DECENT: A Decentralized Configurator for Controlling Elasticity in Dynamic Edge Networks

ILIR MURTURI and SCHAHRAM DUSTDAR, Distributed Systems Group, TU WIEN, Austria

Recent advancements in distributed systems have enabled deploying low-latency and highly resilient edge applications close to the IoT domain at the edge of the network. The broad range of edge application requirements combined with heterogeneous, resource-constrained, and dynamic edge networks make it particularly challenging to configure and deploy them. Besides that, missing elastic capabilities on the edge makes it difficult to operate such applications under dynamic workloads. To this end, this article proposes a lightweight, self-adaptive, and decentralized mechanism (DECENT) for (1) deploying edge applications on edge resources and on premises of Edge-Cloud infrastructure and (2) controlling elasticity requirements. DECENT enables developers to characterize their edge applications by specifying elasticity requirements, which are automatically captured, interpreted, and enforced by our decentralized elasticity interpreters. In response to dynamic workloads, edge applications automatically adapt in compliance with their elasticity requirements. We discuss the architecture, processes of the approach, and the experiment conducted on a real-world testbed to validate its feasibility on low-powered edge devices. Furthermore, we show performance and adaptation aspects through an edge safety application and its evolution in elasticity space (i.e., cost, resource, and quality).

CCS Concepts: • **Information systems** → *Computing platforms*; • **Computer systems organization** → **Edge Computing;**

Additional Key Words and Phrases: Elasticity, edge computing, edge-cloud, Internet of Things (IoT)

ACM Reference format:
Ilir Murturi and Schahram Dustdar. 2022. DECENT: A Decentralized Configurator for Controlling Elasticity in Dynamic Edge Networks. *ACM Trans. Internet Technol.* 22, 3, Article 78 (August 2022), 21 pages.
https://doi.org/10.1145/3530692

1 INTRODUCTION

The **Internet of Things (IoT)** has prominently diffused into society in recent years. A wide range of services are designed on top of IoT technologies in various industries such as Industrial Manufacturing, Healthcare, and Smart Buildings. Simultaneously, cloud-based solutions are no longer sufficient to satisfy the stringent requirements (i.e., low latency and high availability) of the safety-critical and real-time IoT services. To overcome such a gap between cloud and IoT entities, new

Ilir Murturi and Schahram Dustdar contributed equally to this research.

Research partially supported by the "Smart Communities and Technologies (Smart CT)" and it has received funding from the EU's Horizon 2020 Research and Innovation Programme under grant agreement No. 871525. EU web site for Fog Protect: https://fogprotect.eu/.

Authors' address: I. Murturi and S. Dustdar, Distributed Systems Group, TU WIEN, Argentinierstrasse 8, Vienna, Austria, 1040; emails: {imurturi, dustdar}@dsg.tuwien.ac.at.

1533-5399/2022/08-ART78 $15.00
https://doi.org/10.1145/3530692

computational resources named *edge devices* are being introduced at the edges of networks providing low-latency services and enhancing privacy within IoT infrastructures [25]. Edge devices are essentially low-powered computers located at edge networks—closer to the data source, respectively, to IoT domains (i.e., sensors, actuators, etc.). In this context, edge devices can process data streams streamed into an IoT system. Notably, to achieve such an aim, we require deploying and running various analytic or decision functions at edge networks [8]. For instance, a scheduler decides whether the pumped data must be processed at an edge network or forwarded to a cloud infrastructure. Thus, many operational and business challenges can be solved by running decision-making functions on edge resources.

As a newly introduced paradigm, Edge Computing [25] is a key enabler for IoT proliferation. In contrast to cloud infrastructures, edge networks are resource-constrained environments. Edge networks essentially are environments where a set of heterogeneous edge devices are connected in a peer-to-peer manner. Such devices usually have limited resources, referring to their computational and storage capabilities. A wide range of available resources at the edge have introduced new opportunities such as deploying low-latency, privacy-aware, and resilient *edge applications* (e.g., IoT applications). Besides many benefits introduced by edge devices, analyzing high-volume IoT data streams on a single device through monolithic applications poses many limitations and a set of challenges in terms of processing capabilities, storage, energy, and communication bandwidth. To that end, modern applications are no longer monolithic [2]; such edge applications (i.e., *services*) are divided into a set of independently deployable software components (i.e., *microservices*) and distributed over edge resources or on premises of Edge-Cloud infrastructure.[1] Similarly, resource management techniques need to be designed as decentralized systems to run in resource-constrained environments. Thus, this brings completely new challenges where novel lightweight resource management techniques are needed to fully utilize available resources at edge networks. Nonetheless, the broad range of requirements concerning latency, **Quality of Service (QoS)**, or fault tolerance, combined with edge networks' heterogeneous and dynamic nature, make it particularly challenging to manage, configure, deploy, and operate such applications.

Over the past few years, researchers have been widely focused on proposing multiple resource allocation techniques at the edge [24]. However, less attention is given to providing elastic features at the edge [14, 17]. In most cases, elasticity refers to a system's capability to adapt to workload changes by (de)provisioning resources in an automatic manner [13]. Resource demands for a particular running application or component may change over time. Consequently, this may cause poor overall performance and higher latency than the expected response time. For instance, consider a scenario where a health application running in an edge network (e.g., smart home) monitors residents' health through processing data streams created by the users' smartwatches. At the time t_0, a single edge device has sufficient resources to monitor only one resident's health. At the time t_1, there is more than one resident, and the edge device may not have enough resources to process produced data for all residents. To avoid such situations, edge applications should scale over edge resources or on premises of Edge-Cloud infrastructure. Therefore, introducing *elasticity* features at the edge is crucial.

The elasticity concept is heavily related and used in cloud computing, and often is considered one of the main features of the cloud paradigm [7]. In Edge Computing, very few works propose methods for controlling application elasticity [10, 12, 23, 27]. Current approaches exhibit several limitations, such as that they are built as centralized systems, are application specific, and enable application scaling only by considering hardware resources and their capacity to scale.

[1]We consider the Edge-Cloud infrastructure as three-tier architecture composed of edge, fog, and cloud entities [9].

Furthermore, centralized approaches are sensitive to edge system characteristics (i.e., resource constrained, dynamic, and uncertain). Thus, edge networks' dynamic nature requires continuously re-evaluating placement decisions for edge functions. Nevertheless, in our conception, besides *resource* requirements, elasticity in three-tiered infrastructures should also target their relations with the different types of *costs* and *quality*.

To address the aforementioned challenges, we propose a lightweight framework called **Decentralized Configurator for Controlling Elasticity in Dynamic Edge Networks (DECENT)** and its runtime mechanism for controlling elasticity requirements in edge applications. DECENT enables deploying and scaling edge applications in dynamic edge networks and on premises of Edge-Cloud infrastructures. Essentially, the developer defines application requirements, elasticity requirements, and a scaling model for each edge application and its components. DECENT interprets these requirements, deploys components in the three-tier architecture, and enforces various scaling operations at runtime to fulfill edge application demands. The system we propose enables easy configuration, deployment, and operation of edge applications on top of heterogeneous edge infrastructure (i.e., edge network). Furthermore, DECENT is a self-adaptive and decentralized mechanism that can be easily deployed and run on low-powered edge devices.

