group includes the roles that have access permissions to more than one home, but with different limitations; in fact, the $ResidenceOwner$ can retrieve only aggregated information from all houses, while the $CleaningCompanyEmployee$ can read a particularly reduced set of fields from homes.

The scenario has been designed with seventeen users organized as follow: one $ResidenceOwner$, three $CleaningCompanyEmployee$ that have access to two different homes, respectively, two $HomeOwner$ related to $homeA$, one $HomeOwner$ related to $homeB$, two $HomeOwner$ and one $UnderageResident$ related to $homeC$, two $Tenant$ related to $homeD$, one $Tenant$ related to $homeE$ and, in conclusion, one $Tenant$, two $HomeOwner$ and one $UnderageResident$ related to $homeF$. All of them are then simulated with a python program that performs

Table 1
Resulting policies.

Role	Home	Authorized API	Fields access
ResidenceOwner	homeA	Consumption API	Restricted access
ResidenceOwner	homeB	Consumption API	Restricted access
ResidenceOwner	homeC	Consumption API	Restricted access
ResidenceOwner	homeD	Consumption API	Restricted access
ResidenceOwner	homeE	Consumption API	Restricted access
ResidenceOwner	homeF	Consumption API	Restricted access
HomeOwner	homeA	Sensors API	Full access
HomeOwner	homeB	Sensors API	Full access
HomeOwner	homeC	Sensors API	Full access
HomeOwner	homeD	Sensors API	Full access
HomeOwner	homeE	Sensors API	Full access
HomeOwner	homeF	Sensors API	Full access
UnderageResident	homeA	Sensors API	Restricted access
UnderageResident	homeB	Sensors API	Restricted access
UnderageResident	homeC	Sensors API	Restricted access
UnderageResident	homeD	Sensors API	Restricted access
UnderageResident	homeE	Sensors API	Restricted access
UnderageResident	homeF	Sensors API	Restricted access
Tenant	homeA	Sensors API	Full access
Tenant	homeB	Sensors API	Full access
Tenant	homeC	Sensors API	Full access
Tenant	homeD	Sensors API	Full access
Tenant	homeE	Sensors API	Full access
Tenant	homeF	Sensors API	Full access
CleaningCompanyEmployee	homeA	Sensors API	Restricted access
CleaningCompanyEmployee	homeB	Sensors API	Restricted access
CleaningCompanyEmployee	homeC	Sensors API	Restricted access
CleaningCompanyEmployee	homeD	Sensors API	Restricted access
CleaningCompanyEmployee	homeE	Sensors API	Restricted access
CleaningCompanyEmployee	homeF	Sensors API	Restricted access

pseudo-random HTTPS requests. Listing 3 shows an example of user entity.

```
{
  'userId': 'homeOwnerA',
  'home': [
    'homeA'
  ],
  'role': 'HomeOwner',
  'lengthOfStay': {
    'from': '2014-01-01T00:00:00.000Z',
    'to': null
  }
}
```

Listing 3: User entity example

Finally, the developed scenario provides a set of API that allow the users to retrieve information based on the user's authorizations. The developed API has been divided into two groups: *sensors API* and *consumption API*. The first group is used to fetch home specific records; on the other side, the consumption API are used to retrieve aggregated records related to all homes in the residence.

4.3. Attribute-based policies

Once the role and the user entities have been designed, and a list of roles has been identified, the policies can be also defined. The policies are built starting from the user and role parameters, in particular *role type*, *role home*, *access fields*, and *related API group*. In Table 1 an example of list of resulting policies is summarized.

When a user performs a request, the system will return a list of authorized information if and only if the request matches one of the resulting policies defined in Table 1. Moreover, each policy determines the *fields access permission*, which can be *Full access*, that allows the user to read all information related to a home without any restrictions, or *Restricted access*, that allows the user to read only a portion of information based on the role *authorized fields* and *accessTimePermission* parameters. All the requests performed by a user that does not match any of the policies will return a not authorized response.

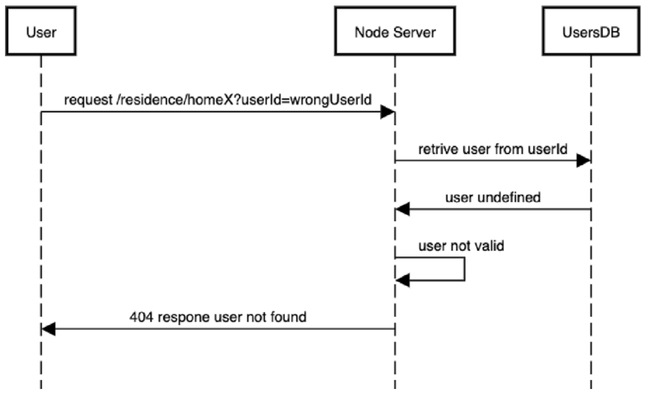

Fig. 3. Invalid user.

The implementation of the core system logic relies on an Express web server written in JavaScript and executed in a node container. Such a container is then replicated by a Kubernetes Deployment object and receives all the HTTPS requests. The consumption API exposed by the web server are:

- */residence*: it can be used to retrieve the last ten records produced for each home
- */residence/kitchens*: returns the max and average value of all the kitchen-related sensors of each home
- */residence/laundry*: returns the max and average value of all the laundry-related sensors of each home
- */residence/power*: returns the max and average value of all the energy consumption-related sensors of each home.

It is worth to remark that the consumption API can be accessed only by a user playing the $ResidenceOwner$ role. On the other side, the sensors API can be accessed by all roles different from the $ResidenceOwner$, and the available routes are:

- */residence/:home* : it can be used to retrieve the last records produced, considering the user's authorizations.
- */residence/:home/kitchen*: returns the last records of all the kitchen-related sensors, based on the role played.
- */residence/:home/laundry*: returns the last records of all the laundry-related sensors, based on the role played.

The developed routes implement distinct types of checks to enforce defined authorizations. The checks performed can be divided into three levels: (i) first level check that ensures the *userId* supplied is valid and related to an existing user or not; (ii) second level check that ensures the user is authorized to access the requested route; (iii) third level check that applies the authorizations obtained. Since each route requires retrieving the user information from the database, the first level check is used to verify if the provided query parameter *userId* corresponds to the same field of a user document stored; if so, the user instance will be retrieved and the execution will continue normally, otherwise, a 404-HTTPS not found response will be sent. Note that a more detailed feedback about the error can be replied to the requesting users. For the sake of simplicity, we just mention the 404 response. The explained verification flow has been implemented by using the Express middleware feature. In Fig. 3, the verification flow is sketched.

Once the user has been confirmed, a role-related check will be performed in order to ensure the user is authorized to access the routes. The two checks have been implemented by introducing the control

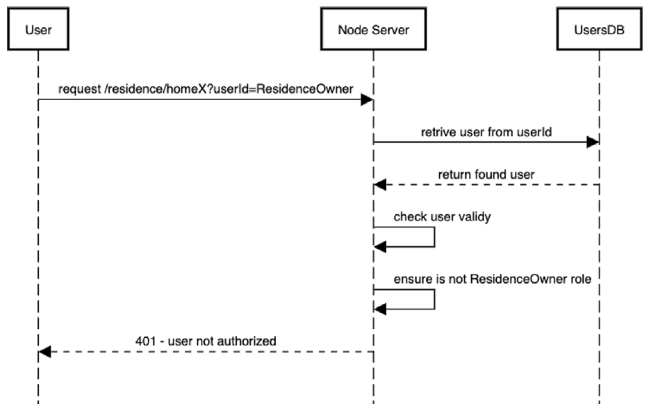
User
Node Server
UsersDB
request /residence/homeX?userId=ResidenceOwner
retrive user from userId
return found user
check user validy
ensure is not ResidenceOwner role
401 - user not authorized

Fig. 4. ResidenceOwner not authorized.

