Project acronym:	**PrEstoCloud**
Project full name:	**Proactive Cloud Resources Management at the Edge for efficient Real-Time Big Data Processing**
Grant agreement number:	**732339**

D3.7 Mobile Offloading Processing Microservice - Iteration 1

Deliverable Editor:	Salman Taherizadeh (JSI & CVS)
Other contributors:	Marko Grobelnik (JSI), Blaz Novak (JSI), Marija Komatar (CVS), Sebastjan Vagaja (CVS)
Deliverable Reviewers:	Noam Amram (LiveU) and Marija Komatar (CVS)
Deliverable due date:	30.6.2018
Submission date:	29.6.2018
Distribution level:	Public
Version:	1.0

This document is part of a research project funded by the Horizon 2020 Framework Programme of the European Union

This document is part of a research project funded by the Horizon 2020 Framework Programme of the European Union

Change Log

Version	Date	Amended by	Changes
0.0	01/3/2018	Salman Taherizadeh (JSI & CVS), Marko Grobelnik, Blaz Novak (JSI), Marija Komatar (CVS).	Table of Contents
0.1	01/4/2018	Salman Taherizadeh (JSI & CVS), Marko Grobelnik, Blaz Novak (JSI), Marija Komatar (CVS).	Draft version
0.2	01/5/2018	Salman Taherizadeh (JSI & CVS), Marko Grobelnik, Blaz Novak (JSI), Marija Komatar (CVS).	Draft version
0.3	15/5/2018	Salman Taherizadeh (JSI & CVS), Marko Grobelnik, Blaz Novak (JSI), Marija Komatar (CVS).	Draft version
0.4	01/6/2018	Salman Taherizadeh (JSI & CVS), Marko Grobelnik, Blaz Novak (JSI), Marija Komatar (CVS).	Draft version
0.5	15/6/2018	Salman Taherizadeh (JSI & CVS), Marko Grobelnik, Blaz Novak (JSI), Marija Komatar (CVS).	Draft version
0.6	20/6/2018	Salman Taherizadeh (JSI & CVS), Marko Grobelnik, Blaz Novak (JSI), Marija Komatar (CVS), Sebastjan Vagaja (CVS).	Draft version ready for internal review
0.7	21/6/2018	Marija Komatar (CVS)	Internal review
0.8	22/6/2018	Noam Amram (LiveU)	Internal review
0.9	27/6/2018	Salman Taherizadeh (JSI & CVS), Marko Grobelnik, Blaz Novak (JSI), Marija Komatar (CVS), Sebastjan Vagaja (CVS).	Addressing internal reviewers' comments
1.0	28/6/2018	Project consortium	Final internal approval

Table of Contents

List of Figures

List of Tables

Abbreviations

The following table presents the acronyms which are used in this deliverable.

Abbreviation	*Description*
API	Application Programming Interface
CPU	Central Processing Unit
ECS	EC2 Container Service
G	Gravitational-force
GB	Gigabyte
GCE	Google Container Engine
IoT	Internet of Things
JSON	JavaScript Object Notation
OS	Operating System
REST	Representational State Transfer
SOA	Service Oriented Architecture
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	Virtual Machine

1. Executive summary

Edge computing, which is currently a hot research topic is a widespread method for providing many different types of Big Data technologies such as Internet of Things (IoT) systems. Edge computing is a new computing paradigm optimising applications to expand their data processing capabilities next to end-users rather than at the faraway centralised datacentres. Such an advanced computing framework provides a highly distributed, heterogeneous environment which can exploit the lightweight container-based virtualisation technology such as Docker for the fast, automatic deployment of services.

However, edge devices such as Raspberry Pi [1] suffer from their hardware resource restrictions for example they have a limited amount of storage capacity or computation power. In essence, the hardware resource limitation of edge nodes is one of major challenges in achieving improvements provided by the edge computing architecture. Therefore, edge devices may fail to react to the changing environment (e.g. increasing workload) at the right time that can lead to a massive software engineering failure for applications orchestrated within edge computing platforms.

To overcome to this problem, the PrEstoCloud solution [2] proposes two important software components: (i) Edge On/Offloading Server and (ii) On/Offloading Client. In order to handle such situations, the containerised service should be offloaded from edge devices to cloud resources. That means a new deployment or reconfiguration of the application needs to occur at runtime.

The Edge On/Offloading Server receives new deployment or even reconfiguration instructions from the Autonomic Data Intensive Application Manager. The Edge On/Offloading Server translates these instructions into different requests which are then sent to the On/Offloading Client. The On/Offloading Client, which is installed on every edge node, responds to the On/Offloading Server's requests to start or stop container instances.

The goal of the present deliverable is therefore threefold:

- At first, to explain modern types of technologies such as containerised microservices which are able to support dynamic adaptation of applications within edge computing frameworks as well as to describe the state-of-the-art review for edge computing technologies able to address highly dynamic workloads at runtime.
- Secondly, to propose a new modern on/offloading method provided by the PrEstoCloud solution which includes two key components: Edge On/Offloading Server and On/Offloading Client.
- Finally, to describe step-by-step instructions for the use of the new implemented on/offloading method.

During the next stages of the project, the Mobile Offloading Processing Microservice will be updated and the results shall be included in the deliverable D3.8 (Mobile Offloading Processing Microservice – Iteration 2 [M30]). In the course of the next year, we need to continuously monitor and systematise various ongoing efforts of European and other international projects related to the Mobile Offloading Processing Microservice.

2. Introduction

The PrEstoCloud project proposes remarkable contributions in the domain of real-time data intensive applications orchestrated upon cloud and edge computing frameworks. The PrEstoCloud solution provides a dynamic, distributed, self-adaptive and proactively configurable architecture for processing Big Data streams, especially at the extreme edge of the network. After 18 successful months of the project, the principal objective of this deliverable is to report on the first version of a set of developed software components in order to perform the on/offloading procedure between edge resources and cloud-based infrastructures. These components are the Edge On/Offloading Server and the On/Offloading Client which play a key role in the PrEstoCloud platform.

2.1. Scope of the deliverable

According to the edge computing paradigm in which the execution environment is becoming more dynamic and workloads change over time, using lightweight virtualisation technologies such as containerised microservices can support self-adaptation of the entire system to address the needs of both application providers and users. In this regard, this deliverable presents not only the significance of such modern technologies, but also introduces two key components, namely the Edge On/Offloading Server and the On/Offloading Client. These two implemented components included in the PrEstoCloud architecture are aimed at keeping the applications appropriately running and operating in situations where edge resources cannot longer provide computing or storing tasks at runtime. To come up with a solution for such conditions, the containerised service needs to be offloaded from edge devices to cloud resources, or vice versa in terms of an onloading action.

This document includes instructions for the use of the implemented Edge On/Offloading Server and On/Offloading Client. This deliverable is significantly influenced by three other deliverables of the PrEstoCloud project, as follows:

- *D3.5: Mobile Context Analyser - Iteration 1 [M15]* since the Mobile Context Analyser is the component which is responsible for detecting the health status of edge nodes. Therefore, the Mobile Context Analyser makes decisions to trigger if an on/offloading action is required to be performed at runtime.
- *D4.3: Autonomic Data-Intensive Application Manager - Iteration 1 [M18]* since the Edge On/Offloading Server receives reconfiguration or even new deployment instructions directly from the Autonomic Data Intensive Application Manager.
- *D6.1: Architecture of the PrEstoCloud platform [M15]* since the deliverable D6.1 presents the whole design of the PrEstoCloud architecture which includes both Edge On/Offloading Server and On/Offloading Client.

This deliverable represents the first iteration of the Mobile Offloading Processing Microservice delivered in M18. The second iteration will contain the final description and further implementation of these materials, and it will be delivered in M30.

2.2. Structure of the deliverable

The rest of this deliverable is structured as follows:

- Section 3 explains the modern technologies to be used by the PrEstoCloud project in order to address on/offloading requirements for real-time, Big Data processing applications orchestrated upon edge computing frameworks.

- Section 4 presents the typical scenario in which the on/offloading action should be accomplished.

- Section 5 describes the container control API exposed by the On/Offloading Client to receive all requests from the Edge On/Offloading Server. Such requests issued by the Edge On/Offloading Server can be terminating or instantiating container instances.

- Section 6 explains the deployment or reconfiguration API exposed by the Edge On/Offloading Server to receive abstractions of the application from the Autonomic Data Intensive Application Manager.
- Section 7 provides a detailed description of the edge node registration API exposed by the Edge On/Offloading Server allowing the On/Offloading Client for the registration of edge node.
- Finally, we conclude the document in Section 8, and discuss next steps of the project with respect to the Mobile Offloading Processing Microservice.

3. Background

This section is divided into four subsections. The first three subsections describe the adoption of microservice architectural approach and also the necessity of using container-based virtualisation which can be seen as enablers of microservices within highly dynamic environments, e.g. edge computing scenarios. Furthermore, the last subsection analyses the state-of-the-art edge computing technologies capable of handling highly dynamic workloads at runtime.

3.1. Microservice architectural approach

Microservices are small, independent, highly decoupled processes which communicate with each other to form complex applications that exploit language-independent Application Programming Interfaces (APIs). Decomposing a single application into smaller microservices allows application providers to distribute the computational load of services among different resources, e.g. different edge nodes in even geographical locations.

In comparison with Service Oriented Architecture (SOA), microservices are usually organised around business priorities and capabilities, and they have the capability of independent deployability [3] and often employ the use of simplified interfaces, such as Representational State Transfer (REST). Resilience to failure could be the next characteristic of microservices. Since in this modern architecture, every request will be separated and translated to various service calls. For instance, a bottleneck in one service brings down only that service and not the entire system. In such a situation, other services will continue handling requests normally. Therefore, the microservice architectural approach is able to support the whole spectrum of requirements namely modularity, scalability, distribution and fault-tolerance [4].

Figure 1 shows a microservice architectural approach compared with a monolithic application. Monolithic application as a single entity includes the entire functional logic of the software. The monolithic application mediates between the end-users and the database—handling requests, retrieving data from the database, processing the information and assembling the result sent to the users. Therefore, all functional logic for handling requests runs within only a single process. An important disadvantage is that even a very tiny update made to a small segment of the application requires the entire monolith to be re-developed and then re-deployed again. In the microservice architectural approach instead of the whole being one large monolith, each business capability can be a self-contained service with a well-defined interface called REST API. Therefore, a large application is composed of multiple, small, discrete, scalable and independent (micro-) services—each one is responsible for its own objective. It should be noted that each microservice running on a specific host can be also easily terminated, and then instantiated on another host at runtime.