Our concrete contributions are as follows:

- We enable application developers or domain experts to specify high-level elasticity requirements for their edge applications in a declarative way. Besides that, the user specifies the deployment and scaling model inherent to a specific infrastructure configuration. To specify edge application elastic requirements, we consider the declarative language called ***Simple Yet Beautiful Language (SYBL)*** [6]. We extend the language and focus on developing novel constraints and enforcement strategies to support edge application characteristics. In our conception, besides *resource* requirements, elasticity in three-tiered architectures should also target their relations with the different types of *costs* and *quality*.
- We extend the prototype of [21] with a lightweight mechanism that enables deploying and controlling the elasticity of edge applications in a decentralized manner at an edge network. Edge devices that run application components capture and interpret their elastic requirement through *Elasticity Interpreters* and report to the configurator device whether the requirements are violated. The configurator device takes action and re-configures the application to meet the specified elastic demands.
- To validate the approach's feasibility, we perform an experimental evaluation that shows that an edge application and its components can easily scale and be controlled by the mechanisms deployed at the edge of a network. Our prototype is evaluated on a real-world testbed composed of several low-powered edge devices.

The rest of the article is structured as follows. Section 2 gives an overview of the platform along with a running example of the article. Related work is considered in Section 3. Section 4 describes edge applications and system modeling, along with describing application requirements. In Section 5, we describe in detail the DECENT framework, the processes, and details regarding prototype implementation. Evaluation and results are presented in Section 6. Finally, Section 7 concludes the article and outlines future work directions.

2 BACKGROUND AND RUNNING EXAMPLE

This section outlines the domain for which we have developed our system. We give a short overview of our previous work on top of which the proposed approach is built. Afterward, we present our motivational scenario through a running example.

2.1 Background

This article substantially extends our previous works [21] with a new elasticity controlling module and its lightweight runtime mechanism. The former introduces the platform and system architecture enabling automatic discovery of heterogeneous resources (i.e., computational, sensing, context data) in edge networks [20].

One prominent approach that has recently emerged is to combine edge, fog, and cloud infrastructures to enable providing low-latency services [8]. Edge and fog paradigms provide almost similar features. Both paradigms foresee enabling more computation resources near end-users and IoT domains. However, the most significant difference between the two tiers is administrative differences and responsibilities. We acknowledge that Edge Computing means different things to different people; we envision Edge Computing as a bridge between IoT devices and the nearest edge device to a user. Furthermore, fog devices may provide much more powerful resources and services for larger geographical areas. For instance, smart transportation systems may benefit from connecting and processing vehicle data in fog infrastructure. Nonetheless, both paradigms aim to provide low-latency services since end devices are closer to the source where the data is produced and consumed.

Furthermore, the three-tier infrastructure architecture shows a seamless opportunity for deploying various applications (e.g., industrial, health, etc.) where low latency, QoS, reliability, and scalability are critical requirements. This enables the distribution of application components over edge, fog, and cloud resources. However, in the past few years, researchers in the field of edge and fog computing have been mostly focused on proposing multiple centralized techniques for scheduling, controlling, and monitoring IoT applications deployed at the edge. In fact, such functions are deployed on powerful devices such as local servers or cloud devices [28]. Nonetheless, as we explore new IoT systems and the heterogeneous and dynamic nature of edge networks, distributing system components among various computation entities becomes an increasingly inevitable requirement. As a result, shifting various system functions closer to edge networks and dynamically placing them in the most suitable devices is crucial (see Section 5). Thus, deploying decision mechanisms at the edge in a decentralized manner makes edge networks *autonomous environments* and less dependent on centralized devices. To that end, in our previous work, we proposed an efficient approach that solves the placement problem of the configurator on the most suited (e.g., powerful) edge device in a given dynamic edge network [21].

The latter introduces the technical framework and a solution to build and organize devices in edge networks such that the resource discovery complexity can be handled [21]. The proposed framework implements the configurator placement approach and enables system designers to define and configure their edge networks. More specifically, edge devices in edge networks are organized into clusters. Each formed cluster has a cluster coordinator and one global coordinator (i.e., configurator) of an edge network. Both coordinators aim to provide various functionalities to support resource discovery, and they act similarly as *superpeers* [15] at the edge. All coordinators are placed dynamically on the most suited edge devices in an edge network. Nevertheless, the dynamic nature of edge networks necessitates continuously re-evaluating placement decisions for coordinators. Thus, using a self-adaptive and decentralized configurator aims to solve such challenges at dynamic edge networks. In this article, our proposed approach enables and supports the execution of edge applications on various edge networks and maintains their correct functionality throughout the execution time. The proposed elasticity control mechanism maintains the correct functionality of edge applications by considering multiple elastic perspectives (i.e., quality, cost, and resources). The proposed elastic mechanism is an extension of the introduced framework in [21].

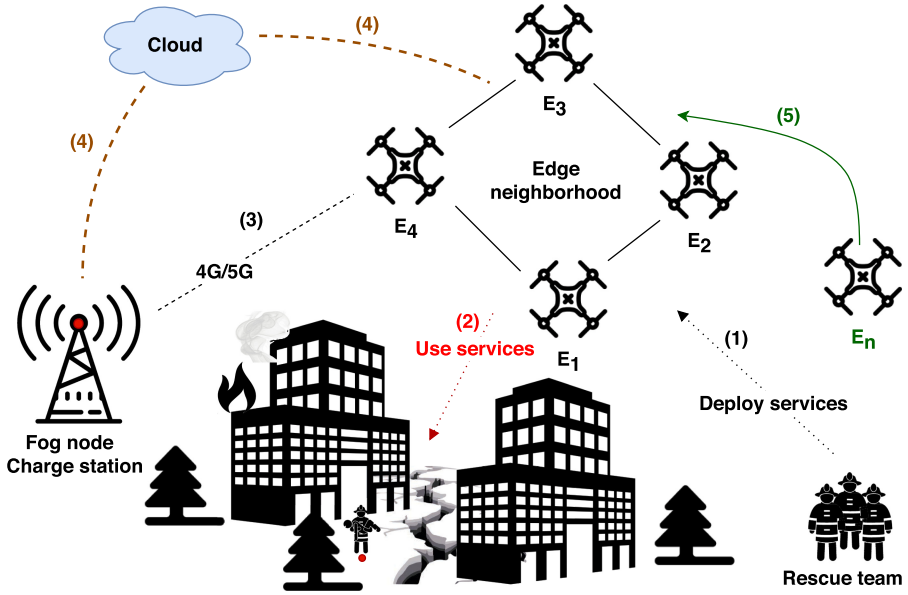

Fig. 1. The lost-person service.

2.2 Running Example

To motivate our subsequent discussion, we consider emergencies such as natural disasters (e.g., earthquakes, fires, floods) in the city. Emergencies like earthquakes may affect various city zones, which can damage infrastructure, cause injury or loss of life, and trap people under buildings. In such situations, time is valuable, and drones may be used to analyze the entire situation and help rescue teams find and communicate with victims under a collapsed building. In this scenario, we consider multiple connected drones (i.e., form an *edge neighborhood*) flying over the city's affected areas aiming to provide services for the rescue team in finding victims under a collapsed building. Each drone (i.e., edge device) is equipped with various computation capabilities and integrated sensors (e.g., radar sensors, infrared cameras, electronic nose, etc.). We consider that drones are multi-purpose devices where the rescue teams may request to deploy various services depending on the emergency. Meanwhile, base stations may provide computational and storage capabilities (i.e., fog devices) and provide docker charge stations for charging drones. At the same time, cloud capabilities may be used to store data for long terms.