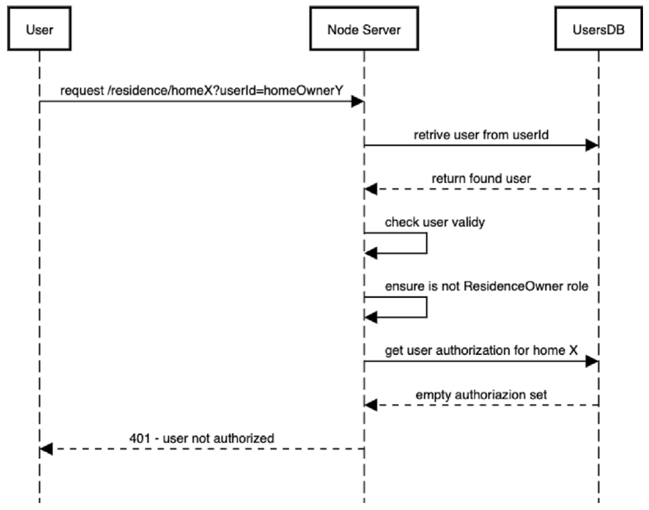
User
Node Server
UsersDB
request /residence/homeX?userId=homeOwnerY
retrive user from userId
return found user
check user validy
ensure is not ResidenceOwner role
get user authorization for home X
empty authoriazion set
401 - user not authorized

Fig. 5. User with empty authorizations.

logic before the authorizations retrieve step; the consumption check is performed at the service level, while the sensor check has been implemented at the Express middleware level. It must be noted that the base sensor API route implements a restriction that allows only the $HomeOwner$ to access it. All the checks return a 401-HTTPS response if failed. Fig. 4 shows this kind of check.

Before proceeding with the authorization fetching step, an intermediary check is performed in the sensors API. In fact, since the *home* for which a user can retrieve the information are the six previously listed, the system must be sure that the requested home exists. To do that, the node server checks that the home specified in the route is valid; if the check passes the request will continue its execution, otherwise a 404-HTTPS response will be returned (along with the detail of the problem).

Once the supplied *userId* is valid and the related user is authorized to access the routes, the authorization fetching step is performed. In order to retrieve the correct list of fields in the defined time period, user authorizations are required. To retrieve them, a query to the authorization collection specifying the *userId* is submitted. It must be noted that a similar query is submitted to the user's collection to retrieve the user's information. Since the authorizations are not required to verify the first level check, the two queries have been performed at different times and only when strictly required. Once the *authorizations* list has been obtained, the system has to filter it and keep only the authorizations set related to the required home; if the resulting *authorizations* set is empty means that the user is trying to retrieve home information for which he/she is not authorized. If so, a 404-HTTPS response will be returned to the user (along with the detail of the related issue); in Fig. 5 the described flow is reported.

The last step consists of the query build and execution phase. This step returns a list of dataset records. It must be noted that all the filter and aggregation phases are included in the built query and performed by the MongoDB instance; for this reason, no other manipulation steps are required.

The query build process is the intermediate step required to enforce the role *authorizations*. This step is responsible for extracting all the required information from the user and role objects, creating the required query, and returning the related data. The user and role objects are composed of a list of parameters that are useful for building the query; such parameters are *authorizations fields*, *lengthOfStay*, and *accessTimePermission*. The *authorizations* must know which fields are readable by a specific user; since each route is responsible for returning a specific family of sensors set, these two sets must intersect. The resulting set of the intersection operation is a list of fields that will be used to perform the query. The second parameter required, which is the *lengthOfStay* field, is necessary to retrieve only the dataset records that have been collected in the time period included between the specified dates. This information must be calculated at each request and strictly depends on the user's information. Moreover, since returning all the records collected between the specified dates is too expensive for the system in terms of bandwidth, CPU, and memory resources, the routes have been developed to consider only the last month available, which should be compatible with the user ending date and the last record collected. The resulting data range will be then used to build the query. The JavaScript code that calculates the time range is shown in listing 4.

The last parameter required to complete the build query process is the *accessTimePermission*. It is an optional parameter related to the role object that defines the daily access time slot. Since it is optional, when not defined, the system will skip the related step; otherwise, it will be included in the query build process. Once the three parameters have been obtained, the query can be built and submitted to the store. The query submission is reported in listing 5.

In conclusion, the database will return a list of MongoDB documents containing only the authorized fields matching the user authorizations, that have been collected in a valid date range, and that match the user timeslot permission, if defined. The node server will then return the information obtained to the user in the HTTPS body response field. In the sequence diagram reported in Fig. 6 a 200-HTTPS response is shown.

4.4. Security in service-to-service communications

Obviously, requests and responses taking place within the network are performed among different (i.e., separated) microservices. Besides the deployment stays on the same cluster, it is still important to protect the message passing, since microservices could not be considered trusted (i.e. we consider a *Zero Trust Model*). Adopting HTTPS protocol ensure channel protection, and this can be considered a best practice in microservices communications.

Instead, at the application level, we should guarantee: (i) the identity of the sender; (ii) the correctness of the receiver (i.e., information are really sent to the right recipient); (iii) the confidentiality and integrity of the information transmitted. To achieve such a goal, each microservice has its own public/private key pair, and the private key is used to sign the *JSON Web Token (JWT)* [30]. Hence, once deployed, each microservice, during authentication, generates a certificate. Afterward, each microservice will use the certificate from the other to authenticate itself. Note that certificates must be also kept secure by a proper authority, which inevitably represents a further entity that impacts on the network performance.

```
export function getUserDates(user, home) {
  const lengthOfStay = {
    from: user.lengthOfStay?.from,
    to: user.lengthOfStay?.to,
  };
  let lastValidDate = home == 'homeD' ?
                 lastHomeDDate : lastCommonDate;
  return getDates(lengthOfStay, lastValidDate);
}

function getDates(
    { from = undefined, to = undefined } = {},
    lastValidDate
) {
  if (from) {
    return
      {
        fromDate: from,
        toDate: getFollowingThirtyDays(from)
      };
  } else if (to) {
    return
      {
        fromDate: getPastThirtyDays(toDate),
        toDate: to
      };
  } else {
    return
      {
        fromDate: getPastThirtyDays(lastValidDate),
        toDate: lastValidDate
      };
  }
}
```

Listing 4: User dates manipulation code

```
async function getKitchenConsumption(
  collectionName,
  authFields,
  fromDate,
  toDate,
  accessFrom,
  accessTo
) {
  const searchFields = buildSearchParams(
    authFields,
    getKitchenParams,
    collectionName
  );

  return await onDataDB((db) =>
    db
      .collection(collectionName)
      .aggregate(buildFilterParams(
              accessFrom,
              accessTo,
              fromDate,
              toDate))
      .project(searchFields)
      .sort({ dateTime: DESC })
      .limit(100)
      .toArray()
  );
}
```

Listing 5: Query submission step

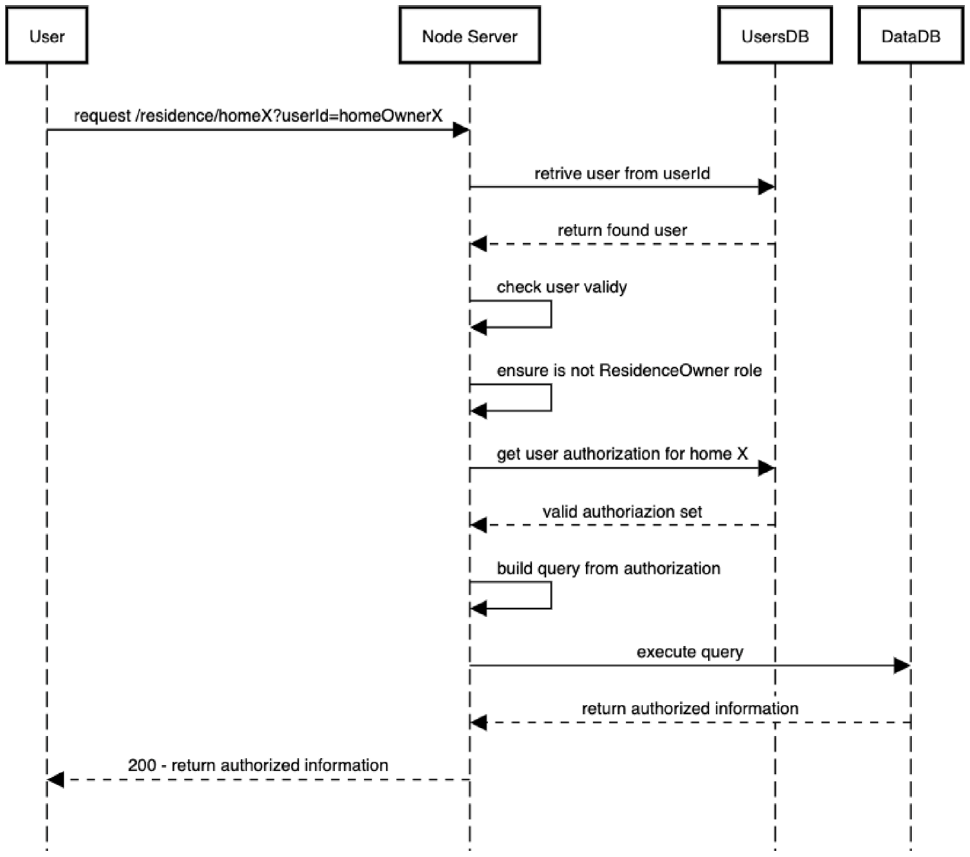
User
Node Server
UsersDB
DataDB
request /residence/homeX?userId=homeOwnerX
retrive user from userId
return found user
check user validy
ensure is not ResidenceOwner role
get user authorization for home X
valid authoriazion set
build query from authorization
execute query
return authorized information
200 - return authorized information