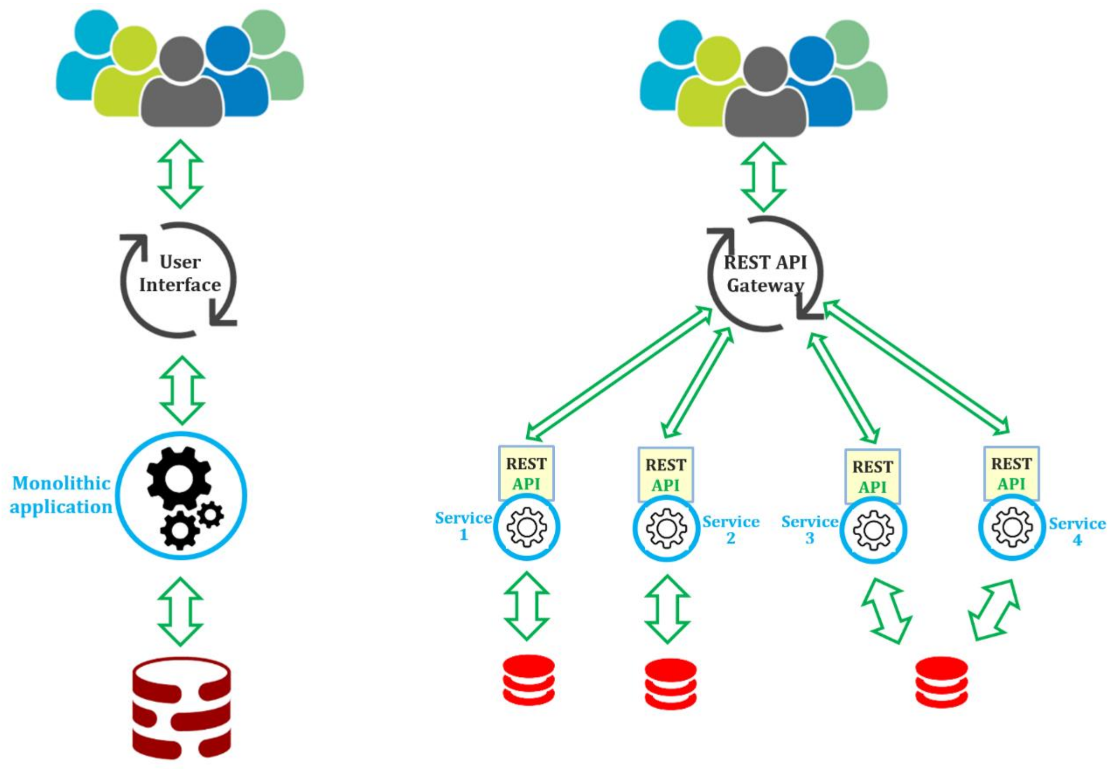

Figure 1: Monolithic architecture vs. microservice architectural approach

Decentralised data management is another significant advantage of microservice architectural approach. This is because each service which provides a particular function of the business process may manage its own database. This competency supports the whole system able to optimise data storage, information processing and knowledge acquisition towards different requirements of each business function.

Moreover, the provisioning or de-provisioning of resources allocated to the monolithic application in response to the changing workload is not optimised. Because the entire application needs to be made scalable—as opposed to the microservice architectural approach in which only a specific service which is under greater demand can be scalable. Figure 2 demonstrates an example of microservice architectural approach in which different services have various amount of demands to process their own jobs at runtime, and hence it is possible to dynamically scale each service at different level.

Microservice architectural approach also supports the efficiency of reusable functionality. In other words, microservices are re-usable that means a single service can be re-used in various applications or even in separate areas of the same application.

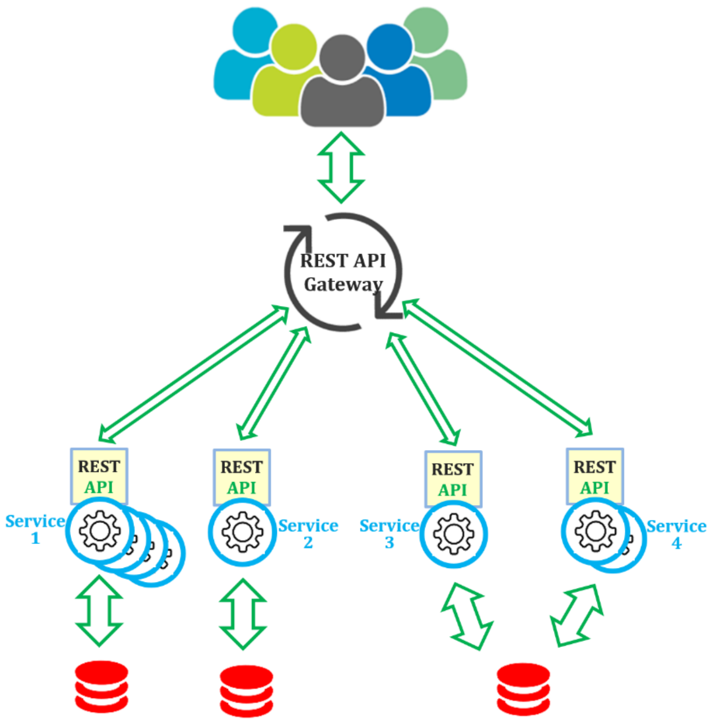

Figure 2: Dynamically scaling each service at different level

3.2. Container-based virtualisation

Hypervisor-based virtualisation technologies [5] are able to support standalone VMs which can be independent and isolated of the host machine. Each VM instance has its own Operating System (OS) and a set of libraries, and operates within an emulated environment provided by the hypervisor. However nowadays, a modern software engineering discipline provides a new method to design cloud-based applications based on a set of components running in containers [6] rather than VMs, shown in Figure 3.

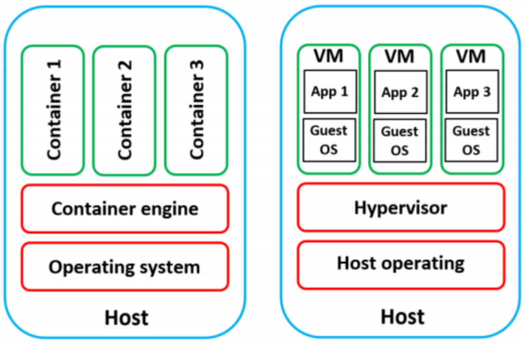

Figure 3: Container versus VM-based virtualisation

Different from VMs, the utilisation of containers does not need an Operating System (OS) to boot up that has gained increasing popularity in the cloud computing frameworks [7]. Resource usage of VMs is extensive, and thus typically they cannot be easily developed on small servers or resource-constrained devices such as Raspberry Pi, in contrast to containers. Table 1 provides a comparison between container-based and VM-based virtualisation [8].

Table 1: Container-based vs VM-based virtualisation

Feature	Containers	VMs
Requirement	Container engine e.g. Docker	Hypervisor e.g. Xvisor
Weight	Lightweight	Heavyweight
Boot Time	Fast	Slow
Footprint	Smaller	Bigger

Since their nature is lightweight, deployment of containerised services at runtime can be accomplished faster than VMs, and hence they have the ability to allow for quick scaling of cloud-based applications [9]. The lightweight nature and portability of containers make it easy to dynamically handle changing workloads, scaling up or scaling down applications when the workload varies over time. In this way, various container-based virtualisation platforms such as Google Container Engine (GCE) [10] and Amazon EC2 Container Service (ECS) [11] have been offered as alternatives to hypervisor-based virtualisation.

Explanations in this deliverable are based on Docker which is a container technology for Linux that allows a developer to package up an application with all of the parts needed [12]. Among all container virtualisation platforms, Docker has been recently very popular and rapidly developing [13]. Docker offers a set of APIs exposed by the Docker engine for creating, instantiating, terminating and managing containers, and also building images, sharing them through repositories and looking for images created and made publically available by any other developer. In other words, the Docker Engine is the core of Docker which provides APIs for starting, stopping, resuming and removing containers.

When a container instance is started, there is a possibility to specify how the container interacts with the host system. To this end, ports can be configured by mapping between internal and external port numbers. In this way, other hosts whether on the same network or on the Internet can communicate with the container via the external port, and the connection will be mapped to the internal port inside the container.

Size of container images is significantly smaller than the size of full VM images. There is a repository named Docker Hub repository [14] by which all developers can share their public container images provided for different purposes, shown in Figure 4. The Docker Hub repository is a database for storing and distributing Docker images. The Docker hub repository exposes APIs which can be called to push (store) and pull (retrieve) container images [15]. The Docker registry can be also installed locally to store Docker images. When a local registry is used, pulling container images will be faster, and hence running container instances will be quick. Using a local Docker registry significantly decreases deployment latency as well as network overhead across the network.

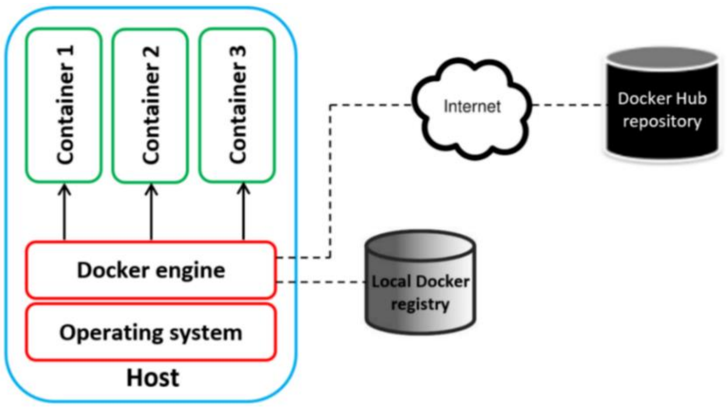

Figure 4: Docker Hub repository

All Docker images stored in this registry are identified by their name, while different versions of a container image can be stored under the same name. A Docker image is generally described in a text file called Dockerfile [16]. Dockerfile consists of consecutive steps on how to create a specific Docker image, what base image should be used, which dependencies need to be downloaded, what commands have to run to execute the containerised application, port numbers which the container listens for network connections at runtime, and so on.

3.3. Containerised microservices

Docker platform is aimed at automating the deployment of services inside portable container instances which are independent of programming language, host Operating System (OS), and hardware characteristics. A microservice may run in a container or VM. However, microservices are able to be deployed based upon containers faster than VMs. In other words, what makes containers an appropriate fit for microservices is their lightweight nature.

Docker container-based virtualisation (such as Docker [17], CoreOS [18], Kubernetes [19], OpenShift Origin [20] and Swarm [21]) is currently considered as a supporting technology for edge computing frameworks in which each container depicts a different functionality and runs as an independent containerised microservice [22]. This is because such promising container technology combined with microservice architectural approach offers a great level of agility in developing, deploying and running applications. This fact can make containerised microservices significantly useful especially during any instantiation and termination of services, or horizontal scaling of services, or when a microservice should be re-deployed due to a network or server failure. In essence, containerised microservices can be dynamically deployed or re-deployed on demand better responding to changing workloads at runtime without any unacceptable application performance degradation.

Figure 5 shows that the PrEstoCloud project is driven by containerised microservices paradigm. In this way, the PrEstoCloud solution provides a dynamic deployment of containerised services—in which case each container instance is able to be used in order to allocate specific tasks and data analytics performed at the edge. The deployment scheme to run the service on whether edge node or cloud resource can be chosen at runtime based on the application requirements and the status of the execution environment.

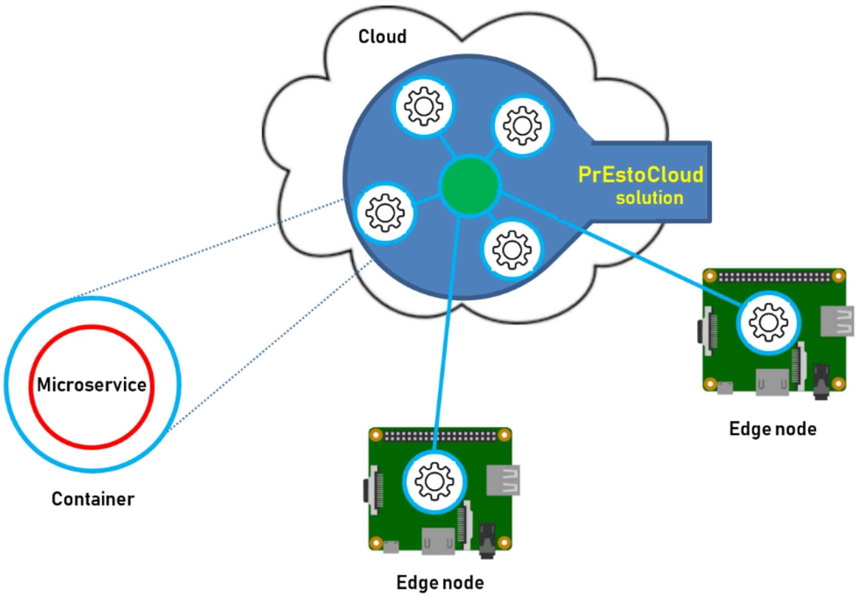

Figure 5: Dynamic deployment scheme within edge computing frameworks

When an edge node is not able to provide computing operations, for example if the free storage capacity on the edge node is not available any more, or if the edge node is overloaded due to an increase in the number of connected sensors during execution and hence there are no spare cycles on the edge resource, the containerised service deployed on the edge node should be terminated and started on the cloud side. Along this line, Figure 6 shows that dynamic on/offloading action needs to occur when there is a limited amount of resources (such as free storage capacity or available computing power) on the edge node at runtime.