Referring to the situation illustrated in Figure 1, we assume that a rescue team deploys (1) the lost-person service (i.e., edge safety application) in the affected area. Such a service aims at helping rescue teams solve missing person cases faster by finding their location in the affected zones (2). The service is dependent on camera resources, which are integrated into various drones. Specifically, the lost-person service consists of components responsible for specific tasks (i.e., front-end, image processing, generating results, storing results, etc.). The service takes as input images provided by the flying drones in the affected area. Each drone every second generates various images to be processed by the service. However, with the increasing number of drones, the number of generated images is increased (5). To that end, application components may require more computing resources to process images to fulfill application requirements. For instance, consider that the front-end component that accepts drones' images has a response time requirement that should be less than 100 ms. When the response time requirement is violated (e.g., 100 ms), the service component must scale to multiple instances on edge or on premises (3–4) to meet the desired service

quality. Thus, it is evident that to meet service demands at runtime and to avoid resource over-provisioning/under-provisioning, we require a lightweight mechanism that dynamically controls application elasticity at the edge. Furthermore, we assume that the service running on an edge neighborhood is accessible by users within the range covered by devices.

3 RELATED WORK

Research efforts associated with the elasticity at the edge are still at a relatively early development stage. Elasticity features in edge infrastructures mostly have been focused on scaling up/down *resources* to meet application demands. In some approaches, tasks/services are reallocated when devices are overloaded [26]. However, such practices, in turn, incur an overhead of resource usage, increased cost, and increased energy consumption. Even though resource over-provisioning can be considered feasible in resource-rich environments such as the cloud, such an assumption is highly impractical in resource-constrained edge networks. More specifically, reserving resources more than needed to support the intended task workload wastes available resources.

Very few approaches address these challenges at the edge. Furst et al. [12] introduce a new framework that enables services to self-adapt and meet the current service demands of their **Service-Level Objectives (SLOs)**. A novel programming model called **Diversifiable Programming (DivProg)** uses function annotations as an interface between the service logic, its SLOs, and the execution framework to achieve such an adaption dynamically. Essentially, a third-party execution framework captures service configuration given by the developer through DivProg, interprets, and scales services that conform to changing SLOs. Tseng et al. [27] provide a lightweight autoscaling mechanism for fog computing in industrial applications. Lujic and Truong [16] propose a novel, holistic approach for architecting elastic edge storage services, featuring three aspects such as data/system characterization (e.g., metrics, key properties), system operations (e.g., filtering, sampling), and data processing utilities (e.g., recovery, prediction). The authors [23] discuss how applications for a fog infrastructure can be packaged into containers and act elastically. Their approach is built on top of the container orchestration tool Kubernetes and extends it to the fog. In [29], the authors investigate the benefits of virtualization to move and redeploy mobile components to fog devices near the targeted end devices. By using geometric monitoring, the approach dynamically scales and provisions the resources for the fog layer. Furthermore, several edge platforms such as EdgeX Foundry,[2] AWS IoT Greengrass,[3] or Google IoT Edge[4] promise to bridge the gap between IoT and the cloud by providing a flexible runtime for applications running at the edge.

Any Edge-Cloud system's goal is to hide the complexity of edge applications' deployment and operations in heterogeneous edge networks and enable developers to specify application requirements in a declarative way. A **domain-specific language (DSL)** specifies the high-level constraints of edge applications, such as QoS, application criticality, and elasticity requirements. The DSL essentially makes it easy for users to develop these specifications. Understanding the current and future requirements of edge applications from various domains remains a prominent challenge. A platform for the described IoT scenarios (e.g., as in our running example) needs to hide this operational complexity from application developers. In particular, programmers should not have to worry about the distribution of data and edge or cloud resources' heterogeneous capabilities. Developers should be able to express the context in which applications are allowed to run and the elasticity requirements in a high-level way [7]. The platform should then take care of resource

[2]EdgeX Foundry, https://www.edgexfoundry.org/.
[3]AWS IoT Greengrass, https://aws.amazon.com/greengrass/.
[4]Google IoT Edge, https://cloud.google.com/solutions/iot.

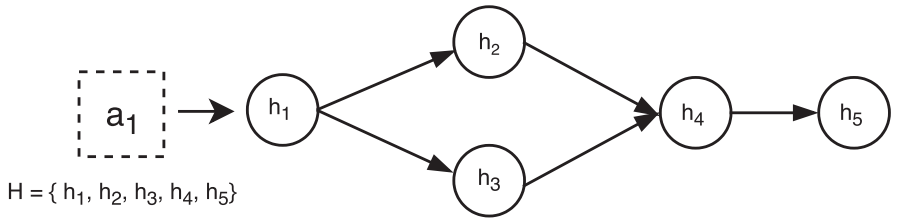

Fig. 2. Edge application model.

provisioning and data movement. However, this requires that the programming model and API are intuitive for developers but expressive enough to help the execution platform make runtime decisions on scheduling edge application components. In this article, we consider the SYBL [6] to specify elastic requirements in terms of resource, cost, and quality. The SYBL enables developers to specify elasticity requirements (i.e., constraints, monitoring, strategies, and priorities) at design time and enables scaling edge applications in an elasticity space. Nevertheless, **Service-Level Objectives for Next-Generation Cloud Computing (SLOC)** is another novel elasticity framework, which promotes a novel performance-driven, SLO-native approach to cloud computing, respectively, Edge-Cloud environments [22]. The new SLO elasticity policy language considers similar elastic dimensions (i.e., resource, cost, and quality) as the SYBL language.

Finally, our work is an effort to advance Edge Computing platforms' current state and enable more straightforward configuration, deployment, and operation of edge applications on top of heterogeneous edge infrastructure. The above-mentioned systems are extremely limited in their operational capabilities and lack of self-adaptive mechanisms required in dynamic edge and IoT settings. In essence, such systems assume static configurations that do not change over time, provide no way to specify elasticity or QoS requirements, and do not have a mechanism to enact them. Our proposed approach aims to bridge this gap and ensure multi-dimensional elasticity control (i.e., cost, resources, and quality) for fulfilling edge application demands deployed on Edge-Cloud infrastructure. It enables edge applications and their components to adapt in response to dynamic changes in their workload. Finally, we allow developers to easily define elasticity requirements captured and executed by our lightweight mechanism in a decentralized manner.

4 EDGE MODELING AND ELASTIC REQUIREMENTS

This section formally defines the concepts of edge applications, the edge system, the deployment and scale policy, and elasticity requirements. First, we model edge applications and the system. Then, we describe application deployment and scaling models in Edge-Cloud architectures. Finally, we extensively explain application elasticity requirements, which enable developers to characterize their edge applications.