Fig. 6. Authorized user.

Furthermore, also the cluster configuration should be kept secret, in order to prevent possible attacks towards the network infrastructure. Cluster configuration can be protected by encrypting the source code related to the cluster setup. We do not performed this operation in this work, in order to leave the code publicly available, and avoiding to also release encryption/decryption functionalities integrated into the code with a public default password. However, even in case of cluster configuration protection, the secure management of the various settings is in charge of the stakeholder(s) of the systems (and this depends on the application domain).

Finally, following the principle of "monitor everything", cluster administrators should determine how to constantly and automatically monitor microservices-based applications for potential threats. To this end, monitoring tools, such as Prometheus (which is adopted in this paper), can bolster centralized monitoring capabilities to identify possible Denial of Services (DoS) attacks. As a countermeasure, a rate limiting can be integrated. Rate limiting is a concept that ensures an application accepts and processes a maximum of n requests within a specified duration of time. This helps ward off an attacker that, for example, tries to break into an application with many different combinations of credentials. Hence, rate limiting is an excellent way to prevent DoS attacks and credentials from being compromised. Another important features we applied to our system is the isolation of microservices. In fact, each service has been developed, tested, deployed, scaled and maintained independent of all other ones. Such an isolation extends to the databases as well. Typically, each microservice has its own copy of the data. The isolation ensures that if one microservice is compromised, it cannot access the data of another microservice. Another benefit is that in the event of a failure, the failed microservice will not bring down the other microservices in the application.

5. Performance evaluation

The experimental phase requires simulating the user's behavior and monitoring the system performance by varying the total requests' number in a certain time window and the available resources. A python script is responsible for executing an independent thread for each interacting user. Each thread then performs pseudo-random HTTPS requests to the node server running and returns an array composed of the request time of each performed request. Moreover, in order to simulate both failing and successful cases, each HTTPS request has an 80% probability to be a successful request and 20% to be a failure request (i.e., the requesting users have no permission to access the requested information). Since the script must be able to perform a different amount of requests in a different amount of time, it has been developed to be configurable simply by tuning a few parameters, like: (i) *number of requests per minute*; (ii) *execution time in minutes*. Note that the total number of requests must be multiplied by the number of users in order to get the total requests submitted to the system at each simulation round.

As explained in Section 3, the system includes a monitoring system named Grafana. This tool shows the hardware metrics of the running node, in particular, the *CPU usage percentage*, the *CPU saturation percentage*, the *memory used*, the *kilobyte sent*, the *disk usage percentage*, and the *disk saturation percentage*. Such metrics, combined with the *request time* collected by the user simulation script, are used to evaluate the system performance. In Figs. 7 and 8, a preview of the Grafana console used to collect metrics is shown.

Performance evaluation has been carried out in a Docker virtual environment running on a local virtual machine equipped with 2 cores and 4 GB of RAM; such resources are used by the whole Kubernetes cluster. Obtained results are listed in Table 2.

The first row shows the system performance "at rest", by executing a single request. This result will be used as comparison metrics. It is

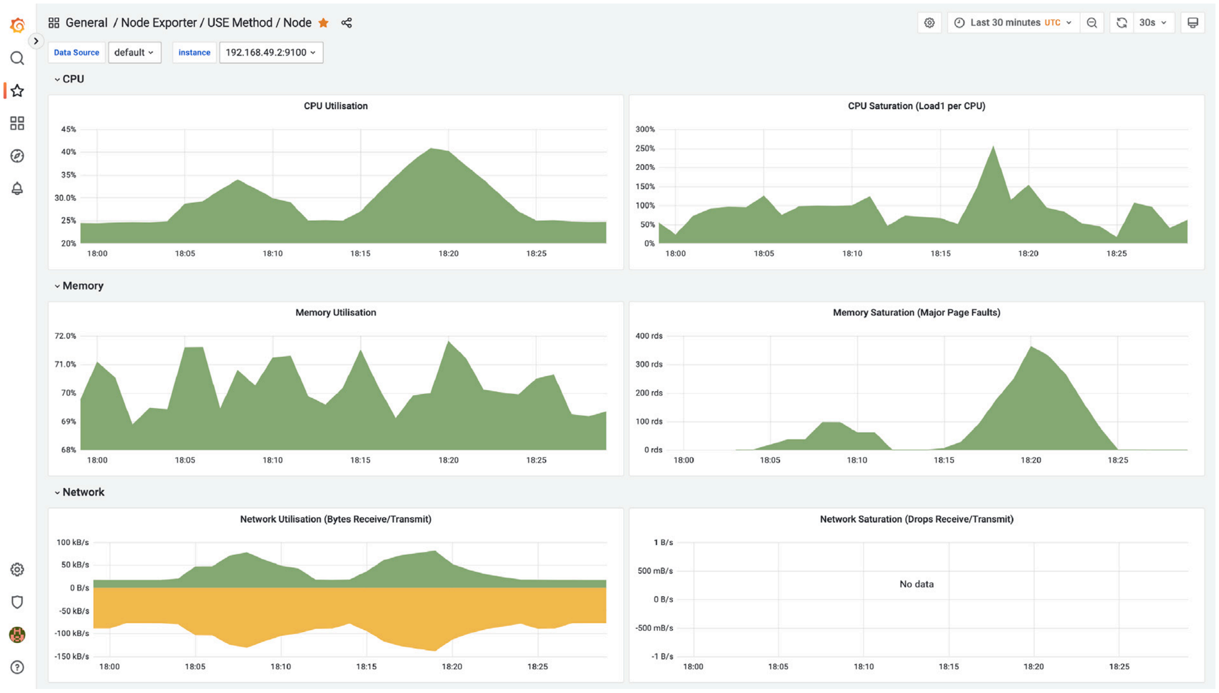
General / Node Exporter / USE Method / Node
Data Source default
instance 192.168.49.2:9100
Last 30 minutes UTC
30s
CPU
CPU Utilisation
CPU Saturation (Load1 per CPU)
Memory
Memory Utilisation
Memory Saturation (Major Page Faults)
Network
Network Utilisation (Bytes Receive/Transmit)
Network Saturation (Drops Receive/Transmit)
No data

Fig. 7. Grafana CPU and memory console.