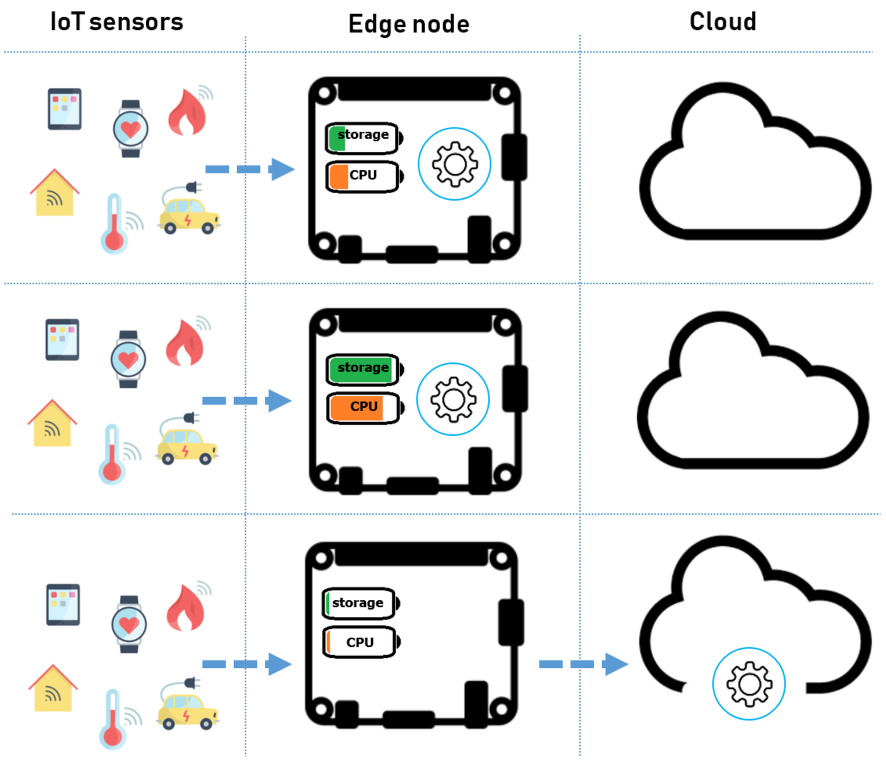

Figure 6: Dynamic on/offloading action in edge computing framework

The PrEstoCloud project continuously monitors different metrics of infrastructures (e.g. CPU, memory, storage, etc.), and hence determines if the application performance needs to be improved at execution time. To this end, the containerised service running on the edge node can be stopped before it goes down, and then it will be launched on the cloud on-the-fly. Therefore, an objective of the PrEstoCloud project is to offer a new computing mechanism to offload or onload dynamic workload during execution. Such a solution creates a substantial contribution to the progress of providing a dynamic, distributed architecture for proactive resource management.

3.4. State-of-the-art review

The size of container image which is smaller than VM images makes it suitable to launch services at the edge faster than VM-based appliances. However, edge nodes allocated to run containers usually have hardware resource limitations in reality, or in other words, they are not strong enough in comparison with cloud-based hosts. Offering desirable application performance provided by edge computing frameworks in conditions where the workload dynamically changes has been discussed in experience studies. We categorise all methods proposed by such experience studies into four groups: (i) Using a static amount of edge resources as a complete replacement to cloud infrastructures, (ii) Discovering available edge resources when the workload increases at runtime, (iii) Replicating services on both edge nodes and cloud resources at the same time, and (iv) Using only edge nodes allocated to process incoming requests by default and exploiting cloud-based resources when they are required because of the increase in the workload. Here, we present a review of these four types of methods proposed in edge computing frameworks. The differences and similarities among these presented works give us an opportunity for comprehensive conception of an appropriate edge computing architecture able to cope with highly dynamic workloads at runtime.

- **Using a static amount of edge resources as a complete replacement to cloud infrastructures:** The best practice in such a solution is exploiting general-purpose edge nodes able to be involved in heterogeneous types of computation and data analytics. To this end, the LightKone project [23] is recently aimed at moving computation completely out of the cloud and directly on the extreme edge of the network. As another example, the main objective of Open Edge Computing (OEC) project [24] is enabling all nearby edge components such as Wi-Fi access points, DSL-boxes, base stations to offer computing resources through standardised, open mechanisms to any types of applications. Similarly, ParaDrop [25] was also proposed that only considers using edge nodes as the replacement to the centralised cloud. Such projects mentioned focus mainly on offering distributed orchestration frameworks through which edge resources can be dynamically assigned to running services. Such solutions can be also extended to be able to address continuously changing workloads. For example, exploiting edge-based distributed caching policies [26] may increase the chance to react to runtime fluctuations in the workload before a performance issue arises. However, achieving the goal to have all edge nodes such as Wi-Fi access points, DSL-boxes and base stations able to handle the workload as general-purpose computation is still challenging. This is because, at first, such static edge resources cannot be virtualised as they have physical items like attached antennas, and hence they may not be scalable enough to handle increasing workloads at runtime. Moreover, different types of services typically are not convenient to be co-located on the same edge node used in these projects since, for example, some applications are network-intensive while others are compute-intensive.

- **Discovering available edge resources when the workload increases at runtime:** Exploiting the edge of the network in this approach needs discovery methods to find available edge nodes able to be leveraged in a distributed computing environment [27, 28, 29]. Resource discovery methods employed in the cloud environment cannot be useful in this context for the discovery of edge nodes. This is because edge nodes usually are in a private network, and hence organisations should articulate regulations of using their own edge resources with those who may exploit these devices [30]. In this regard, Zenith as a resource allocation model was proposed in order to allocate edge-based resources that allows edge infrastructure providers to establish resource sharing contracts with edge service providers [31]. Moreover, consideration of security associated with multi-tenancy on edge nodes comes first in such environments in which the edge-based resource provider and each of customers should have different views on the infrastructure [32]. Besides, it is not an easy task to convince other entities to make use of their own volunteer edge nodes in order to enhance computing capability at run time [33]. It should be also noted that such volunteer edge resources are generally less reliable and less predictable since they may leave the execution environment at any time.
- **Replicating services on both edge nodes and cloud resources at the same time:** The system includes replicated services running not only on edge resources close to the users, but also on cloud infrastructures [34, 35, 36, 37] which means resources are wasted when the workload drops, that is not desirable. In this context, if edge resources allocated to process incoming requests are overloaded, and hence edge nodes are no longer capable of improving the application QoS, additional arrived requests will be sent to the application server running on the cloud. However, replication of servers comes with its own technical challenges, for example temporary inconsistencies are required to be taken into account. Moreover, different organisations have various regulations of using computing and storage infrastructures such as legislation on the geographic location of service instances or data storage servers [38]. Moreover, the load-balancing mechanism has a key role in such methods to achieve a high level of resource efficiency and avoid bottlenecks.
- **Using edge nodes allocated to process incoming requests by default and exploiting cloud-based resources when they are required because of an increase in the workload:** Edge resources are relatively limited in comparison with cloud-based infrastructures. While serving an increasing number of arrival requests coming from users nearby, an edge node may be inevitably overloaded and unable to avoid QoS degradation experienced by the end-users. As a baseline in such situations, PrEstoCloud is capable of offloading further requests from the edge node to the remote cloud which can handle unlimited amount of workloads. However, an open challenge and important technical problem in presenting an edge computing framework is to decide to what extent the proposed method is able to ensure application QoS requirements, while the provisioning approach can avoid wasting costly resources. To this end, the PrEstoCloud solution is able to allocate required cloud-based resources and release unnecessary cloud-based resources to achieve the optimal resource allocation at run-time. In order to manage cloud-based resources, PrEstoCloud directly interacts with the Infrastructure as a Service (IaaS) APIs to acquire and release infrastructures.

4. On/offloading scenario

The typical scenario in the PrEstoCloud solution is an IoT system which includes sensors (e.g. camera [39], accelerometer [40], etc.) able to measure some parameters and send the raw data to a container-based service running on the edge node. Each container instance running on the edge node may consist of two parts: (i) aggregating data part and (ii) processing data part.

The aggregating data part is connected to sensors. The part used to aggregate data receives the raw sensory data (e.g. streaming data [41]), aggregates the data and then sends the aggregated data to another part which is used to process the data. Raw input sensory data collected from different sensors should be pre-processed by the aggregating data part since, for example, the negative effect of undesired noises in measurements needs to be smoothed out. The processing data part is exploited in order to process the data. This part provides the actual real-time data-intensive computing services. Both aggregating data part and processing data part are performed in the container running on the edge node from the beginning by default, shown in Figure 7.

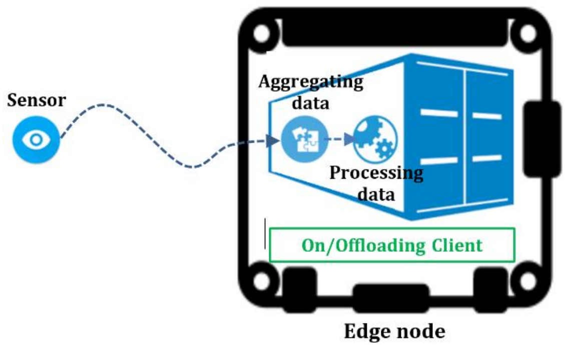

Figure 7: Container running on the edge node consisting of aggregating data part and processing data part

As depicted in Figure 7, the On/Offloading Client is already installed on the edge node. Moreover, the aggregating data part as well as the processing data part run on the edge node by default if there are enough resources in terms of disk, CPU and memory to handle the current workload or store any information. In this case, the part applied to aggregate data receives the raw streaming data from the sensor part, collects and sends the aggregated data to another part which is used to process the data. Under this circumstance, both aggregating and processing data are performed in the container running on the edge node.

There are conditions where an edge node is not able to provide computing operations any more. For example, if the edge node is overloaded because of a drastic increase in the number of sensors connected to the system during execution time, or there is only a small amount of available storage capacity left on the edge node. This is because edge devices such as Raspberry Pi practically suffer from hardware resource restrictions such as limited amount of computation power or storage capacity. Therefore, an on/offloading action should be accomplished to terminate the containerised service running on the edge node, and then deploy a new service on the cloud side.

In such situations, the container instance running on the edge node should be stopped and started on the cloud. The On/Offloading Client responds to the Edge On/Offloading Server's request which is to stop the container which includes both aggregating and processing data on the edge node and start the container on the cloud. In this case, both aggregating and processing data will be then executed in the container running on the cloud. To this end, two successive steps should be accomplished by the On/Offloading Client: (1) stop the current container instance running on the edge node, (2) start a new container instance on the cloud that includes both data aggregating part and data processing part, as shown in Figure 8.