4.1 Edge Application and System Model

A service-based edge application a_i can be described as a set of components $H=\{h_1, h_2, \ldots, h_m\}$ divided by the developer prior to deployment. In fact, to enable executing such components on low-powered devices and for better utilization of distributed resources dividing the edge application into components is crucial. To this end, edge application components may be distributed and executed on various available devices such that their resource requirements are fulfilled upon deployment. Each component h_i can be modeled as a **Directed Acyclic Graph (DAG)** where vertices represent components and edges represent their dependencies as given in Figure 2.

Each component h_i is a black box, representing a set of instructions aiming to provide specific functionalities. Various hardware-related requirements characterize each component (e.g., processing, memory, storage) and latency requirements between components [4]. Moreover, components are characterized by varying workloads, and their deployment should also be properly adapted at runtime. To that end, for each component, we have the following: (1) *edge application resource requirements*, (2) *elasticity requirements*, and (3) *deployment and scaling policy*. In the following subsections, we have discussed both application requirements in detail.

The Edge-Cloud architecture consists of edge infrastructures (i.e., multiple edge devices connected in a peer-to-peer manner forms an edge infrastructure), fog infrastructure, and cloud infrastructure. As mentioned in Section 2, our approach is built on top of an edge network, which is built as a **Distributed Hash Table (DHT)** network [19]. We assume that every edge device trusts all devices to establish a direct communication link; they all belong to the same local administrative domain. Furthermore, in this work, our primary focus resides at edge infrastructures, while we assume that the fog and cloud infrastructures are considered as **Infrastructure as a Service (IaaS)** [18]. We assume that the system designer configures the edge network to connect to the IaaS services.

Executing components on heterogeneous environments (i.e., edge tier, fog tier, or cloud tier) is crucial. For instance, an application with multiple components can scale on multiple instances running on different locations in either edge or cloud, depending on current demands and the application's constraints. To overcome the challenges introduced with heterogeneous environments, we consider Docker[5] as our homogeneous application runtime platform that follows the *"run once, run anywhere"* model. The Docker platform represents a lightweight, stand-alone, executable package that contains everything needed to run the specifically added component. The application runtime is essentially responsible for executing edge applications (i.e., container based) on edge devices or on premises. Thus, to deploy edge applications in Edge-Cloud infrastructure, components are packaged in individual Docker containers.

4.2 Deployment and Scaling Models

In our conception, edge applications can be thought of as a set of deployable software components running on premises of the Edge-Cloud infrastructure. Thus, edge application components can be deployed and scaled according to the following models [3]:

- `Everything in the Cloud:` The application components are deployed in the cloud. Essentially, this model is suitable when applications require significant computation and storage capabilities.
- `Everything in the Edge:` In this model, application components are distributed across available devices at the edge of the network. In essence, edge networks may exist in different settings starting from private edge networks (e.g., smart home) to enterprise edge networks (e.g., industries, health, etc.), as well as the public edge networks (e.g., smart city). Furthermore, researchers exchangeably use two terms for the available devices at this layer, such as edge and fog devices [8]. This study refers to an edge device as low powered and resource constrained (i.e., lower computation and storage capabilities). In contrast, a fog device is much more powerful than edge devices and less than the cloud.
- `Hybrid Edge-Cloud:` In this model, application components are distributed across available resources at the edge, fog, and cloud. Essentially, the hybrid model enables deploying and executing applications with low-latency requirements and resource-demanding processes.

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

Each mentioned model has different characteristics in terms of cost, latency, privacy, and other quality properties [3].

4.3 Elasticity Requirements

Elasticity properties at the edge are crucial for executing edge applications and fulfilling their dynamic resource demands. In Edge-Cloud architectures, elasticity targets not only resources and their capacity to scale but also their relationships with various forms of costs and quality [7]. In this context, multiple stakeholders may be involved in specifying elastic requirements. For instance, the developer could specify that the latency between application components must not reach 20ms without carrying out how many resources should be used to achieve the desired state. An edge network provider could specify its resource utilization schema; for example, when overall utilization at the edge is higher than 90%, it enables scaling applications toward fog or cloud infrastructure. To enable such a feature, we consider a declarative language called SYBL to allow users to specify an edge application's elasticity requirements at the design time [6]. We extend the language and focus on developing novel constraints and enforcement strategies to support edge application characteristics running on Edge-Cloud infrastructures. In addition, we extend and optimize the language runtime engine to support controlling Docker-based edge applications and enable execution on low-powered edge devices. We provide a time-based mechanism that analyzes workloads generated by incoming requests to optimize and avoid unnecessary scaling operations. More specifically, the time-based mechanism controls for a few seconds (i.e., a configurable value, e.g., 20 seconds) if the increased load on a particular component is handled without executing any scaling operation. The feature mentioned above is useful in situations when the increased or decreased load in a component is occasional and not persistent.

SYBL enables users (i.e., developer or system user) to specify application elasticity requirements in a declarative way represented in the form of (1) *monitoring* (i.e., specifying which metrics to monitor), (2) *constraints* (i.e., specifying the limits in which the monitored metrics can oscillate), (3) *strategies* (i.e., specifying actions to be followed in case the constraint is violated or becomes true), and (4) *priorities* (i.e., specifying constraints with higher priority than the other ones). The user can specify elastic requirements at different levels of edge application. Thus, elasticity controls can be achieved at the (1) *edge application level* (i.e., specifying high-level application elastic requirements) and (2) *edge component level* (i.e., specifying low-level application elastic requirements). Listing 1 shows an example of elasticity requirements specified by the user. At the edge application level, the user may specify the maximum cost allowed for the entire edge application executed in an Edge-Cloud infrastructure. The user could specify that the application needs to scale down when the cost is high and CPU usage is below 20%. Or, when the cost is below the predefined value (e.g., 5 euros) and the CPU usage is higher than 80%, the application needs to scale up. At the edge component level, for instance, elasticity requirements from the developer side can be applied regarding the quality, such that, e.g., if the edge device battery is less than 10%, the component must scale up to avoid application failure.

The SYBL elastic requirements can be easily injected/integrated into various description languages. For instance, the elastic requirements can be easily integrated into the cloud application description language TOSCA standard,[6] docker-compose files (i.e., YAML), or JavaScript Object Notation (JSON), or specified separately through XML descriptions. In the current version of our prototype, we specify edge application elastic requirements in a JSON file. Nevertheless, future

[6]OASIS, Topology and Orchestration Specification for Cloud Applications (TOSCA), http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html.

```
#ApplicationLevel.AppID
Mo1: MONITORING cpuUsage = cpu.usage
Mo2: MONITORING memUsage = mem.usage
Co1: CONSTRAINT totalCost > 5Euro AND cpu.usage < 20
Co2: CONSTRAINT totalCost < 5Euro AND cpu.usage > 80
St1: STRATEGY CASE Violated(Co1): ScaleIn
St2: STRATEGY CASE Violated(Co2): ScaleOut
#ComponentLevel.AppID.componentID
Co3: CONSTRAINT mem.usage < 80 AND cpu.usage < 20
Co4: CONSTRAINT battery.Level < 10
St3: STRATEGY CASE Violated(Co3): ScaleIn
St4: STRATEGY CASE Violated(Co4): ScaleOut
Pr1: Priority (Co4)>Priority(Co3)
```

Listing 1. An example of elastic requirements.

work remains to provide a mechanism that will inject elastic requirements easily in the YAML file (i.e., since our application runtime platform considered is Docker).