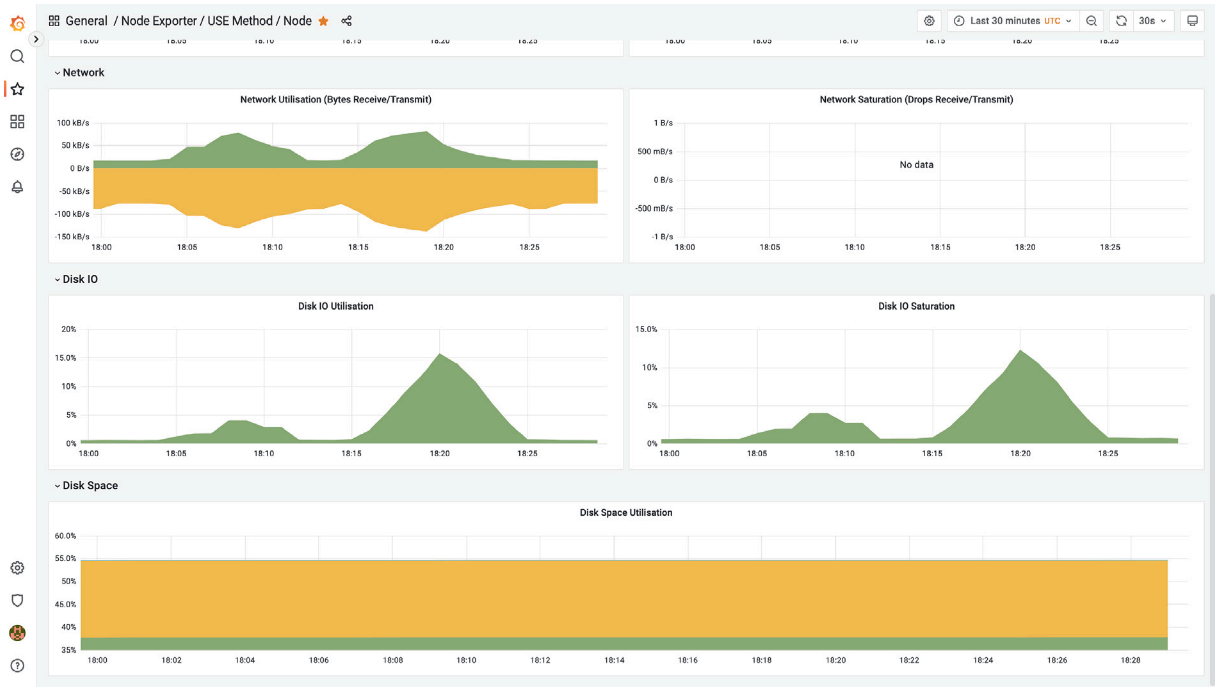
General / Node Exporter / USE Method / Node
Last 30 minutes UTC
30s
Network
Network Utilisation (Bytes Receive/Transmit)
Network Saturation (Drops Receive/Transmit)
No data
Disk IO
Disk IO Utilisation
Disk IO Saturation
Disk Space
Disk Space Utilisation

Fig. 8. Grafana network and disk console.

important to highlight that the *response time* is 29 ms. This value is useful because it allows determining if the system performance scales with the number of requests. The first simulation has been performed with 10 requests per minute for 10 min for each user, for a total of 1600 requests. This simulation, like the previous one, shows a stable system with an increase in CPU usage and kilobytes sent. The requests time has increased too, anyway it is still acceptable considering that all requests are successful. In all four simulations executed, the response time grew between the 70th and 80th requests; this peak can be justified by the system overloading caused by the increasing number of requests enqueued by the system. The response time related to 4 users of 4 different simulations are reported in Fig. 9.

The second simulation was performed with 12 requests per minute for 10 min for each user, by executing 1920 requests. Even in this case, the simulations performed show a stable system with an increase in the CPU and memory used. Fig. 10 shows the response time related to 4 users of 4 different simulations.

The third group of simulations has been performed with 13 requests per minute for 10 min per user, for a total of 2080 requests submitted to the system. Such parameters mark the threshold from which the system

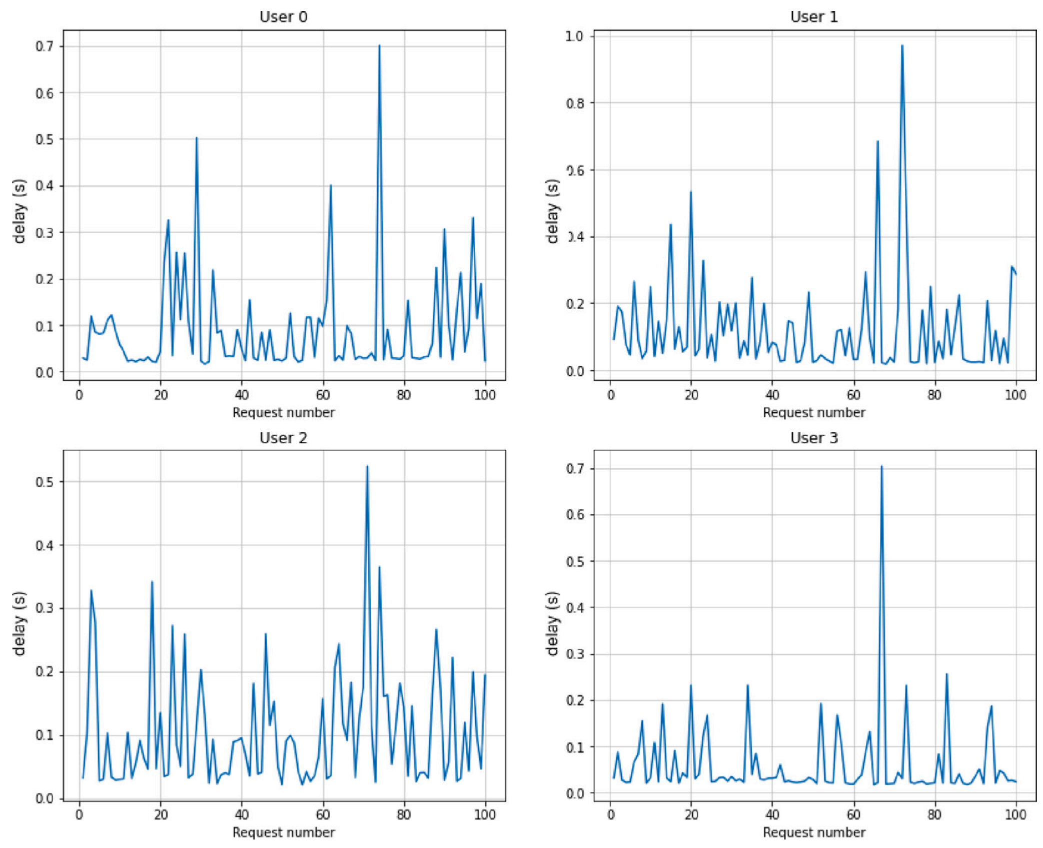

Fig. 9. Response time - 10 requests in 10 min of 4 different simulations.

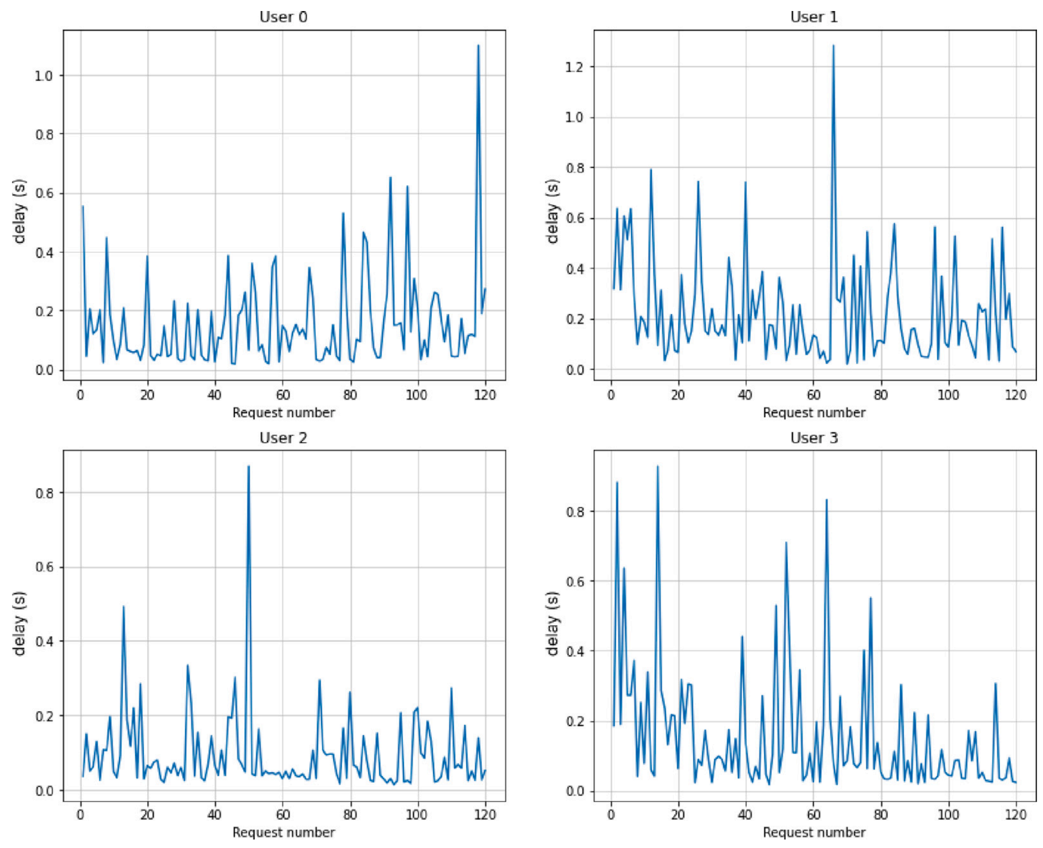

Fig. 10. Response time - 12 requests in 10 min of 4 different simulations.