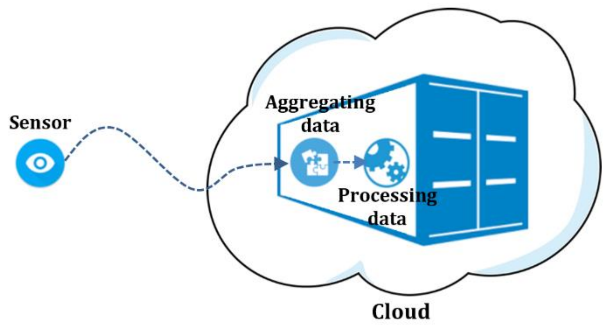

Figure 8: Container running on the cloud consisting of aggregating data part and processing data part

In some cases, sensors are physically connected to the edge node, and the part applied to aggregate the data should be run only on the edge node. In such conditions, the part which is employed to process the data should be stopped on the edge node and started on the cloud. Therefore in such cases, the edge node operates as intermediary to transmit the data to the cloud for data processing. To this end, the On/Offloading Client responds to the Edge On/Offloading Server's request for the on/offloading actions. In this case, the request is terminating (stop) the processing data on the edge node and launching (start) the processing data on the cloud. In other words, if this condition happens, there will be two running containers concurrently when the on/offloading action occurs. One container is employed to aggregate data on the edge node, and another one is used to process data on the cloud, as illustrated in Figure 9.

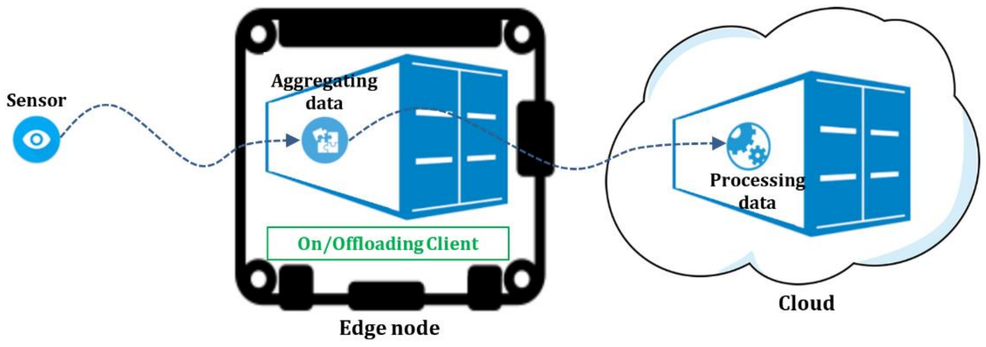

Figure 9: Aggregating data part on the edge and processing data part on the cloud

To this end three successive steps should be accomplished by the On/Offloading Client:

- Request 1: stop the current container instance running on the edge node.
- Request 2: start a new container instance on the cloud that includes only the data processing part.
- Request 3: start a new container instance on the edge node that includes only the data aggregating part.

5. Container control API exposed by the On/Offloading Client

The On/Offloading Client is installed and running on the edge node by default. A container control API, presented in Table 2, is exposed by the On/Offloading Client able to receive all requests from the Edge On/Offloading Server. Such requests issued by the Edge On/Offloading Server can be terminating or instantiating container instances.

Table 2: Container control API

Property	Description
API name	Container control
API explanation	This API allows the Edge On/Offloading Server to send requests (e.g. start or stop requests) in order to launch or terminate container instances.
API provider	On/Offloading Client
API consumer	Edge On/Offloading Server
Implementation	Java
State	Synchronous
API port number	10001

As shown in Figure 10, if the request is successfully performed by the On/Offloading Client, the return value will be 200 or 300 according to the type of request that can be start or stop a container instance, respectively. If the execution of request failed due to an error, the return value will be 201 or 301 according to the type of request that can be start or stop a container instance, respectively.

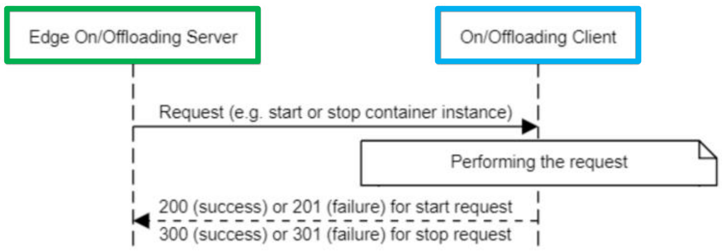
Edge On/Offloading Server
On/Offloading Client
Request (e.g. start or stop container instance)
Performing the request
200 (success) or 201 (failure) for start request
300 (success) or 301 (failure) for stop request

Figure 10: Request issued by the Edge On/Offloading Server to terminate or instantiate container instances

According to the typical on/offloading scenario explained in Section 4, each containerised application can include two parts: (i) aggregating data and (ii) processing data. One part is employed to aggregate the data, and another part is used to process the data. The On/Offloading Client which responds to the Edge On/Offloading Server's requests (e.g. start or stop requests) is able to launch or terminate container instances. In Section 4, different conditions of deployment schemes were explained. Regardless of where a container is started (whether on the edge node or cloud), three different container instantiations may happen as follows:

- When a container is launched, both the aggregating data part and the processing data part should be performed by the container.
- When a container is launched, only the aggregating data part needs to be executed by the container.
- When a container is launched, only processing data part should be performed by the container.

Both Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are protocols used for sending the data over the network. All packets sent via TCP protocol are tracked, and hence no data will be lost or corrupted in transit. When using UDP, packets are sent to the recipient without any acknowledgement. The UDP protocol is used when the connection speed is desirable for the system such as live broadcasts, video conferencing and online games. The communication protocol used to transfer the data between the aggregating data part and the processing data part can be either TCP or UDP, depending on the application type.

Since on one edge node or on one cloud-based host, more than one container may need to be running, port numbers on one resource should be managed. Because, same port cannot be accessed by multiple containers co-located on the same edge node or on the same cloud-based host. As a consequence, at least one port used to communicate the data aggregating part and the data processing part should be defined. In addition to this port, the application itself may need to access to a set of other ports in order to perform its functionalities. Therefore, a set of ports should be defined when a container instance has to be launched. It should be noted that the On/Offloading Agent is developed to be able to consider this point.

Each request issued by the Edge On/Offloading Server to start or stop container instances includes different elements. The general form of a start or stop request defined for the API exposed by the On/Offloading Agent is shown in Figure 11.

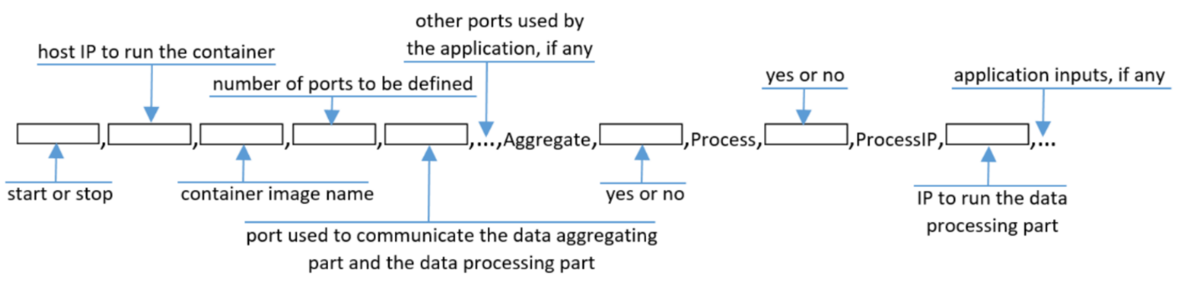
other ports used by the application, if any
host IP to run the container
number of ports to be defined
yes or no
application inputs, if any
,…,Aggregate,
,Process,
,ProcessIP,
,…
start or stop
container image name
yes or no
IP to run the data processing part
port used to communicate the data aggregating part and the data processing part

Figure 11: General form of a request sent to the container control API exposed by the On/Offloading Agent

About input elements included in the request, shown in Figure 11, the following points should be noted:

Point 1: Three different values should be determined for each port determined in a request, illustrated in Figure 11. For example:

- `6790,6790,tcp` ⟶ means mapping TCP port 6790 in the container to port 6790 on the Docker host.
- `80,8080,tcp` ⟶ means mapping TCP port 80 in the container to port 8080 on the Docker host.
- `6790,6790,udp` ⟶ means mapping UDP port 6790 in the container to port 6790 on the Docker host.
- `3333,6790,udp` ⟶ means mapping UDP port 3333 in the container to port 6790 on the Docker host.

Point 2: The application may have its own different input elements depending on the service type. Such inputs need to be determined at the end of a request, illustrated in Figure 11. These inputs are defined by application developers who make the application container image. For example:

- `Threshold,0.3` ⟶ for a logistic use case, one input can be the threshold defined for the acceleration at the rate of 0.3 Gravitational-force (G).
- `Resolution,480` ⟶ for a video service use case, one input can be resolution value that is appropriate to the current streaming data.

Point 3: In both start and stop request, the order of elements is important.

Point 4: If a request, illustrated in Figure 11, is going to be used for terminating (stop) a container, all elements defined in the request are not required to be determined.

EXAMPLE 1

Assume that we need to `start` a container instance on an edge node with the IP address of `194.349.1.175`. The container image name is `salmant/cvs_container_image_tcp:latest`. In order to instantiate the container, the TCP port `6790` in the container should be mapped to port `6790` on the Docker host that can be also different from each other; however, only one port number is chosen. In this case, this port is employed to communicate the data aggregating part and the data processing part that both should be run on the edge node. Moreover, the application has its own parameters associated with the service. To reach the mentioned situation, this following input defined as request should be sent to the container control API exposed by the On/Offloading Agent:

- `"start,194.349.1.175,salmant/cvs_container_image_tcp:latest,1,6790,6790,tcp,Aggregate,yes,Process,yes,ProcessIP,127.0.0.1,Threshold,0.3,FileName,RAW_ACCELEROMETERS.csv,Y_column,6,Z_column,7,Interval,5,DB_IP,194.249.1.42,Driver_ID,782369"`

EXAMPLE 2

Assume that the system works based on the situation presented in Example 1. After a while, a set of dynamic on/offloading actions need to occur since there is a limited amount of free storage capacity at runtime on the edge node with the IP address of `194.349.1.175`. In this example, sensors are physically connected to the edge node, and the part applied to aggregate the data should be run on the edge node. Therefore, only the part which is employed to process the data should be stopped on the edge node and started on the cloud, located on a host machine with the IP address of `83.77.2.48`. In such a case, the edge node operates as intermediary to receive the data from sensors and transmit the data to the cloud for processing. In order to reach the mentioned situation from Example 1, the running container instance should be stopped, and then two new container instances have to be started on both the edge node and the cloud. The first container instance is employed as the aggregating data part, and the second one represents as the processing data part. Therefore, these three successive inputs defined as requests should be respectively sent to the container control API exposed by the On/Offloading Agent:

- `"stop,194.349.1.175,salmant/cvs_container_image:latest,1,6790,6790,tcp"`
- `"start,83.77.2.48,salmant/cvs_container_image:latest,1,6790,6790,tcp,Aggregate,no,Process,yes,ProcessIP,83.77.2.48,Threshold,0.3,FileName,RAW_ACCELEROMETERS.csv,Y_column,6,Z_column,7,Interval,5,DB_IP,194.249.1.42,Driver_ID,782369"`
- `"start,194.349.1.175,salmant/cvs_container_image:latest,1,6790,6790,tcp,Aggregate,yes,Process,no,ProcessIP,83.77.2.48,Threshold,0.3,FileName,RAW_ACCELEROMETERS.csv,Y_column,6,Z_column,7,Interval,5,DB_IP,194.249.1.42,Driver_ID,782369"`

EXAMPLE 3

Assume that the system works based on the situation presented in Example 1. After a while, a set of dynamic on/offloading actions need to occur since there is a limited amount of free storage capacity at runtime on the edge node with the IP address of `194.349.1.175`. As a consequence, at first, we should `stop` the current container instance running on the edge node, and then `start` a new one on the cloud, located on a host machine with the IP address of `83.77.2.48`. In this example, the data captured by sensors should be sent directly to the container instance running on the cloud without the intervention of the edge node. Therefore, in order to reach the mentioned situation from Example 1, these two successive inputs defined as requests should be respectively sent to the container control API exposed by the On/Offloading Agent:

- `"stop,194.349.1.175,salmant/cvs_container_image:latest,1,6790,6790,tcp"`
- `"start,83.77.2.48,salmant/cvs_container_image_tcp:latest,1,6790,6790,tcp,Aggregate,yes,Process,yes,ProcessIP,127.0.0.1,Threshold,0.3,FileName,RAW_ACCELEROMETERS.csv,Y_column,6,Z_column,7,Interval,5,DB_IP,194.249.1.42,Driver_ID,782369"`

5.1. Implementation of aggregating data

The aggregate data part receives the raw sensory data, aggregates the data and then provides the processing data part the aggregated information. The communication protocol used to transfer the data between the aggregating data part and the processing data part can be either TCP or UDP, depending on the application type. This communication can be implemented via different mechanisms such as socket programming. In order to develop the aggregating data part, a Java file should be coded. If the application needs the TCP protocol, the sample Java code, shown in Figure 12, can be considered to implement the aggregating data part.

```java
// ----------------------------------------
// Author: Jozef Stefan Institute (JSI)
// ----------------------------------------
// args[0]: Port to communicate the aggregating and processing data parts together.
// args[1]: IP address where the processing data part is running.
// args[2],...,args[n]: Application input elements, if any.
// ----------------------------------------
import java.io.*;
import java.net.*;
// ----------------------------------------
....... application-specific libraries .......
// ----------------------------------------
class AggregateData_TCP {
   public static void main(String args[]) throws Exception {
      try{
         // ----------------------------------------
         Socket clientSocket = new Socket(args[1], Integer.parseInt(args[0]));
         DataOutputStream outToServer = new
                                 DataOutputStream(clientSocket.getOutputStream());
         // ----------------------------------------
         ....... application-specific code .......
         // ----------------------------------------
         clientSocket.close();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Figure 12: Sample Java code to implement the aggregating data part based on TCP

Each service has its own programming libraries and the way of aggregating data which are specific to the application. These parts which are specific to the application are highlighted in green, shown in Figure 12. The implementation of the proposed aggregating data part based on TCP is also kept publically available on GitHub [42]:

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/AggregateData_TCP.java

If the application needs the UDP protocol, the sample Java code, shown in Figure 13, can be considered to implement the aggregating data part.

```
// ----------------------------------------
// Author: Jozef Stefan Institute (JSI)
// ----------------------------------------
// args[0]: Port to communicate the aggregating and processing data parts together.
// args[1]: IP address where the processing data part is running.
// args[2],...,args[n]: Application input elements, if any.
// ----------------------------------------
import java.io.*;
import java.net.*;
// ----------------------------------------
....... application-specific libraries .......
// ----------------------------------------
class AggregateData_TCP {
   public static void main(String args[]) throws Exception {
      try{
         // ----------------------------------------
         InetAddress IPAddress = InetAddress.getByName(args[1]);
         DatagramSocket clientSocket = new DatagramSocket();
         byte[] sendData = new byte[1024];
         byte[] receiveData = new byte[1024];
         // ----------------------------------------
         ....... application-specific code .......
         // ----------------------------------------
         clientSocket.close();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Figure 13: Sample Java code to implement the aggregating data part based on UDP

The implementation of the proposed aggregating data part based on UDP is also kept publically available on GitHub [43]:

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/AggregateData_UDP.java

About the implementation of aggregating data, presented in Figure 12 and Figure 13, the following points should be taken into consideration:

Point 1: In the sample Java code offered above to implement the aggregating data part, arguments from `args[2]` to `args[n]` are application-specific input elements, if there would be any of them. It should be noted that these elements are determined at the end of the request which is sent to the container control API exposed by the On/Offloading Agent, already explained in Section 5. The application developer should define which input elements need to be determined for the application.

Point 2: In the sample Java code offered above to implement the aggregating data part, there is a part which is highlighted as `....... application-specific libraries .......`. In this part, the application developer should import all libraries which are application-specific and required for the application.

Point 3: In the sample Java code offered above to implement the aggregating data part, there is a part which is highlighted as `....... application-specific code .......`. This part is application-specific and it should be provided by the application developer. For example, the application developer can smooth out data from undesired noises in measurements collected by sensors, or filter some values according to the application's needs.

5.2. Implementation of processing data

In the age of Big Data, it is essential to understand how to manipulate the data in order to extract useful information from the data. For example, vehicle telematics [44] such as fleet management systems and vehicle tracking solutions has gained increasing attention in the context of advanced real-time analytics paradigm. As another example, there is an increasing need in surveillance systems for the real-time processing of huge amount of video data streams to provide a quick summary of interesting events that are happening during a specified time frame in a particular location.

In order to develop the processing data part, a Java file should be coded. If the application needs the TCP protocol, the sample Java code, shown in Figure 14, can be considered to implement the processing data part.

```
// -----------------------------------------
// Author: Jozef Stefan Institute (JSI)
// -----------------------------------------
// args[0]: Port to communicate the aggregating and processing data parts together.
// args[1],...,args[n]: Application input elements, if any.
// -----------------------------------------
import java.io.*;
import java.net.*;
// -----------------------------------------
....... application-specific libraries .......
// -----------------------------------------
class ProcessData_TCP {
   public static void main(String[] args) throws Exception {
      try {
         // -----------------------------------------
         ServerSocket welcomeSocket = new ServerSocket(Integer.parseInt(args[0]));
         Socket connectionSocket = welcomeSocket.accept();
         BufferedReader inFromClient = new BufferedReader(new
                                InputStreamReader(connectionSocket.getInputStream()));
         // -----------------------------------------
         ....... application-specific code .......
      // -----------------------------------------
      } catch (Exception e) {
         //e.printStackTrace();
      }
    }
}
```

Figure 14: Sample Java code to implement the processing data part based on TCP

The implementation of the proposed processing data part based on TCP is also kept publically available on GitHub [45]:

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/ProcessData_TCP.java

If the application needs the UDP protocol, the sample Java code, shown in Figure 15, can be considered to implement the processing data part.

```
// -----------------------------------------
// Author: Jozef Stefan Institute (JSI)
// -----------------------------------------
// args[0]: Port to communicate the aggregating and processing data parts together.
// args[1],...,args[n]: Application input elements, if any.
// -----------------------------------------
import java.io.*;
import java.net.*;
// -----------------------------------------
....... application-specific libraries .......
// -----------------------------------------
class ProcessData_TCP {
   public static void main(String[] args) throws Exception {
      try {
         // -----------------------------------------
         DatagramSocket welcomeSocket = new
                          DatagramSocket(Integer.parseInt(args[0]));
         byte[] receiveData = new byte[1024];
         byte[] sendData = new byte[1024];
         DatagramPacket receivePacket = new
                      DatagramPacket(receiveData, receiveData.length);
         // -----------------------------------------
         ....... application-specific code .......
      // -----------------------------------------
      } catch (Exception e) {
         //e.printStackTrace();
      }
    }
}
```

Figure 15: Sample Java code to implement the processing data part based on UDP

The implementation of the proposed processing data part based on UDP is also kept publically available on GitHub [46]:

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/ProcessData_UDP.java

5.3. Container instantiation performed by the On/Offloading Client

When a container needs to be started, the On/Offloading Client specifies if the container instance is meant to aggregate or process the data. Consequently, when a container is launched, three types of different instantiation may be performed by the On/Offloading Client: (i) both aggregating and processing data parts should be accomplished by the container instance, (ii) only the aggregating data part should be executed by the container instance, and (iii) only the processing data part should be accomplished by the container. To this end, when a container is going to be instantiated, a shell (.sh) file is executed at first to define which one of aggregating data part or processing data part, or even both should be performed in the container.

If the application needs the TCP protocol to provide its functionality, the shell file, shown in Figure 16, can be considered.

```
#!/bin/bash

if [ $Process == "yes" ]&&[ $Aggregate == "yes" ]; then
        java ProcessData_TCP $ServicePort $A $B ... $Z &
        exec java AggregateData_TCP $ServicePort $ProcessIP $a $b ... $z
fi

if [ $Process == "no" ]&&[ $Aggregate == "yes" ]; then
        exec java AggregateData_TCP $ServicePort $ProcessIP $a $b ... $z
fi

if [ $Process == "yes" ]&&[ $Aggregate == "no" ]; then
        exec java ProcessData_TCP $ServicePort $A $B ... $Z
fi
```

Figure 16: Shell file is executed when a container providing a TCP-based service is instantiated

The shell file prepared for a container providing a TCP-based service is also kept publically available on GitHub [47]:

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/start_TCP.sh

If the application needs the UDP protocol to provide its functionality, the shell file, shown in Figure 17, can be considered.

```
#!/bin/bash

if [ $Process == "yes" ]&&[ $Aggregate == "yes" ]; then
        java ProcessData_UDP $ServicePort $A $B ... $Z &
        exec java AggregateData_UDP $ServicePort $ProcessIP $a $b ... $z
fi

if [ $Process == "no" ]&&[ $Aggregate == "yes" ]; then
        exec java AggregateData_UDP $ServicePort $ProcessIP $a $b ... $z
fi

if [ $Process == "yes" ]&&[ $Aggregate == "no" ]; then
        exec java ProcessData_UDP $ServicePort $A $B ... $Z
fi
```

Figure 17: Shell file is executed when a container providing a UDP-based service is instantiated

The shell file prepared for a container providing a UDP-based service is also kept publically available on GitHub [48]:

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/start_UDP.sh

About the shell file which is executed at container instantiation time, presented in Figure 16 and Figure 17, the following points should be taken into consideration:

Point 1: Some application-specific input elements which are determined at the end of the request sent to the On/Offloading Agent are used in the aggregating data part, and some of them are used in the processing data part. In the shell code offered above, `$A` `$B` ... `$Z` are the name of application-specific input elements (if there would be any of them) which are used in the processing data part. In the shell code offered above, `$a` `$b` ... `$z` are the name of application-specific input elements (if there would be any of them) which are used in the aggregating data part. It should be noted that these names should be exactly the names which are determined at the end of the request sent to the container control API exposed by the On/Offloading Agent. If the application does not have any input element, `$A` `$B` ... `$Z` and `$a` `$b` ... `$z` should be eliminated from the code.

Point 2: In the shell code offered above in Figure 16 and Figure 17, it has to be noted that there is a specific character `&` which should not be deleted. This character at the end of Java call means that the process will be started in background.

5.4. Container image used by the On/Offloading Client to run an instance

A Docker container image is a lightweight, stand-alone, executable package of a service which includes all required code, system tools, system libraries, settings, etc. In order to create a container image, a file named Dockerfile needs to be prepared. A Dockerfile consists of consecutive steps on how to create a specific Docker image. In other words, a Dockerfile says what base image should be used, which dependencies need to be downloaded, what commands have to run to execute the containerised application, and so on. In order to make a container image which includes both aggregating data part and processing data part, application developers should follow the following instructions, respectively:

Step 1: At first, login to your server and update the software repository. Then, make a new folder via `mkdir`, and go inside the directory via `cd`.

Step 2: Compile your code—both AggregateData_TCP.java (or AggregateData_UDP.java) and ProcessData_TCP.java (or ProcessData_UDP.java) via `javac` to have AggregateData_TCP.class (or AggregateData_UDP.class) and ProcessData_TCP.class (or ProcessData_UDP.class).