5 DECENT - DESIGN AND PROCESSES

This section provides an overview of our approach to deploy service-based edge applications and control their elasticity on an edge network. We extensively outline the main components of DECENT and the interaction of its components during runtime.

5.1 System Overview

An overview of our approach at design time and runtime of the system is illustrated in Figure 3. The developer defines the edge application model and its requirements at design time, as described in the previous section. In essence, the developer defines the application structure and resource requirements for each component. The elasticity requirements and deployment policy can be determined by the developer as well as by the system user (i.e., owner) before deployment. Along these lines, the deployment process starts when the user requests the system's configurator device to deploy an edge application. Exploring the module that enables users to interact with the system is out of this article's scope.

Each edge device consists of similar system components, as illustrated in Figure 3. However, the edge device that becomes the system's configurator provides the main features to control edge applications and runtime aspects. The architecture of the approach comprises five main modules, as described in the following.

`Deployment Planner(DP)`: The goal of this module is to generate QoS-aware deployment plans for deploying edge applications on premises of Edge-Cloud infrastructure. The `DP` module provides two main sub-modules: (1) *Planner* and (2) *Resource Manager*. The Planner generates deployment plans for a given edge application by considering both its app requirements and deployment policy. The Planner is essentially responsible for finding all possible eligible deployment plans by considering application hardware requirements (i.e., CPU, RAM, and storage), bandwidth, and latency between components. Moreover, the deployment and scaling policies tell the Planner which infrastructures are allowed to be considered when deploying application components. The Planner gets the current state of the infrastructure(s) through the Resource Manager. The Resource Manager is responsible for monitoring and storing the infrastructure-specific metrics such as resource

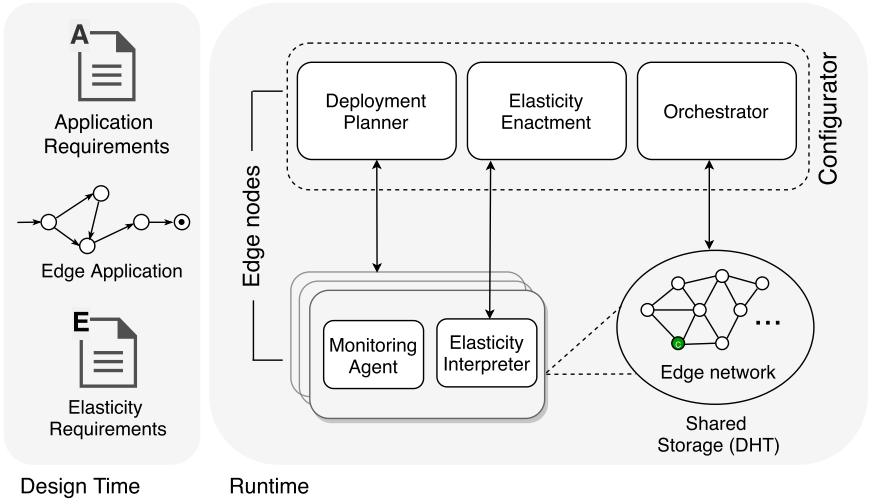

Fig. 3. Overview of DECENT's components and their interaction during runtime.

capabilities of edge devices (i.e., hardware), their current resource utilization, and link latencies between devices. Such information is provided by *Monitoring Agents* deployed at each edge device.

`Monitoring Agent(MA):` This module measures a set of infrastructure-specific metrics and application performance metrics continuously. In essence, the infrastructure-specific metrics are CPU, RAM, storage, battery levels (when applicable), and their current utilization. Nevertheless, the `MA` module continuously measures application metrics such as hardware-related resource consumption, response time, application status, and so forth. However, note that our approach monitors only metrics specified in the elasticity requirements. The `MA` module periodically sends information to the Resource Manager. Such data is temporally stored locally on the configurator device, and it is regularly updated when an application is deployed. In this article, we do not investigate how the monitoring agents are implemented in cloud and fog environments. Several studies [5, 11] address relevant aspects for monitoring agents, and several monitoring tools exist for providing the necessary information.[7,8]

`Elasticity Enactment Engine (3E):` This is an event-driven module with the goal to handle the actual coordination between the application's desired state and the current application elasticity state. In the DECENT runtime, elasticity requirements are interpreted by the *Elasticity Interpreter* deployed on each edge device. When a component's elasticity constraint is violated, the Elasticity Interpreter communicates such information to the `3E` module. Afterward, the `3E` module enforces required actions such that the component requirements are fulfilled. Thus, the `3E` module is the central part of the runtime system, which manages edge applications. In contrast, Elasticity Interpreters are local runtime engines that capture application component elastic requirements, interpret, and communicate necessary actions to the enactment engine. The following section briefly outlines the overall process of managing edge applications at runtime.

`Orchestrator:` This module provides several functionalities to support executing edge applications in dynamic edge networks. The Orchestrator functionalities are mostly addressed in our previous works (as discussed in Section 2) and are not evaluated in this article. In addition to

[7]Nagios, https://www.nagios.com/.
[8]Ganglia, http://ganglia.sourceforge.net.

that, the Orchestrator is responsible for creating, controlling, and managing the cluster of Docker Engines called *swarm*. Essentially, the system's configurator device simultaneously is the *swarm manager*, and the other edge devices are *worker devices*. The swarm manager maintains the swarm state through the Raft Consensus Algorithm.[9] On the contrary, the Orchestrator is responsible for keeping the quorum of managers in the system consistent.

At system design time, the Orchestrator is configured regarding the number of swarm manager devices that should be consistent at runtime and the expected size of the edge network. The system designer should consider a tradeoff between performance and fault tolerance when it defines the number of swarm managers. Having more swarm manager devices makes the system more fault tolerant, while writing performance is reduced (i.e., due to the network round-trip traffic). We configure an odd number of swarm manager devices to take advantage of the swarm mode's fault-tolerance features. The Orchestrator promotes new swarm manager devices whenever the edge network doubles the expected network size. Notice that we may have a maximum of five managers in an edge network.

Nevertheless, the Orchestrator periodically monitors the desired swarm manager number (i.e., system designer perspective) and the current number of swarm managers. Thus, if the desired state is violated, it takes the required actions to keep swarm managers' quorum in the system. Notice that the Orchestrator configuration data (i.e., swarm managers, swarm cluster joining key, etc.) is stored as DHT, meaning that it is shared and kept consistent between all devices within the whole network.

5.2 The Process

Edge applications are multi-container Docker-based applications. This means that the developer defines components that make up an edge application, including their hardware requirements specified in the *docker-compose file*. Essentially, an edge application and each component have their own unique name when they are deployed. Furthermore, at design time, the user specifies the deployment and scaling model and the elastic requirements (as presented in Listing 1). Both these requirements are formatted and stored as a single JSON file. Thus, we assume that the mentioned requirements are given at the design time.