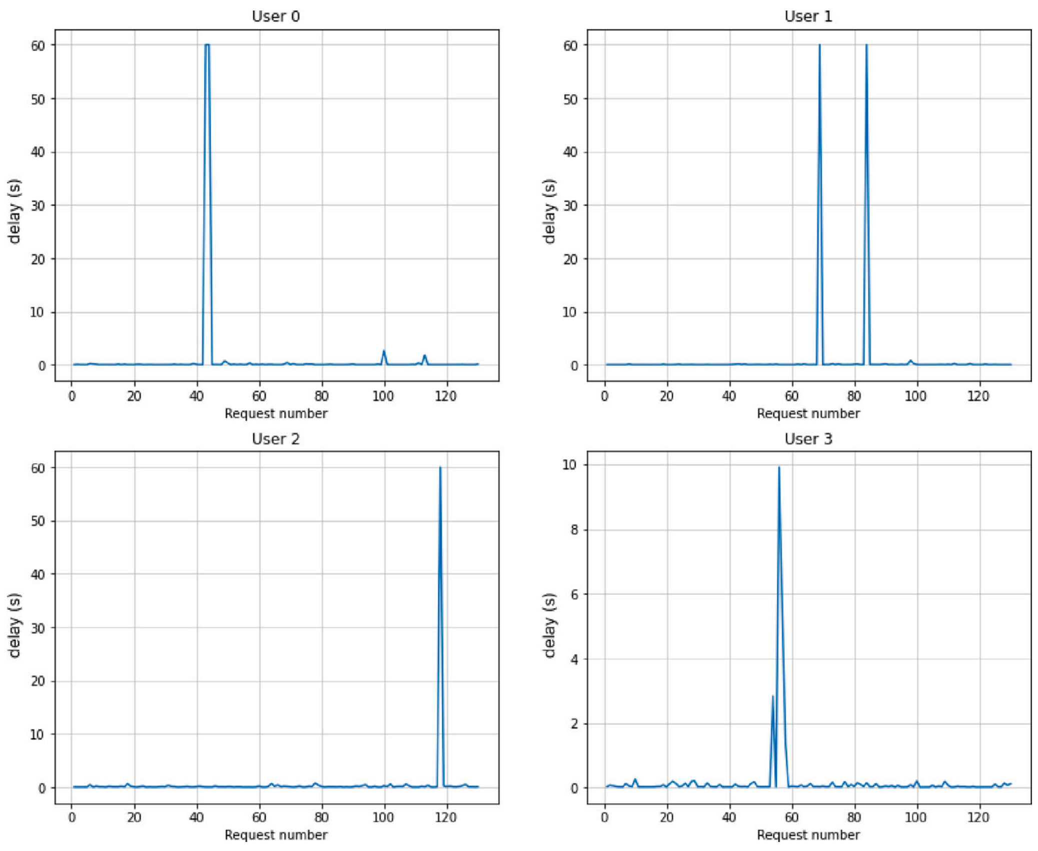

Fig. 11. Response time - 13 requests in 10 min of 4 different simulations.

Table 2
Simulation results - single-node cluster.

Request/Minute	Minutes	CPU	RAM	KB sent	Disk
1	1	21.9%	64.1%	91 kbps	0.55%
10	10	27.3%	64.7%	141 kbps	0.72%
12	10	29.0%	65.0%	142 kbps	1.0%
13	10	30.4%	66.2%	144 kbps	1.46%
15	10	38.1%	65.7%	126 kbps	21.0%
20	10	44.9%	65.8%	139 kbps	23.8%

Table 3
Simulation results - multi-node cluster.

Request/Minute	Minutes	CPU	RAM	KB sent	Disk
1	1	16.4%	48.7%	96 kbps	0.56%
10	10	20.5%	48.6%	152 kbps	0.77%
12	10	21.8%	48.2%	151 kbps	1.56%
13	10	23.3%	49.5%	154 kbps	1.89%
15	10	28.6%	49.8%	159 kbps	23.6%
20	10	34.2%	50.1%	164 kbps	26.7%

starts to be unstable; in fact, even if the recorded system performances are similar to the previous simulations, the number of requests that failed is between 9 and 45. As can be seen in Fig. 11, all the simulations present peaks highlighting the unhandled requests.

The fourth simulation has been performed with 15 requests per minute for 10 min per user, by submitting a total of 2400 requests in 10 min. In this case, like the previous one, the system was not able to handle all HTTPS requests, by counting a minimum of 495 and a maximum of 911 unhandled requests. It must be noted that an increase of 320 requests (15.35%) compared to the previous simulation, has led to an increase in failed responses from 45 (2.13%) to 911 (37.96%) in the worst scenario. Results are presented in Fig. 12.

The last simulation has been performed with 20 requests per 10 min per user, by executing a total of 3200 requests. As expected, this test case produced the worst system metrics, with the highest number of failed HTTPS requests: 2201 (68.78%). It is interesting to note that disk usage at its peak is about 23.8%; for this reason, all the system limitations can be addressed to the node server constraints. In fact, the deployment object has been designed with strict limits on the CPU quota and memory quota. Fig. 13 shows the request time graphs, highlighting the massive number of failed requests.

In light of the obtained results, to counteract the failure rate for requests, we conducted further simulations adding a node to the cluster. More in detail, since Minikube does not allow by default to manage multi-node clusters, we added the *CSI Hostpath Driver* addon, in order to deploy a second node, which brings two more cores to the simulation environment (i.e., now the system includes 4 cores). Minikube autonomously orchestrates the load balance between nodes, putting an enhanced virtual machine on the field. Hence, the two nodes cooperate to satisfy the requests by users, but must also manage the concurrent access to certain resources (e.g., database instances); note that an overhead is introduced to handle the synchronization for balancing the load among the nodes themselves. However, as revealed by the results shown in Fig. 14, such an overhead is acceptable with respect to the overall system performance. In fact, the distribution of the delays generated by the single and multi-node cluster configurations and the results obtained in Table 3, for the same metrics and the same hardware reported in Table 2, show that a better load balance can be achieved using a larger cluster, while at the same time changing the network conditions (i.e., by varying the requests per minute, we simulate different network conditions concerning network traffic).