Step 3: Inside the directory made at the first stage, the application developer should make a plain text file named `Dockerfile` without extension as follows:

If the application needs the TCP protocol to provide its functionality, the Dockerfile file, shown in Figure 18, can be considered to create container image.

```
FROM openjdk:7-jre

MAINTAINER "Jozef Stefan Institute"

##############################################################
RUN mkdir -p "/JavaFiles"
WORKDIR /JavaFiles
COPY AggregateData_TCP.class /JavaFiles/AggregateData_TCP.class
COPY ProcessData_TCP.class /JavaFiles/ProcessData_TCP.class
COPY start_TCP.sh /JavaFiles/start_TCP.sh
RUN chmod 777 /JavaFiles/start_TCP.sh
##############################################################
....... application-specific commands .......
##############################################################

ENTRYPOINT ["/JavaFiles/start_TCP.sh"]
```

Figure 18: Dockerfile used to create container image for TCP-based services

The Dockerfile used to create container image for TCP-based services is also kept publically available on GitHub [49]:

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/Dockerfile_TCP

If the application needs the UDP protocol to provide its functionality, the Dockerfile file, shown in Figure 19, can be considered to create container image.

```
FROM openjdk:7-jre

MAINTAINER "Jozef Stefan Institute"

##############################################################
RUN mkdir -p "/JavaFiles"
WORKDIR /JavaFiles
COPY AggregateData_UDP.class /JavaFiles/AggregateData_UDP.class
COPY ProcessData_UDP.class /JavaFiles/ProcessData_UDP.class
COPY start_UDP.sh /JavaFiles/start_UDP.sh
RUN chmod 777 /JavaFiles/start_UDP.sh
##############################################################
....... application-specific commands .......
##############################################################

ENTRYPOINT ["/JavaFiles/start_UDP.sh"]
```

Figure 19: Dockerfile used to create container image for UDP-based services

The Dockerfile used to create container image for UDP-based services is also kept publically available on GitHub [50]:

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/Dockerfile_UDP

In the Dockerfile which is used to create container image offered above in Figure 19, there is a specific part which is highlighted as `....... application-specific commands ........` In this part, the application developer should write all commands which are application-specific and required for the application. Such a script in this part contains a set of application-specific instructions which will be automatically executed in sequence in the Docker environment for building the container image.

Step 4: When the application developer has prepared all Java codes as well as the Dockerfile, the container image should be built via `docker build`.

If the application needs the TCP protocol to provide its functionality, the following command can be considered to build the container image:

```
docker build -t jsi/x_container_image_tcp -f Dockerfile_TCP .
```

If the application needs the UDP protocol to provide its functionality, the following command can be considered to build the container image:

```
docker build -t jsi/x_container_image_udp -f Dockerfile_UDP .
```

It should be noted that application developers should put their repository name instead of `jsi`, and put their container image name instead of `x_container_image_tcp/udp` to be assigned to the service.

Step 5: When the container image has been built, the application developer should push the built container image on the repository via `docker push`.

If the application needs the TCP protocol to provide its functionality, the following command can be considered to push the container image:

```
docker push jsi/x_container_image_tcp
```

If the application needs the UDP protocol to provide its functionality, the following command can be considered to build the container image:

```
docker push jsi/x_container_image_udp
```

5.5. Docker's Remote API as a requirement for the On/Offloading Client

The developed On/Offloading Client works based on the Docker's Remote API to instantiate new containers or terminate running containers. In essence, the Docker's Remote API is exploited for communication with Docker daemon which is the core module of the Docker virtualisation platform and it controls status of containers. On every edge node or cloud-based host, Docker engine should be installed, and Docker's Remote API should be enabled. To this end, following instructions should be done, respectively:

Step 1: The command `apt-get update` should be executed. This command downloads the package lists from the repositories and updates them to get information on the newest versions of packages and their dependencies.

Step 2: The command `apt-get -y install docker.io` should be executed. This command installs Docker.

Step 3: The command `ln -sf /usr/bin/docker.io /usr/local/bin/docker` should be executed. This command links the Docker path.

Step 4: The command `service docker stop` should be executed. This command stops the Docker engine.

Step 5: The line `DOCKER_OPTS='-H tcp://0.0.0.0:4243 -H unix:///var/run/docker.sock'` should be added to this file `/etc/default/docker`.

Step 6: The command `service docker start` should be executed. This command starts the Docker engine again.

5.6. Running the On/Offloading Client

Regardless of what communication protocol (either TCP or UDP) is used by the application, the Java code, shown in Figure 20, should be compiled in order to run the On/Offloading Client deployed on every edge node. In the Java code offered below, the container control API exposed by the On/Offloading Agent is listening to the port with the number of 10001 to receive requests (start or stop requests) sent from the Edge On/Offloading Server, explained in previous sections.

```
// -----------------------------------------------------------------------
// Author: Jozef Stefan Institute (JSI)
// -----------------------------------------------------------------------

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Properties;
import java.net.*;

public class On_Offloading_Client extends Thread implements Runnable {

        public static void main(String arg[]) throws Exception {
                ServerSocket welcomeSocket = new ServerSocket(10001);
                while (true){
                        String ServicePort[];
                        String HostPort[];
                        String tcp_or_udp[];
                        String input_name[];
                        String input_value[];
                        int inputs_count = 0;
                        int ports_count = 0;
                        Socket connectionSocket = welcomeSocket.accept();
                        BufferedReader inFromClient =
                        new BufferedReader(new
                        InputStreamReader(connectionSocket.getInputStream()));
                        String clientSentence = inFromClient.readLine();
                        String[] argv = clientSentence.split(",");
                        String IP = argv[1];
                        String ContainerImageName = argv[2];
                        ports_count= Integer.parseInt(argv[3]);
                        ServicePort = new String[ports_count];
                        HostPort =  new String[ports_count];
                        tcp_or_udp =  new String[ports_count];
                        for (int i=0; i < ports_count; i++) {
                                        ServicePort[i] = argv[4+(i*3)];
                                        HostPort[i] = argv[5+(i*3)];
                                        tcp_or_udp[i] = argv[6+(i*3)];
                        }
                        if (argv[0].equals("start")){
                                input_name = new String
                                [argv.length - 4 - (ports_count * 3) + 1 ];
                                input_value = new String
                                [argv.length - 4 - (ports_count * 3) + 1 ];

                                for (int i=(4 + (ports_count * 3)); i <
                                argv.length; i=i+2) if (argv[i]!=null){
                                        input_name[inputs_count] = argv[i];
                                        input_value[inputs_count] = argv[i+1];
                                        inputs_count++;
                                }
                                input_name[inputs_count] = "ServicePort";
                                input_value[inputs_count] = argv[4];
                                inputs_count++;
                                String Id = create_a_container(IP,
                                ContainerImageName, ServicePort, HostPort,
                                tcp_or_udp, input_name, input_value, inputs_count,
                                ports_count);
                                if (Id==null || Id.trim().equals("")){
                                        System.out.println("201");
                                }
                                else {
                                        String Result = start_a_container(Id, IP);
                                        if (!(Result==null ||
                                        Result.trim().equals(""))){
                                                System.out.println("202");
                                        }
                                        else System.out.println("200");
                                }
                        } else
                        if (argv[0].equals("stop")){
                                String Id = get_container_id(ServicePort[0],
                                HostPort[0], tcp_or_udp[0], IP);
                                if (Id.equals("")) System.out.println("301");
                                else {
                                        stop_a_container(Id, IP);
                                        System.out.println("300");
                                }
                        }
                }
        }

        /////////////////////////////////////////////////////////////////

        public static String create_a_container(String IP, String
        ContainerImageName, String ServicePort[], String HostPort[], String
        tcp_or_udp[], String input_name[], String input_value[], int inputs_count,
        int ports_count){
        try{
                        String mainCommand =
                        "       curl -X POST -H \'Content-Type: application/json\'
                                -d \'{   " +
                        "       \"Image\":\"" + ContainerImageName + "\",  " ;
                        if (inputs_count>0){
                                mainCommand += " \"Env\": [        " ;
                                for (int i=0; i < inputs_count; i++)
                                        if (i != inputs_count-1) mainCommand
                        +="\"" + input_name[i] + "=" + input_value[i] + "\"," ;
                                        else mainCommand +="\"" + input_name[i] +
                        "=" + input_value[i] + "\"" ;
                                mainCommand += "            ],    " ;
                        }
                        mainCommand +=
                        "       \"Tty\": true,    " +
                        "       \"ExposedPorts\": { " ;
                        for (int i=0; i < ports_count; i++)
                                if (i != ports_count-1) mainCommand += "\"" +
                                ServicePort[i] + "/" + tcp_or_udp[i] + "\": {},";
                                else mainCommand += "\"" + ServicePort[i] + "/" +
                                tcp_or_udp[i] + "\": {}";
                        mainCommand += "},        ";
                        mainCommand +=
                        "       \"PortBindings\": { " ;
                        for (int i=0; i < ports_count; i++)
                                if (i != ports_count-1) mainCommand += "\"" +
                                ServicePort[i] + "/" + tcp_or_udp[i] + "\": [{
                                \"HostPort\": \"" + HostPort[i] + "\" }],";
                                else mainCommand += "\"" + ServicePort[i] + "/" +
                                tcp_or_udp[i] + "\": [{ \"HostIp\":
                                \"\",\"HostPort\": \"" + HostPort[i] + "\" }]";
                        mainCommand += "}         " +
                        "       }\' http://" + IP + ":4243/containers/create"
                        ;
                        //System.out.println(mainCommand);
                        String[] command={"/bin/bash", "-c", mainCommand};
                        Process P = Runtime.getRuntime().exec(command);
            P.waitFor();
            BufferedReader StdInput = new BufferedReader(new
                                          InputStreamReader(P.getInputStream()));
            String TopS ="";
            TopS= StdInput.readLine();
            TopS = TopS.substring(6);
            TopS = TopS.substring(TopS.indexOf("\"") + 1);
            TopS = TopS.substring(0, TopS.indexOf("\""));
            return TopS;
        }catch(Exception ex){
                        ex.printStackTrace();
                        return "";
        }
    }
        ////////////////////////////////////////////////////////////////

        public static String start_a_container(String Id, String IP){
        try{
                        String mainCommand = "curl -X POST http://" + IP +
                        ":4243/containers/" + Id + "/start";
                        //System.out.println(mainCommand);
                        String[] command={"/bin/bash", "-c", mainCommand};
                        Process P = Runtime.getRuntime().exec(command);
            P.waitFor();
            BufferedReader StdInput = new BufferedReader(new
                                          InputStreamReader(P.getInputStream()));
            String TopS ="";
            TopS= StdInput.readLine();
            return TopS;
        }catch(Exception ex){
                        ex.printStackTrace();
                        return "";
        }
    }
        ////////////////////////////////////////////////////////////////

        public static String get_ListOfContainers(String IP){
                try{
                        String mainCommand = "curl -X GET http://" + IP +
                        ":4243/containers/json";
                        String[] command={"/bin/bash", "-c", mainCommand};
                        Process P = Runtime.getRuntime().exec(command);
                        P.waitFor();
                        BufferedReader Input = new BufferedReader(new
                                          InputStreamReader(P.getInputStream()));
                        String ListOfContainers = Input.readLine();
                        ListOfContainers = ListOfContainers.substring(1,
                                                  ListOfContainers.length()-1);
                        return ListOfContainers;
                }catch(Exception ex){
                        ex.printStackTrace();
                        return "";
                }
        }
        ////////////////////////////////////////////////////////////////

        public static int findClosingBracketMatchIndex(String str, int pos) {
                int depth = 1;
                for (int i = pos + 1; i < str.length(); i++) {
                        switch (str.charAt(i)) {
                                case '{':
                                        depth++;
                                        break;
                                case '}':
                                        if (--depth == 0) {
                                                return i;
                                        }
                                        break;
                        }
                }
                return -1; // No matching closing parenthesis
        }
        ////////////////////////////////////////////////////////////////

        public static String get_container_id(String PrivatePort, String PublicPort,
        String Type, String IP){
                String ListOfContainers = get_ListOfContainers(IP);
                int i=-1;
                while (i!=ListOfContainers.length()-1){
                        i = findClosingBracketMatchIndex(ListOfContainers, 0);
                        String tmp = ListOfContainers.substring(0, i+1);
                        String match = "\"PrivatePort\":" + PrivatePort +
                        ",\"PublicPort\":" + PublicPort + ",\"Type\":\"" +
                        Type + "\"";
                        if (tmp.contains(match)){
                                tmp = tmp.substring(7, 71);
                                return tmp;
                        }
                        if (i==ListOfContainers.length()-1) break;
                        ListOfContainers = ListOfContainers.substring(i+2,
                                                ListOfContainers.length());
                }
                return "";
        }
        ////////////////////////////////////////////////////////////////

        public static void stop_a_container(String Id, String IP){
        try{
                        String mainCommand = "curl -X POST http://" + IP +
                                        ":4243/containers/" + Id + "/stop";
                        String[] command={"/bin/bash", "-c", mainCommand};
                        Process P = Runtime.getRuntime().exec(command);
            P.waitFor();
            BufferedReader StdInput = new BufferedReader(new
                                InputStreamReader(P.getInputStream()));
            String TopS ="";
            while((TopS= StdInput.readLine())!=null){}
        }catch(Exception ex){
            ex.printStackTrace();
        }
    }
    ////////////////////////////////////////////////////////////////
}
```

Figure 20: Java code to implement the On/Offloading Client

The Java code which should be compiled in order to run the On/Offloading Client is also kept publically available on GitHub [51]:

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/On_Offloading_Client.java

6. Deployment or reconfiguration API exposed by the Edge On/Offloading Server

The deployment or reconfiguration API, presented in Table 3, is exposed by the Edge On/Offloading Server to receive JSON-based abstractions of applications running on edge resources. In other words, the Edge On/Offloading Server receives new deployment or reconfiguration instructions on the port number 20001 from the Autonomic Data Intensive Application Manager. Afterwards, the Edge On/Offloading Server translates these instructions into platform-dependent deployment requests which are sent to the container control API exposed by the On/Offloading Client, explained before in Section 5.