The process starts when the user requests (1) the configurator device to deploy an edge application at an edge network (as illustrated in Figure 4). At this phase, the deployment planner interacts with the Orchestrator to get the current hardware infrastructure status, available edge devices, resource information, resource utilization rates, and latency of the communication links between devices ((2)–(3)). Afterward, it gets the edge application docker-compose file and specified hardware requirements and generates all eligible deployment plans (3). To generate such plans, it runs the algorithm presented in [4]. In essence, for each application component, the DP module finds all possible devices that fulfill the component's hardware requirements. Furthermore, the DP module notifies the user whether the edge application can be deployed at the specified deployment and scaling model. For instance, if the deployment and scaling model is set to the only *edge*, it means that all application components must be deployed at the edge (if possible).

Suppose the DP module generates at least one or more deployment plans. For each component, we have a list of compatible devices that can run them. Afterward, the DP module gets the list and updates the docker-compose file by adding the placement constraints. This means that for each component, it specifies devices where the component can be deployed. After this process is finished, the configurator executes Docker-based commands to deploy the edge application. Deploying and starting components (i.e., containers) can take several seconds and depend on edge

[9]Raft Consensus Algorithm, https://raft.github.io/.

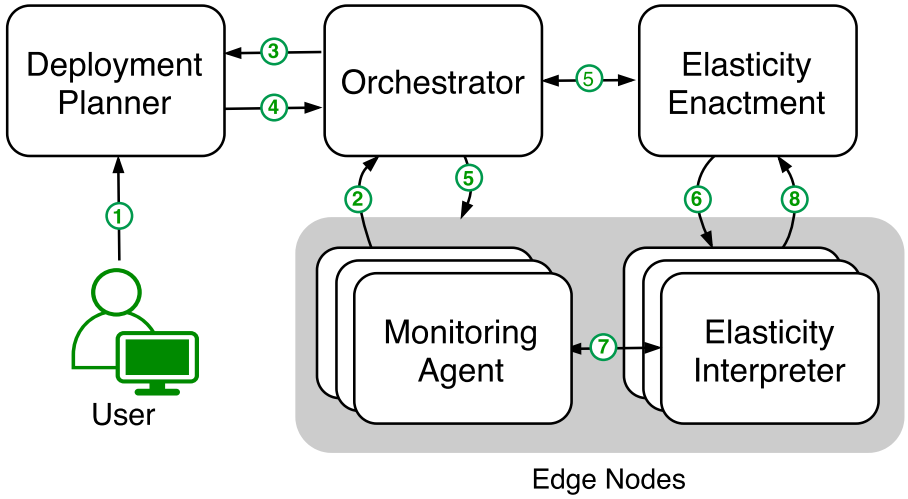

Fig. 4. The process at runtime.

device hardware capabilities. However, in this article, we do not investigate performance aspects when deploying and starting containers. A study that acknowledges the problem and addresses relevant aspects is presented in [1].

Through the Orchestrator module, the configurator shares elastic requirements with edge devices (5). Elastic requirements are shared by using DHT. Essentially, each edge device automatically identifies when the configurator assigns a container (e.g., named φ_1) to them. Thus, when φ_1 is in the running state, elasticity interpreters on each device query DHT to receive elastic requirements for the running applications. The user can change elastic requirements at runtime. The changes made on elastic requirements (i.e., in the configurator device) are automatically updated by other edge devices and captured by corresponding Elasticity Interpreters. Afterward, before starting elasticity monitoring (7), the elasticity interpreter first checks whether it is the only device running φ_1. The configurator device provides information (6), and such information is required to avoid situations where multiple elasticity interpreters start monitoring φ_1 (i.e., when φ_1 runs on several devices). Moving on, consider a situation when an elasticity interpreter monitors a constraint that says the edge application component requires to scale up when it uses 80% of the edge device CPU (e.g., see Listing 1). Thus, when the specified constraint is violated, the interpreter communicates it to the 3E module (8), which is responsible for enforcing the scaling operation. Note that monitoring agents provide hardware-related metrics and container-based metrics.

The 3E module is triggered when it receives information to enforce a specific strategy in the particular application component. In essence, the enforcement operation for the violated constraint is specified in elastic requirements. Before executing the action, the 3E module checks whether the application component should scale up or down. If the application component requires scale-up, it requests the DP module to check whether the component can scale in the current infrastructure state. We apply this strategy to avoid enforcing scaling operations in infrastructure with insufficient resources. Otherwise, the Docker runtime environment will continuously try to scale the application component without the mentioned strategy, causing network congestion and computation overload. Moreover, the configurator device for each edge application periodically monitors elastic requirements at the edge application level. The overall resource usage (i.e., for all running components) is considered and whether constraints are violated at the application-level requirements is analyzed.

In case the configurator device fails, edge devices hold an election to find a new configurator device as presented in [21]. Elasticity interpreters contact other swarm managers to enforce scaling operations until a new configurator device is elected. Since all edge devices keep consistent data through DHTs, the newly elected configurator is initialized quickly by considering the locally stored data. Nevertheless, each device in the network knows the system's current configurator device at any time. Note that the aspects mentioned above are primarily addressed in our previous works (as discussed in Section 2) and are not evaluated in this article. Furthermore, edge applications running at the edge network are not affected by possible configurator failure.

6 EVALUATION

This section first presents details about the prototype implementation, setup environment, and limitations. Furthermore, we experimentally evaluate the approach's effectiveness and present the evolution of an edge application in elasticity space. We conclude with a discussion in Section 6.3.

6.1 Prototype Implementation, Setup, and Limitations

To assess the proposed approach, we extend the prototype of [21] with a lightweight mechanism that enables deploying and controlling the elasticity of edge applications in a decentralized manner at an edge network. The prototype is partially developed and written in Java. The prototype is tested in a real environment on edge devices (i.e., Raspberry Pi 3 Model B V1.2) with 4 × ARM Cortex-A53 CPU at 1.2 GHz, 1 GB of RAM, and 16 GB disk storage. The prototype is deployed on each edge device, and each edge device runs the Docker Engine as the edge application runtime platform. To implement the deployment generator, we refined and extended parts of the FogTorchΠ [4] simulator to generate all eligible deployment plans for an edge application. In addition, the simulator does not consider dynamic environments or runtime aspects, does not provide elasticity features, does not implement monitoring tools, and does not implement any communication protocol between computation entities at the edge. Thus, the extensions we refer to are further functionalities developed to support the runtime aspects of edge applications (e.g., elasticity, etc.) in a realistic testbed. Along these lines, the planner is fully integrated into the prototype. It gets the infrastructure state (i.e., available devices, network structure) and generates plans by considering real-time infrastructure-specific metrics.

The 3E module is implemented as a thread that runs continuously and listens for requests generated by elasticity interpreters. This module enforces various operations by Docker Java API[10] that allows building, controlling, updating, and running containers. Docker Engine REST APIs allow us to configure and update containers (i.e., running services) whenever it is needed. Additionally, some features that the mentioned APIs do not support were implemented by executing Docker commands using the command-line interface. Elasticity Interpreters are deployed on each edge device, and for each running container with its elastic requirements, a single thread is executed to interpret those requirements. To implement the monitoring agent, we used *Hyperic Sigar*[11] to collect hardware information on edge devices. The network latency between devices is collected by using the *ping* command. Furthermore, Docker Java API is used to collect hardware utilization data about running containers (i.e., state, CPU, memory, storage, etc.). For each running container with elastic requirements, a single thread is executed to monitor specified elastic metrics. Thus, the thread number is dependent on the number of running containers on an edge device.