5.1. Discussion

Experimental results have been just described, paying particular attention to the cluster's hardware performance, the system's response

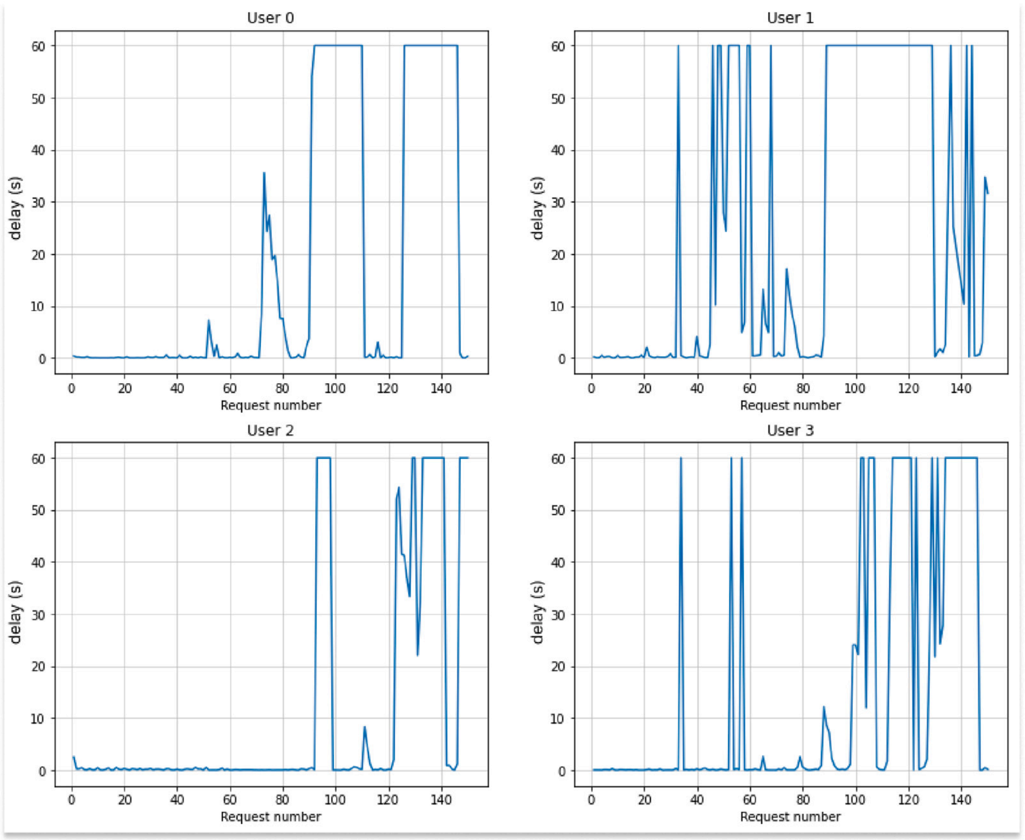

Fig. 12. Response time - 15 requests in 10 min of 4 different simulations.

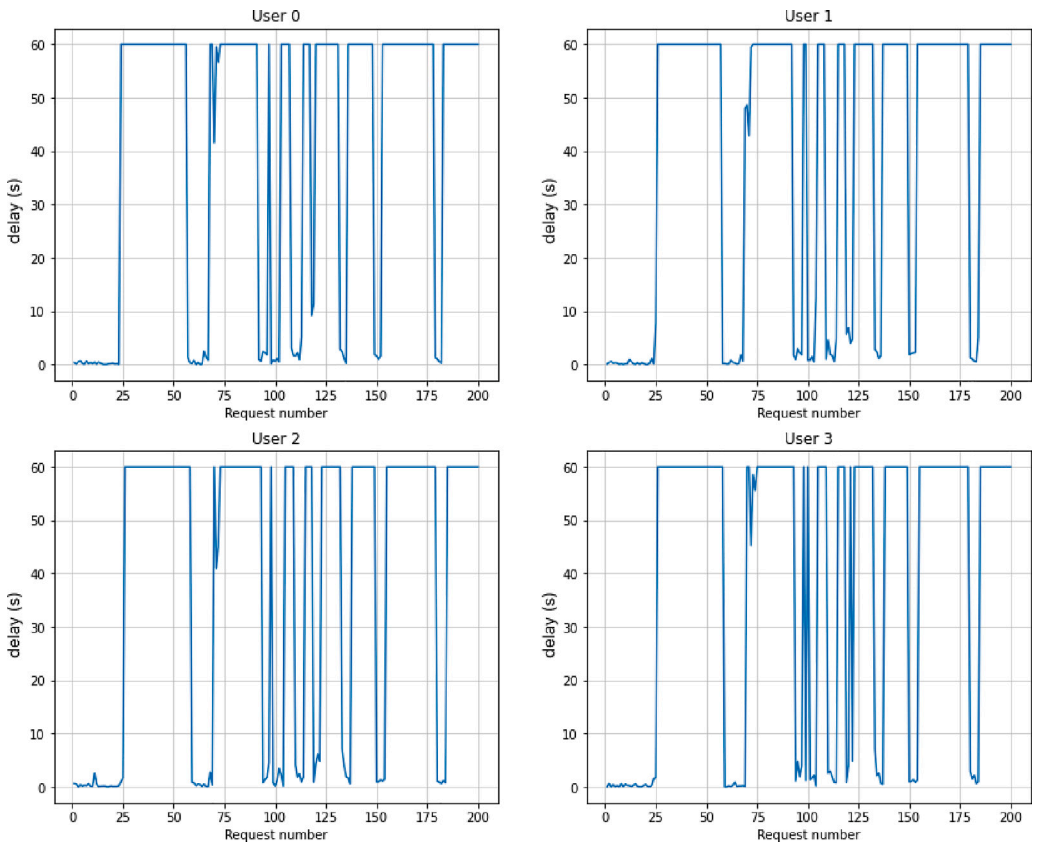

Fig. 13. Response time - 20 requests in 10 min of 4 different simulations.

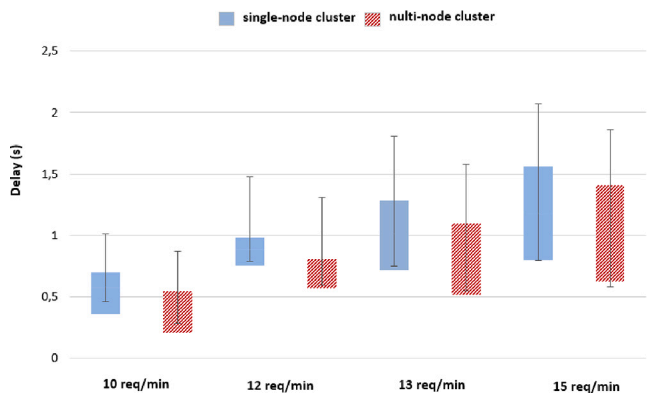

Fig. 14. Whiskers-box diagram comparison between the simulation with single- and multi-node cluster.

time, and the number of unhandled requests with respect to the total request number. The most important outcome revealed by simulations is that, without the further node addition, the system works correctly if the number of requests is about 200 per minute. While, as soon as the number of requests exceeds the threshold of 200 requests per minute, the number of unhandled requests start growing. Adding a second node to the system, we obtained a better load balance of the system, thus reducing the number of unhandled requests.

To further overcome the system's limitations, other few improvements can be made, as explained hereby. The first solution consists of increasing the node hardware resources dedicated to the cluster. Such a proposal is not always applicable and could be expensive in a real scenario. Moreover, it is not always possible to increase the resources of a physical machine. The second improvement consists in modifying the Kubernetes components resource and replication criteria. As resulting from the experiments, even if the Node server Deployment is configured to run two containers, it still acts as the bottleneck within the network. Such an issue can be partially solved by increasing the number of replicas or, since the running Node server containers are configured to use at most 25% CPU and 64 MB of RAM, by increasing those values. However, such a change must be made considering that all the newly allocated resources to the node server containers are potentially stolen from the others cluster components. A third solution is related to the inner organization of the cluster components. In fact, while the MongoDB instances are correctly balanced in terms of resource constraints, the Node server has many different tasks' responsibilities, like retrieving user information, evaluating user checks, building the queries, and retrieving the information required; ideally, such responsibilities can be divided into two groups, respectively *user responsibility* and *data responsibility*. Such two groups can be then executed in two different Kubernetes components; in this way, it is possible to increase the resource quota limits and the replicas of the group that performs the heaviest tasks, optimizing the node resource distribution.

Note that all the presented improvements can be applied together. Certainly, the first solution should be applied to a Kubernetes system that is correctly balanced in terms of resource distribution and components responsibilities, but that still suffers the physical machine resources limitations. The secondary solution should be applied to those Kubernetes systems that have many physical resources, that are not correctly distributed among all the cluster components. Instead, the last solution suits those Kubernetes systems that have many physical resources, which are correctly distributed among the cluster components, but some components are responsible for many heavy tasks.

Finally, fault tolerance and system recovery mechanisms should be taken into account in potential highly dynamic or failure-prone environments. In order to preliminary assess the capability of the proposed system to recover from fault conditions, such as a node disruption,

Table 4
Simulation results - monolithic approach.