Table 3: Deployment or reconfiguration API

Property	Description
API name	Deployment or reconfiguration
API explanation	This API allows the Autonomic Data Intensive Application Manager to send instructions which can be new deployment or reconfiguration issued for applications running on edge resources.
API provider	Edge On/Offloading Server
API consumer	Autonomic Data Intensive Application Manager
Implementation	Java
State	Synchronous
API port number	20001

As shown in Figure 21, if the deployment or reconfiguration action is successfully performed, the return value will be 500. If the execution of the deployment or reconfiguration action failed due to an error, the return value will be 501.

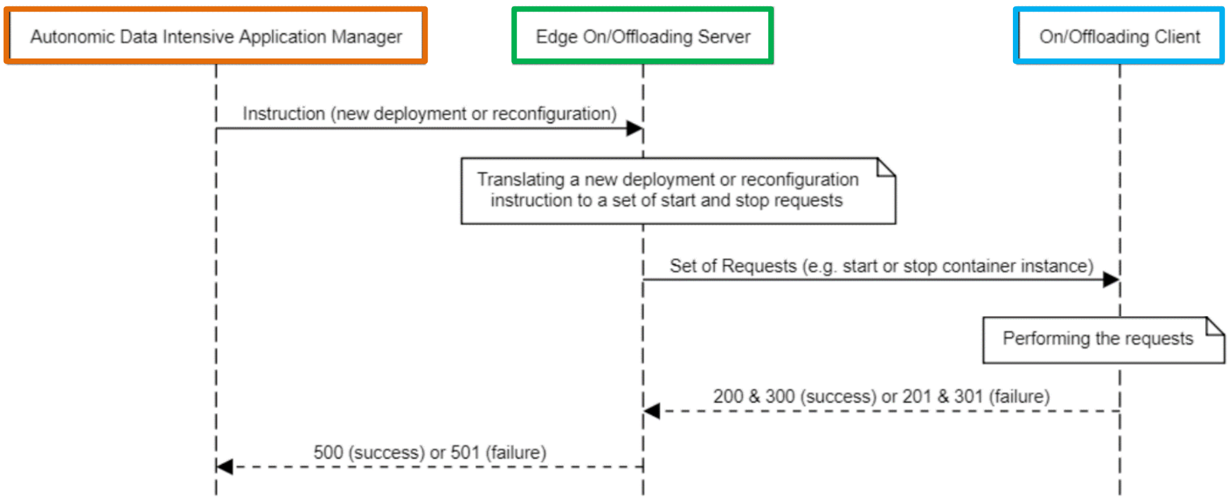
Autonomic Data Intensive Application Manager
Edge On/Offloading Server
On/Offloading Client
Instruction (new deployment or reconfiguration)
Translating a new deployment or reconfiguration instruction to a set of start and stop requests
Set of Requests (e.g. start or stop container instance)
Performing the requests
200 & 300 (success) or 201 & 301 (failure)
500 (success) or 501 (failure)

Figure 21: Instruction issued by the Autonomic Data Intensive Application Manager for the deployment or reconfiguration of the application running on the edge node

It should be noted that the communication between the Edge On/Offloading Server and the On/Offloading Client, depicted in Figure 21, was thoroughly explained by the container control API in detail. Figure 22 shows an example of a JSON-based instruction sent to the Edge On/Offloading Server.

```
{
        "ContainerImageName":"jsi/x_container_image_tcp",
        "Redeployment":"192.168.1.17",
        "DeploymentAggregate":"192.168.1.6",
        "DeploymentProcess":"192.168.1.6",
        "Port":{
                "Type":"tcp",
                "ContainerPort":"6790",
                "HostPort":"6790"
        },
        "InputElements":{
                "Threshold":"0.3",
                "FileName":"RAW_ACCELEROMETERS",
                "Y_column":"6",
                "Z_column":"7",
                "Interval":"5"
        }
}
```

Figure 22: An example of a JSON-based instruction sent to the Edge On/Offloading Server

The JSON-based instruction, shown in Figure 22, represents an abstraction of the application in order to perform a redeployment action at runtime. In this example, the container image name is `jsi/x_container_image_tcp`. The container instance running on the edge node with the IP address of `192.168.1.17` should be stopped. Moreover, a new container instance has to be started on a resource with the IP address of `192.168.1.6`. In order to instantiate this new container, the TCP port `6790` in the container should be mapped to port `6790` on the Docker host. This port will be employed to communicate the data aggregating part and the data processing part that both should be run in the new container instance. Furthermore, in this example, the application has its own different input elements namely `Threshold`, `FileName`, `Y_column`, `Z_column` and `Interval` along with their associated values.

The Java code for the implementation of deployment or reconfiguration API exposed by the Edge On/Offloading Server is also kept publically available on GitHub [52]:

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/ReDeployment.java

7. Edge node registration API exposed by the Edge On/Offloading Server

The Edge On/Offloading Server provides the edge node registration API, presented in Table 4, allowing the On/Offloading Client installed on every edge resource for the registration of edge node. When the Edge On/Offloading Server receives a request on the port number 30001 for the registration of an edge node, it consequently calls an interface exposed by the Autonomic Data Intensive Application Manager in order to provide the current state of the registered edge node and its features. Such information is required to be finally stored in the database called Cloud and Edge Resources Repository.

Table 4: Edge node registration API

Property	Description
API name	Edge node registration
API explanation	This API allows On/Offloading Clients to send requests for the registration of edge nodes.
API provider	Edge On/Offloading Server
API consumer	On/Offloading Client
Implementation	Java
State	Synchronous
API port number	30001

As shown in Figure 23, if the node registration is successfully performed, the return value will be 100. If the execution of the node registration failed due to an error, the return value will be 101.

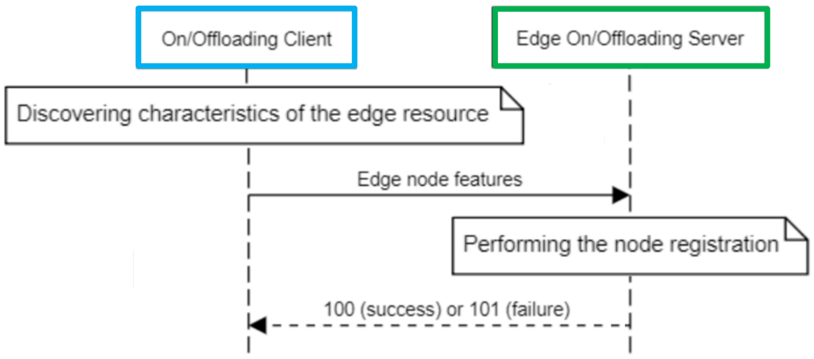
On/Offloading Client
Edge On/Offloading Server
Discovering characteristics of the edge resource
Edge node features
Performing the node registration
100 (success) or 101 (failure)

Figure 23: Request issued by the On/Offloading Client for the registration of edge node

In order to register an edge node, different characteristics of an edge resource are taken into account as follows:

- The Java code, shown in Figure 24, is the implementation of discovering the Operating System (OS) of the edge node. This part is executed by the On/Offloading Client running on the edge resource.

```
private String discoverOS(){
        String os = ""; // e.g. CentOS Linux release 7.4.1708
        try{
                String[] cmd = {"/bin/sh","-c","cat /etc/*-release"};
                Process p = Runtime.getRuntime().exec(cmd);
                p.waitFor();
                BufferedReader b = new BufferedReader(new
                                        InputStreamReader(p.getInputStream()));
                String line = "";
                while ((line = b.readLine()) != null){
                        if (line.contains("CentOS"))
                                os = line;
                        else if (line.contains("PRETTY_NAME"))
                        os = line.split("=")[1].replace("\"", "");//Ubuntu+Debian
                }
                b.close();
        }
        catch(Exception e){
                e.printStackTrace();
        }
        return os;
}
```

Figure 24: Discovering the Operating System of the edge node

- The Java code, shown in Figure 25, is the implementation of discovering CPU cores of the edge node. This part is executed by the On/Offloading Client running on the edge resource.

```
private String discoverCPUCores(){
        String cpucores = "1"; // defaul value
        String[] cmd = {"/bin/sh","-c","cat /proc/cpuinfo | grep -c processor"};
        try{
                Process p = Runtime.getRuntime().exec(cmd);
                p.waitFor();
                BufferedReader b = new BufferedReader(new
                                        InputStreamReader(p.getInputStream()));
                cpucores = b.readLine();
                b.close();
        }
        catch(Exception e){
                e.printStackTrace();
        }
        return cpucores;
}
```

Figure 25: Discovering CPU cores of the edge node

- The Java code, shown in Figure 26, is the implementation of discovering the Instruction Set Architecture (ISA) of the edge node. This part is executed by the On/Offloading Client running on the edge resource.

```
private String discoverISA(){
        String isa = "";
        try{
                Process p = Runtime.getRuntime().exec("uname -m");
                p.waitFor();
                BufferedReader b = new BufferedReader(new
                                        InputStreamReader(p.getInputStream()));
                isa = b.readLine();
                b.close();
        }
        catch(Exception e){
                e.printStackTrace();
        }
        return isa;
}
```

Figure 26: Discovering the Instruction Set Architecture (ISA) of the edge node

- The Java code, shown in Figure 27, is the implementation of discovering the total memory of the edge node. This part is executed by the On/Offloading Client running on the edge resource.

```
private String discoverMemTotal(){
        String memtotal = "";
        String[] cmd = {"/bin/sh","-c","awk '/MemTotal/ {print $2}' /proc/meminfo"};
        try{
                Process p = Runtime.getRuntime().exec(cmd);
                p.waitFor();
                BufferedReader b = new BufferedReader(new
                                        InputStreamReader(p.getInputStream()));
                memtotal = b.readLine();
                b.close();
        }
        catch(Exception e){
                e.printStackTrace();
        }
        return memtotal;
}
```

Figure 27: Discovering the total memory of the edge node

- The Java code, shown in Figure 28, is the implementation of discovering the total disk of the edge node. This part is executed by the On/Offloading Client running on the edge resource.

```
private String discoverDiskTotal(){
        String disktotal = "";
        String[] cmd = {
        "/bin/sh","-c","df -P | awk 'NR>2 && /^\/dev\//{sum+=$2}END{print sum}'"};
        try{
                Process p = Runtime.getRuntime().exec(cmd);
                p.waitFor();
                BufferedReader b = new BufferedReader(new
                                        InputStreamReader(p.getInputStream()));
                disktotal = b.readLine();
                b.close();
        }
        catch(Exception e){
                e.printStackTrace();
        }
        return disktotal;
}
```

Figure 28: Discovering the total disk of the edge node

- The Java code, shown in Figure 29, is the implementation of discovering the boot time of the edge node. This part is executed by the On/Offloading Client running on the edge resource.

```
private String discoverBootTime(){
        String boottime = "";
        String[] cmd = {"/bin/sh","-c","cat /proc/stat | grep btime"};
        try{
                Process p = Runtime.getRuntime().exec(cmd);
                p.waitFor();
                BufferedReader b = new BufferedReader(new
                                        InputStreamReader(p.getInputStream()));
                boottime = b.readLine();
                if(!boottime.equals(""))
                        boottime = boottime.split(" ")[1];
                b.close();
        }
        catch(Exception e){
                e.printStackTrace();
        }
        return boottime;
}
```

Figure 29: Discovering the boot time of the edge node

8. Conclusions

Recently, the perspective of IoT-based and Big Data driven environments has significantly evolved as these types of applications are becoming more and more time-sensitive, deployed at decentralised locations and easily influenced by the varying workload at runtime. Therefore, a promising paradigm is emerging from previously centralised computation to distributed edge computing framework to address such challenging scenarios.