To evaluate our prototype, we exploited the testbed (i.e., edge network) composed of 10 edge devices placed close to each other. Edge devices in the testbed are connected through a wireless

[10]Docker Java API, https://github.com/docker-java/docker-java.
[11]Hyperic Sigar, https://github.com/hyperic/sigar.

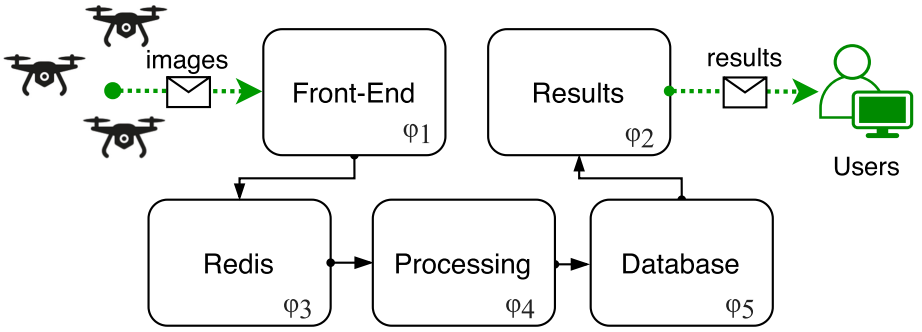

Fig. 5. Edge safety application.

connection with a nominal speed of 10 Mbps and 5 Mbps in download and upload. Furthermore, the prototype's main limitation is being executed in the **Java Virtual Machine (JVM)** environment. We acknowledge that the JVM is resource expensive; however, we aim to show the approach's feasibility within this article.

6.2 Use Case, Experiments, and Results

Consider an edge application (i.e., edge safety application) providing a service as described in our motivation scenario (Section 2.2). The edge safety application is partially developed, and it is made out of five components, with three of them written in Python (as illustrated in Figure 5). The front-end component φ_1 enables edge devices (i.e., drones) to interact with service and continuously upload their real-time images, including location coordinates. The Redis component φ_2 collects new images and stores them in binary format. The processing component φ_3 consumes data and processes (i.e., image analysis) and stores them in the database component φ_4 (i.e., Postgres). Finally, the results component visualizes the safe path for the rescue team member residing in the affected zone.

Software components are containerized (docker images). Each container is configured with specific resource requirements (i.e., 1 CPU (1.2 GHz) and 60 MB memory) and resources that containers can use on the hosting edge device. Notice that to reserve and use a various number of CPU resources per container, the RPi3 must be upgraded to the latest firmware.[12] Furthermore, we assume that the component images are already available on each edge device. Such an assumption is made due to the latency issues introduced when images are downloaded from centralized devices. In [1], the authors acknowledge the problem and address relevant aspects to improve deployment time.

Figure 6 illustrates the average time required to generate all possible valid deployment plans for each edge safety application component when needed to be deployed and scaled. We simulate the generation of deployment plans 10 times and illustrate their maximum average time requirements. For the edge safety application with five components and the given testbed, we may have up to 84,960 generated valid deployment plans (i.e., maximum average time is 6.73s). Nevertheless, once a single valid plan is founded, the process is interrupted, and the application components are deployed or scaled. Notice that when the infrastructure changes, the DP module must generate all valid deployment plans. Table 1 presents the maximum deployment plans generated for each software component (i.e., container). Notice that the maximum time requirement and the number of generated valid plans change based on the available resources at edge networks as well as edge application requirements specified at design time.

[12]Raspberry Pi, https://github.com/raspberrypi/firmware.

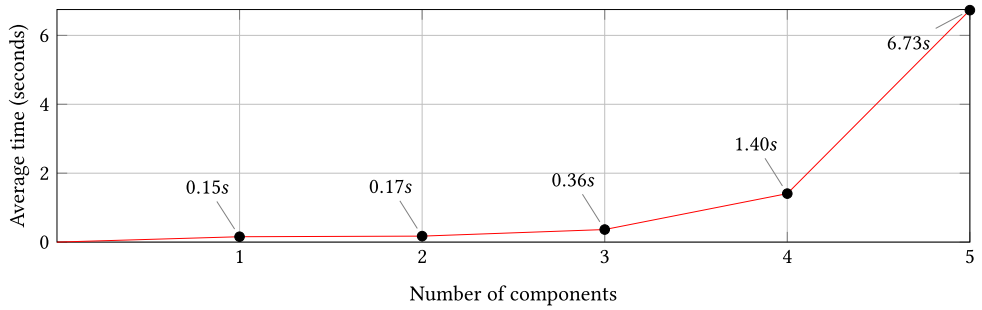

Fig. 6. Edge safety application deployment at an edge network with 10 low-powered edge devices.

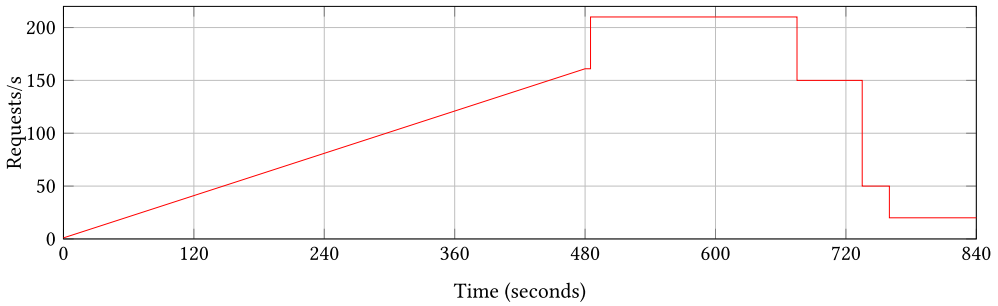

Fig. 7. Workload used in the experiment.

Table 1. Number of Eligible Deployments Plans (i.e., in a Testbed with 10 Edge Devices)

Edge Safety Application	Generated Plans	Time (seconds)
One Component	10	0.15 s
Two Components	100	0.17 s
Three Components	1,000	0.36 s
Four Components	9,720	1.40 s
Five Components	84,960	6.73 s

The edge safety application is configured to run and scale only at the available devices at the target edge network. To simplify the scenario, we evaluate the front-end component and show the adaption process in response to its changing workload during its runtime. The front-end is the first component in which drones (i.e., edge devices) interact with the edge application by uploading their images continuously (i.e., every 1 s).