Request/Minute	Minutes	CPU	RAM	KB sent	Disk
1	1	21.9%	63.9%	88 kbps	0.53%
10	10	28.2%	64.7%	132 kbps	0.81%
12	10	29.85%	68.1%	144 kbps	1.49%
13	10	31.08%	72.5%	148 kbps	1.69%
15	10	45.56%	76.9%	179 kbps	38.63%
20	10	59.3%	86.1%	202 kbps	47.4%

we inserted into the code of the two nodes some random instructions to stop the node's functionalities (i.e., in order to simulate a real disruption). This allows to assess if the system continues to work in case of node failure, autonomously redistributing the tasks transparently to the user, and, also, it allows to test the recovery system. In fact, to cope with potential failure-prone environments, on the one hand, we configured Docker in order to be able to manage the restart after a failure of the container itself. On the other hand, we equipped Docker with the *healthcheck* functionality, which allows to assess at runtime the status of the nodes (i.e., the application running in the Docker container is functioning correctly or not). More in detail, the *healthcheck* reports the health of a running container based on the application it hosts. A Docker *healthcheck* operates by executing a command within the container at regular intervals. If the command executes successfully, the container is considered healthy. If it fails, the container is flagged as unhealthy. Hence, it is possible to can keep track of the operational status of containerized applications, thus improving availability and reliability; moreover, it helps in early problem detection, failover, scalability, and auditing. In case of failover, the system is configured to force the restart of the node. Obviously a slight overhead is introduced with the integration of the *healthcheck* mechanism. Note that the rate for checking the status of the nodes should be set accordingly to the requirements (e.g., the level of fault tolerance and the failure probability) of the application domain and to the configuration of the application running inside container (e.g., the number of existing replicas). In the simulations reported in this paper, we set the rate for the *healthcheck* to 1 min.

5.2. Comparison

The proposed microservices-based solution must be compared with a monolithic one, in order to appreciate the obtained efficiency gain. To this end, the work presented in [31] is considered hereby. It proposes a flexible security enforcement framework, coherently integrated within a distributed IoT architecture, based, as well, on attribute-based policies. The same case study and simulations parameters are taken into account. Table 4 shows the results obtained for the same metrics and the same hardware reported in Table 2. The difference is minimal with a low number of requests/minute, but increases with the increment in the number of requests per minute. Hence, we can conclude that the microservices-based scales better than the monolithic one.

A similar result is achieved considering the comparison of the distribution of the delays generated by the two approaches, as reported in Fig. 15. Note that, to make the graph readable, peaks in delays (as shown in Figs. 9 to 12) are omitted.

6. Conclusions

In this paper, a microservices-based cluster system has been developed by using the well-known Kubernetes technology. The system has been developed to replicate a residence scenario, composed of six different homes. The work also included the design and implementation of an organization composed of many roles played by a certain number of users. Authorizations are managed as well as they are strictly related to roles. Once the scenario has been defined, the microservices system

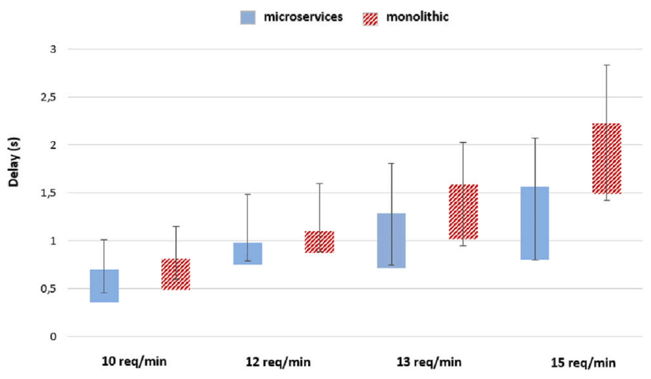

Fig. 15. Whiskers-box diagram of delay comparison: microservices vs. monolithic approach.

has been implemented by using different Kubernetes objects, like Statefulsets and Deployments to implement the core logic and the MongoDB instances, Services to create the network organization, ConfigMaps, and Secrets to store and inject cluster parameters, and Jobs to implement the users and dataset upload tasks. Then, the cluster was executed by using the Minikube technology, which runs an Ubuntu container into a Docker execution environment. Then, Grafana and Node server access points were exposed to perform and show the simulations. Comparison with a monolithic approach has been also provided.

As a future research direction, the system performance could be tested with increased node Deployment resource limits and with different replica numbers, paying special attention to the MongoDB instances. Moreover, it could be interesting to re-design the cluster structure by creating different logical groups, each one related to a specific home and composed of a single MongoDB instance containing only the information related to the specific home, with a non-replicated node server that implements only the logic required to build the query and retrieve the information. Another future work could consist in verifying the system performance in a real environment executed on a remote cluster. We will also evaluate the current Node server Deployment object by introducing a clever Kubernetes autoscaling mechanism [32] in large scale environments.

Finally, comparison with other microservices frameworks could allow to identify the better choice accordingly to the requirements of the application domain. With this respect, some existing works investigated the features and performance of several container orchestration frameworks. The authors of [33] conduct a feature comparison study of the three most prominent orchestration frameworks, which are *Docker Swarm*, *Kubernetes*, and *Mesos*, revealing that: (i) Docker Swarm is expected to be used and customized for specific technology segments, such as the IoT domain, due to its low memory footprint, its support for co-existing virtual networks and its support for run-time updates of container images without the need to restart containers; (ii) Kubernetes provides the highest number of functionalities (e.g., autoscaling, replicas' management, orchestration engine, clusters' maintenance); (iii) Mesos is the best choice for large-scale cluster deployments (i.e., in terms of number of deployed nodes). Specifically related to the IoT context, the work in [34] compares *Docker Swarm* and *Kubernetes* with wired and wireless communication and different configurations for the clusters. The analysis demonstrates that, on one hand, Docker Swarm outperforms Kubernetes, both in use of resources and processing time, but, on the other hand, Kubernetes includes a feature set useful for higher performance devices (i.e., servers) that are not included natively in Docker Swarm, such as auto-scaling and the initial test to perform a default load balancing. The same outcome has been reached in [13], which evaluates the overheads of the same two container orchestration tools and identify their pros and cons, by means of a number of benchmarking exercises. The results show that the overall performance of Kubernetes is slightly worse than that of Docker in Swarm mode. However, Docker in Swarm mode is not as flexible or powerful as Kubernetes in more complex situations. A growing interest is on the container orchestration tools in edge/fog computing environments [35], which aim to replicate the advantageous traits posed by cloud (i.e., virtualization of workloads), moving them closer to the local action. The most relevant challenge in this field lies in the orchestration of workloads considering the heterogeneity of computing equipment characterizing edge scenarios. A strong research line is the creation of lightweight versions of Kubernetes, like $K3s$ and $MicroK8s$. Another approach is also gaining popularity, which adapts the Kubernetes architecture to the edge requirements instead of just reducing Kubernetes components' size and footprint. The most promising solution in this direction is $KubeEdge$. The authors of [36], in a comparative experiment on a real edge deployment, compare Kubernetes, K3s and KubeEdge. What emerged from the analysis is that Kubernetes presents good performance even in a resource-constrained edge environment, disproving the assumption that only its distributions (K3s and KubeEdge) are more lightweight and suitable for edge computing nodes. However, K3s and KubeEdge solve the main problem of Kubernetes at the edge, namely by supporting nodes in private networks, albeit with some performance degradation.

CRediT authorship contribution statement

Alessandra Rizzardi: Writing – original draft, Validation, Methodology, Funding acquisition, Conceptualization. **Sabrina Sicari:** Writing – original draft, Supervision, Conceptualization. **Alberto Coen-Porisini:** Supervision, Funding acquisition, Conceptualization.

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.

Acknowledgments

This work was supported in part by the SERENA-IIoT project, which has been funded by MUR (*Ministero dell'Universita' e della Ricerca*), Italy under the PRIN 2022 program (project code 2022CN4EBH), and in part by project SERICS (project code PE00000014), under the NRRP MUR program funded by the EU - NGEU, Italy.

Data availability

No data was used for the research described in the article.