Container-based virtualisation technology can be seen as enablers of edge computing scenarios. This is because, the deployment of containerised services on edge nodes is faster and more efficiently than using VMs. Such lightweight container technology combined with microservice architectural approach provides agility in deploying and running applications. Therefore, containerised microservices may be advantageous especially during any instantiation and termination of services at the extreme edge of the network. Along these lines, this deliverable describes the adoption of microservice architectural approach according to the necessity of using container-based virtualisation. This deliverable also presents the state-of-the-art edge computing frameworks proposed to handle highly dynamic workloads at runtime.

Despite all benefits of edge computing paradigm, in conditions where the execution environment may dynamically change (e.g. a drastic increase suddenly appears in the workload intensity), providing a platform able to keep the running application appropriately operating is essential. To this end, the PrEstoCloud solution tracks dynamic changes of the operational environment at the extreme edge of the network, and hence it detects situations when an edge node is no longer capable of providing storing or computing operations. Consequently, the running service will be offloaded from the edge node to the cloud-based infrastructure through a method called Mobile Offloading Processing. In this regard, the PrEstoCloud solution proposes two key software components: (i) Edge On/Offloading Server and (ii) On/Offloading Client, thoroughly explained in this deliverable in detail according to Task 3.4 (Mobile offload processing).

References

[1] Raspberry Pi 3 Model B, https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[2] Yiannis Verginadis, Iyad Alshabani, Gregoris Mentzas and Nenad Stojanovic, "PrEstoCloud: Proactive Cloud Resources Management at the Edge for Efficient Real-Time Big Data Processing", 7th International Conference on Cloud Computing and Services Science (CLOSER 2017), 24-26 April 2017.

[3] Joe Stubbs, Walter Moreira, and Rion Dooley, "Distributed Systems of Microservices Using Docker and Serfnode", In Proceedings of the 7th International Workshop on Science Gateways (IWSG). IEEE, Budapest, 2015, pp. 34-39.

[4] Johannes Thones, "Microservices," IEEE Softw., vol. 32, no. 1, p. 116, Jan. 2015.

[5] Anup Patel, Mai Daftedar, Mohamed Shalan, and M. Watheq El-Kharashi, "Embedded hypervisor Xvisor: A comparative analysis", In Proc. of the 23rd Euromicro International Conference on Parallel, Distributed and NetworkBased Processing (PDP 2015), 2015.

[6] Valery V. Trofimov, Vladimir I. Kiyaev, and Stanislav M. Gazul, "Use of virtualization and container technology for information infrastructure generation", In 2017 IEEE International Conference on Soft Computing and Measurements (SCM), IEEE, 2017, pp. 788-791.

[7] Salman Taherizadeh, Andrew Jones, Ian Taylor, Zhiming Zhao, and Vlado Stankovski. Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. Journal of Systems and Soware, 136:19–38, 2018. doi: https://doi.org/10.1016/j.jss.2017.10.033.

[8] Salman Taherizadeh and Vlado Stankovski. Auto-scaling applications in edge computing: Taxonomy and challenges. In Proc. of the International Conference on Big Data and Internet of Thing, pages 158–163, London, United Kingdom, 2017. ACM. doi: 10.1145/3175684.3175709.

[9] Salman Taherizadeh and Vlado Stankovski, Dynamic multi-level auto-scaling rules for containerized applications, The Computer Journal, Oxford University Press, 2018. doi: 10.1093/comjnl/bxy043

[10] Google container engine, https://cloud.google.com/container-engine/

[11] Amazon EC2 Container Service, https://aws.amazon.com/ecs/

[12] Iain Emsley and David De Roure, "A framework for the preservation of a Docker container", Digital Curation Centre, 2017.

[13] S.P. Polyakov, A.P. Kryukov, and A.P. Demichev, "Docker Container Manager: A Simple Toolkit for Isolated Work with Shared Computational, Storage, and Network Resources", Journal of Physics: Conference Series (Vol. 955, No. 1, p. 012039). IOP Publishing, 2018.

[14] Docker Hub repository, https://hub.docker.com/

[15] Pushing and Pulling Docker Images,
https://cloud.google.com/container-registry/docs/pushing-and-pulling

[16] Dockerfile reference, https://docs.docker.com/engine/reference/builder/

[17] Docker, https://www.docker.com/

[18] CoreOS, https://coreos.com/

[19] Kubernetes, https://kubernetes.io/

[20] OpenShift Origin, https://www.openshift.org/

[21] Swarm, https://docs.docker.com/swarm/

[22] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. "Architectural principles for cloud software." ACM Transactions on Internet Technology (TOIT) 18, no. 2 (2018): 17.

[23] The LightKone project, https://www.lightkone.eu/

[24] The Open Edge Computing project, http://openedgecomputing.org/

[25] Peng Liu, Dale Willis, and Suman Banerjee, "Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge", In Edge Computing (SEC), IEEE/ACM Symposium on, pp. 1-13. IEEE, 2016.

[26] Ketan Bhardwaj, Pragya Agrawal, Ada Gavrilovska, and Karsten Schwan, "Appsachet: Distributed app delivery from the edge cloud", In International Conference on Mobile Computing, Applications, and Services, pp. 89-106. Springer, Cham, 2015.

[27] Yuvraj Sahni, Jiannong Cao, Shigeng Zhang, and Lei Yang. "Edge Mesh: A new paradigm to enable distributed intelligence in Internet of Things." IEEE Access 5 (2017): 16441-16458.

[28] Mathew Ryden, Kwangsung Oh, Abhishek Chandra, and Jon Weissman, "Nebula: Distributed edge cloud for data intensive computing", In Cloud Engineering (IC2E), 2014 IEEE International Conference on, pp. 57-66. IEEE, 2014.

[29] Brian B. Amento, Bharath Balasubramanian, Robert J. Hall, Kaustubh Joshi, Gueyoung Jung, and K. Hal Purdy, "FocusStack: Orchestrating Edge Clouds Using Location-Based Focus of Attention", In Proceedings of IEEE/ACM Symposium on Edge Computing, 2016, pp. 179–191.

[30] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S. Nikolopoulos, "Challenges and opportunities in edge computing", In Smart Cloud (SmartCloud), IEEE International Conference on, pp. 20-26. IEEE, 2016.

[31] Jinlai Xu, Balaji Palanisamy, Heiko Ludwig, and Qingyang Wang, "Zenith: Utility-aware resource allocation for edge computing", In Edge Computing (EDGE), 2017 IEEE International Conference on, pp. 47-54. IEEE, 2017.

[32] Mayra Samaniego and Ralph Deters, "Supporting IoT Multi-Tenancy on Edge Devices", In Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2016 IEEE International Conference on, pp. 66-73. IEEE, 2016.

[33] Muhammad Shiraz and Abdullah Gani, "A Lightweight Active Service Migration Framework for Computational Offloading in Mobile Cloud Computing", The Journal of Supercomputing, vol. 68, no. 2, pp. 978–995, 2014.

[34] Wenlu Hu, Ying Gao, Kiryong Ha, Junjue Wang, Brandon Amos, Zhuo Chen, Padmanabhan Pillai, and Mahadev Satyanarayanan, "Quantifying the impact of edge computing on mobile applications", In Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems, p. 5. ACM, 2016.

[35] Xiaolong Xu, Shucun Fu, Qing Cai, Wei Tian, Wenjie Liu, Wanchun Dou, Xingming Sun, and Alex X. Liu, "Dynamic Resource Allocation for Load Balancing in Fog Environment", Wireless Communications and Mobile Computing 2018 (2018).

[36] Yifan Yu, "Mobile edge computing towards 5G: Vision, recent progress, and open challenges." China Communications 13, no. 2 (2016): 89-99.

[37] Yaser Jararweh, Fadi Ababneh, Abdallah Khreishah, and Fahd Dosari, "Scalable cloudlet-based mobile computing model", Procedia Computer Science 34 (2014): 434-441.

[38] Ali Gholami, "Security and privacy of sensitive data in cloud computing", PhD diss., KTH Royal Institute of Technology, 2016.

[39] Yevgeniya Sulema, Noam Amram, Oleksii Aleshchenko, Olena Sivak, "Quality of Experience Estimation for WebRTC-based Video Streaming", 24th International Conference European Wireless EW2018, May 2-4, 2018, Catania, Italy.

[40] Salman Taherizadeh, Blaz Novak, Marija Komatar, and Marko Grobelnik, "Real-Time Data-Intensive Telematics Functionalities at the Extreme Edge of the Network: Experience with the PrEstoCloud Project", The 42nd Annual IEEE International Conference on Computers, Software and Applications (COMPSAC 2018), Tokyo, Japan, 2018. DOI: 10.1109/compsac.2018.10288

[41] Salman Taherizadeh, Ian Taylor, Andrew Jones, Zhiming Zhao, and Vlado Stankovski. A network edge monitoring approach for real-time data streaming applications. In Proc. of the 13th International Conference on Economics of Grids, Clouds, Systems and Services (GECON 2016), pages 293–303, Athens, Greece, 2016. Springer.

[42] Sample Java code to implement the aggregating data part based on TCP,

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/AggregateData_TCP.java

[43] Sample Java code to implement the aggregating data part based on TCP,

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/AggregateData_UDP.java

[44] J. Wahlström, I. Skog, and P. Händel, "Smartphone-based vehicle telematics: A ten-year anniversary," IEEE Trans. Intell. Transp. Syst., 18(10), 2017, pp. 2802-2825.

[45] Sample Java code to implement the processing data part based on TCP,

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/ProcessData_TCP.java

[46] Sample Java code to implement the processing data part based on UDP,

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/ProcessData_UDP.java

[47] Shell file is executed when a container providing a TCP-based service is instantiated,

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/start_TCP.sh

[48] Shell file is executed when a container providing a UDP-based service is instantiated,

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/start_UDP.sh

[49] Dockerfile used to create container image for TCP-based services,

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/Dockerfile_TCP

[50] Dockerfile used to create container image for TCP-based services,

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/Dockerfile_UDP

[51] Java code to implement the On/Offloading Client,

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/On_Offloading_Client.java

[52] Deployment or reconfiguration API exposed by the Edge On/Offloading Server,

https://github.com/salmant/PrEstoCloud/blob/master/OnOffloading/ReDeployment.java