References

[1] Francisco Ponce, Gastón Márquez, Hernán Astudillo, Migrating from monolithic architecture to microservices: A Rapid Review, in: 2019 38th International Conference of the Chilean Computer Science Society, SCCC, IEEE, 2019 pp. 1–7.

[2] Nichlas Bjørndal, Antonio Bucchiarone, Manuel Mazzara, Nicola Dragoni, Schahram Dustdar, Fondazione Bruno Kessler, T. Wien, Migration from Monolith to Microservices: Benchmarking a Case Study, Technical report, Tech. Rep., 2020, [Online]. Available: https://www.researchgate.net/profile.

[3] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, Stefan Wagner, From monolith to microservices: A classification of refactoring approaches, in: Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment: First International Workshop, DEVOPS 2018, Chateau de Villebrumier, France, March 5-6, 2018, Revised Selected Papers 1, Springer, 2019, pp. 128–141.

[4] Alan Megargel, Venky Shankararaman, David K. Walker, Migrating from monoliths to cloud-based microservices: A banking industry example, in: Software Engineering in the Era of Cloud Computing, Springer, 2020, pp. 85–108.

[5] Abdul Razzaq, Shahbaz A.K. Ghayyur, A systematic mapping study: The new age of software architecture from monolithic to microservice architecture - awareness and challenges, Comput. Appl. Eng. Educ. 31 (2) (2023) 421–451.
[6] Seunghwan Lee, Linh-An Phan, Dae-Heon Park, Sehan Kim, Taehong Kim, Edgex over kubernetes: Enabling container orchestration in edgex, Appl. Sci. 12 (1) (2021) 140.
[7] Linh-An Phan, Taehong Kim, et al., Traffic-aware horizontal pod autoscaler in Kubernetes-based edge computing infrastructure, IEEE Access 10 (2022) 18966–18977.
[8] Brandon Thurgood, Ruth G. Lennon, Cloud computing with Kubernetes cluster elastic scaling, in: Proceedings of the 3rd International Conference on Future Networks and Distributed Systems, 2019, pp. 1–7.
[9] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, Ferhat Khendek, Kubernetes as an availability manager for microservice applications, 2019, arXiv preprint arXiv:1901.04946.
[10] José Flora, Paulo Gonçalves, Miguel Teixeira, Nuno Antunes, A study on the aging and fault tolerance of microservices in Kubernetes, IEEE Access (2022).
[11] Isam Mashhour Al Jawarneh, Paolo Bellavista, Filippo Bosi, Luca Foschini, Giuseppe Martuscelli, Rebecca Montanari, Amedeo Palopoli, Container orchestration engines: A thorough functional and performance comparison, in: ICC 2019-2019 IEEE International Conference on Communications, ICC, IEEE, 2019, pp. 1–6.
[12] Lubos Mercl, Jakub Pavlik, The comparison of container orchestrators, in: Third International Congress on Information and Communication Technology: ICICT 2018, London, Springer, 2019, pp. 677–685.
[13] Yao Pan, Ian Chen, Francisco Brasileiro, Glenn Jayaputera, Richard Sinnott, A performance comparison of cloud-based container orchestration tools, in: 2019 IEEE International Conference on Big Knowledge, ICBK, IEEE, 2019, pp. 191–198.
[14] Marek Moravcik, Martin Kontsek, Overview of Docker container orchestration tools, in: 2020 18th International Conference on Emerging eLearning Technologies and Applications, ICETA, IEEE, 2020, pp. 475–480.
[15] Neha Deshpande, Autoscaling Cloud-Native Applications using Custom Controller of Kubernetes (Ph.D. thesis), National College of Ireland, Dublin, 2021.
[16] Salman Taherizadeh, Vlado Stankovski, Dynamic multi-level auto-scaling rules for containerized applications, Comput. J. 62 (2) (2018) 174–197.
[17] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, Devesh Tiwari, Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes, in: 2019 IEEE 12th International Conference on Cloud Computing, CLOUD, IEEE, 2019, pp. 33–40.
[18] Valter Rogério Messias, Julio Cezar Estrella, Ricardo Ehlers, Marcos José Santana, Regina Carlucci Santana, Stephan Reiff-Marganiec, Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure, Neural Comput. Appl. 27 (2016) 2383–2406.
[19] Mahmoud Imdoukh, Imtiaz Ahmad, Mohammad Gh. Alfailakawi, Machine learning-based auto-scaling for containerized applications, Neural Comput. Appl. 32 (2020) 9745–9760.
[20] Nhat-Minh Dang-Quang, Myungsik Yoo, Deep learning-based autoscaling using bidirectional long short-term memory for kubernetes, Appl. Sci. 11 (9) (2021) 3835.
[21] Nico Surantha, Felix Ivan, Secure kubernetes networking design based on zero trust model: A case study of financial service enterprise in indonesia, in: Innovative Mobile and Internet Services in Ubiquitous Computing: Proceedings of the 13th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS-2019, Springer, 2020, pp. 348–361.
[22] John Kindervag, S. Balaouras, et al., No more chewy centers: Introducing the zero trust model of information security, Forrester Res. 3 (2010).
[23] Catherine de Weever, Marios Andreou, Zero Trust Network Security Model in Containerized Environments, University of Amsterdam, Amsterdam, the Netherlands, 2020.
[24] Gerald Budigiri, Christoph Baumann, Jan Tobias Mühlberg, Eddy Truyen, Wouter Joosen, Network policies in kubernetes: Performance evaluation and security analysis, in: 2021 Joint European Conference on Networks and Communications & 6G Summit, EuCNC/6G Summit, IEEE, 2021, pp. 407–412.
[25] Md. Shazibul Islam Shamim, Farzana Ahamed Bhuiyan, Akond Rahman, Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices, in: 2020 IEEE Secure Development, SecDev, IEEE, 2020, pp. 58–64.
[26] Pavel Hristov, Designing an Intrusion Detection System for a Kubernetes Cluster (B.S. thesis), University of Twente, 2022.
[27] Nanzi Yang, Wenbo Shen, Jinku Li, Xunqi Liu, Xin Guo, Jianfeng Ma, Take over the whole cluster: Attacking kubernetes via excessive permissions of third-party applications, in: ACM SIGSAC Conference on Computer and Communications Security, 2023, pp. 3048–3062.
[28] Mauro Femminella, Martina Palmucci, Gianluca Reali, Mattia Rengo, Attribute-based management of secure Kubernetes cloud bursting, IEEE Open J. Commun. Soc. 5 (2024) 1276–1298.
[29] Sebastian Böhm, Guido Wirtz, Profiling lightweight container platforms: MicroK8s and K3s in comparison to Kubernetes, in: ZEUS, 2021, pp. 65–73.
[30] Salman Ahmed, Qamar Mahmood, An authentication based scheme for applications using JSON web token, in: 2019 22nd International Multitopic Conference, INMIC, IEEE, 2019, pp. 1–6.
[31] Sabrina Sicari, Alessandra Rizzardi, Daniele Miorandi, Cinzia Cappiello, Alberto Coen-Porisini, Security policy enforcement for networked smart objects, Comput. Netw. 108 (2016) 133–147.
[32] Abeer Abdel Khaleq, Ilkyeun Ra, Intelligent autoscaling of microservices in the cloud for real-time applications, IEEE Access 9 (2021) 35464–35476.
[33] Eddy Truyen, Dimitri Van Landuyt, Davy Preuveneers, Bert Lagaisse, Wouter Joosen, A comprehensive feature comparison study of open-source container orchestration frameworks, Appl. Sci. 9 (5) (2019) 931.
[34] Rafael Fayos-Jordan, Santiago Felici-Castell, Jaume Segura-Garcia, Jesus Lopez-Ballester, Maximo Cobos, Performance comparison of container orchestration platforms with low cost devices in the fog, assisting Internet of Things applications, J. Netw. Comput. Appl. 169 (2020) 102788.
[35] Rafael Vaño, Ignacio Lacalle, Piotr Sowiński, Raúl S-Julián, Carlos E. Palau, Cloud-native workload orchestration at the edge: A deployment review and future directions, Sensors 23 (4) (2023) 2215.
[36] Ivan Čilić, Petar Krivić, Ivana Podnar Žarko, Mario Kušek, Performance evaluation of container orchestration tools in edge computing environments, Sensors 23 (8) (2023) 4008.