Figure 7 illustrates the workload generated for the safety edge application and used in the experiment. First, the workload increases linearly every 3 seconds. Afterward, we stress test our approach by creating concurrent requests (i.e., up to 30 requests/s) and examine the front-end component behavior during its runtime. Such a workload is generated by new edge devices that use the service. For instance, referring to Figure 7, when the component receives up to 50 requests per second, the component may be required to scale out to operate correctly since it may utilize the CPU more than 80%. Or, when the component receives fewer than 50 requests per second, it means that it may need to scale in to not overuse resources. Generally speaking, edge applications

or their software components may experience various workloads over time (i.e., periodically, continuously, or unpredictably). The given workload is just an example used for testing purposes to show our approach's goal to automatically control edge application behavior in elasticity space. To enable elastic behavior for the front-end component, we define elastic requirements in Listing 2.

```
#ComponentLevel.EdgeApp01.Front_End
Co1: CONSTRAINT cpuUsage > 75
Co2: CONSTRAINT cpuUsage < 30 AND memUsage < 30
Co3: CONSTRAINT averageRes > 100
Mo1: MONITORING cost = cost.instant
Mo2: MONITORING cpuUsage = cpu.usage
Mo3: MONITORING memUsage = mem.Usage
Mo4: MONITORING averageRes = IO.response
St1: STRATEGY CASE Violated(Co1): ScaleOut
St2: STRATEGY CASE Violated(Co2): ScaleIn
St3: STRATEGY CASE Violated(Co3): ScaleOut
PR1: Priority(Co3) > Priority(Co2)
PR2: Priority(Co3) > Priority(Co1)
```

Listing 2. An example of elastic requirements: Front-end component.

Elastic requirements given in Listing 2 define the elastic behavior of the front-end component. Strategy St3 states that if the average response latency is higher than 100 ms, the component should scale out to ensure the service's quality. When Co1 or Co2 is violated, strategies St1 or St2 enforce specified actions to keep resource utilization in acceptable ranges. As can be noted, the specified metrics are monitored continuously for the front-end component. Furthermore, each constraint may have various priorities. For instance, no matter how much the CPU is utilized, the front-end component must scale if the provided service has higher latency than a specified threshold. To that end, if both constraints are violated, the Co3 is enforced since it is prioritized over Co2. Similarly, Co3 is enforced first since it is prioritized over Co1.

The first graph of Figure 8 shows the CPU utilization by the front-end component under the given workload (see Figure 7). The second graph of Figure 8 shows the front-end component adaption process in response to the workload. As can be noted, whenever elastic constraints (i.e., Co1 and Co2) are violated, the front-end component scales up or scales down. The adaption process occurs automatically in response to the current workload. The front-end component scales on multiple instances (i.e., containers) to provide the desired service quality. As can be noted, even with continually changing the workload of the front-end component, the CPU utilization remains between elastic boundaries. This ensures that the desired service quality is always guaranteed. The other important aspect is to overcome resource over-provisioning. As can be noted, when the front-end component's workload decreases, the container number is decreased as well (see the second graph of Figure 8). Besides, the front-end component's memory utilization remained within elastic boundaries and didn't violate elastic constraints. The CPU of an edge device may fluctuate up and down very quickly due to various workloads. This may cause undesired scaling operations for the same workload. Thus, specified metrics are monitored for 5 seconds to overcome the mentioned problem. The scaling operation is enforced if the mean value violates elastic constraints. Furthermore, Figure 9 shows the front-end component latency over time and the elastic constraint Co3 violation. However, in this situation, both Co1 and Co3 constraints are violated. As can be noted from the elastic requirement, the Co3 constraint is prioritized over the Co1. In this case, strategy St3 will be enforced to keep the latency within the elastic boundary.

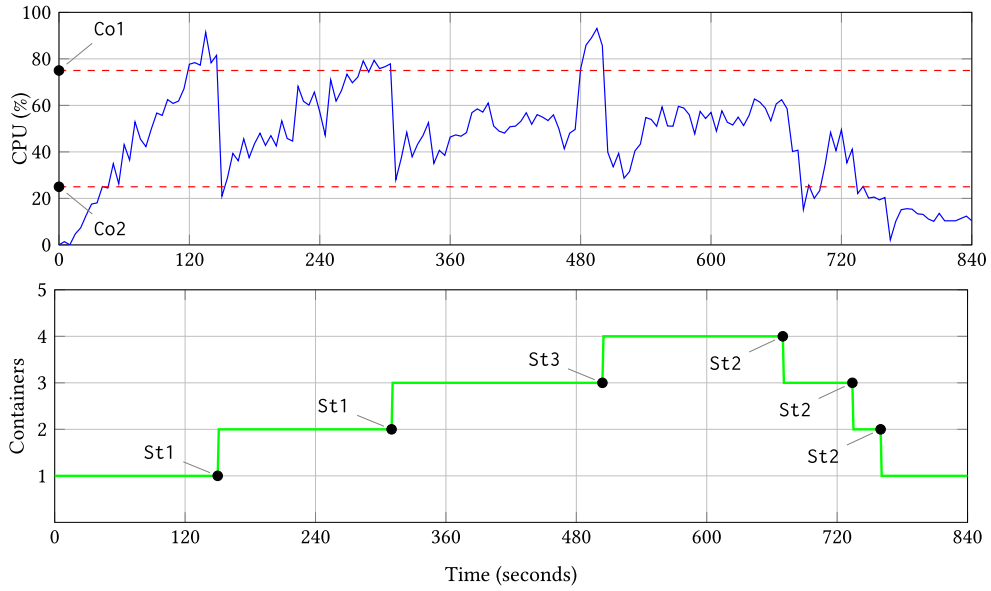

Fig. 8. The CPU utilization and adaptation process.

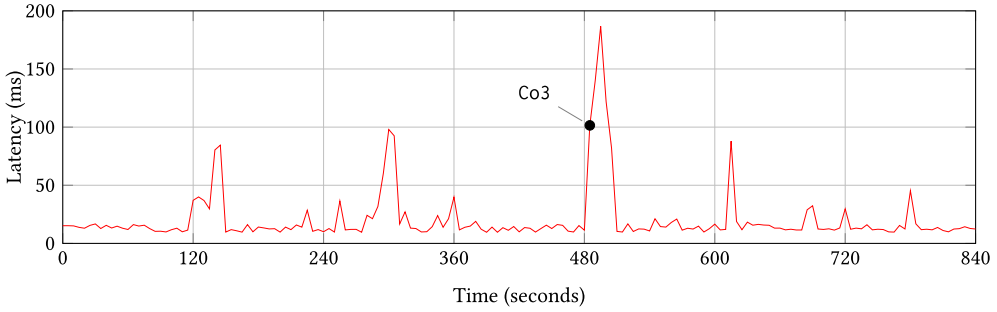

Fig. 9. Front-end component latency and adaptation process.

A significant challenge remains in the time required to start containerized components' low-powered edge devices. In our case, starting safety edge application components takes between 20 and 30 seconds. After the scaling operation is enforced, our approach checks and waits for whether a container is started or shut down. Thus, we avoid undesired situations such as enforcing multiple scaling operations for the same workload. Contrary to the starting operation, the shutdown process occurs in a few seconds for all containers. Nevertheless, edge application components can scale vertically and horizontally depending on the available resources at the edge. The application runtime platform (i.e., Docker Engine) manages this process and scales components within the list of eligible edge devices generated by the DP planner.

In Figure 10, we show the evolution of the front-end component in the three-dimensional space (cost, quality, and resources). The quality refers to the latency, the resources refer to the allocated CPU (i.e., edge devices), and the cost is estimated based on the resource allocated. In this case, the cost value is an assumption made to simulate the price paid for resource usage. As can be noted from Figure 10, when the service quality decreases (i.e., starting point with green dots), the

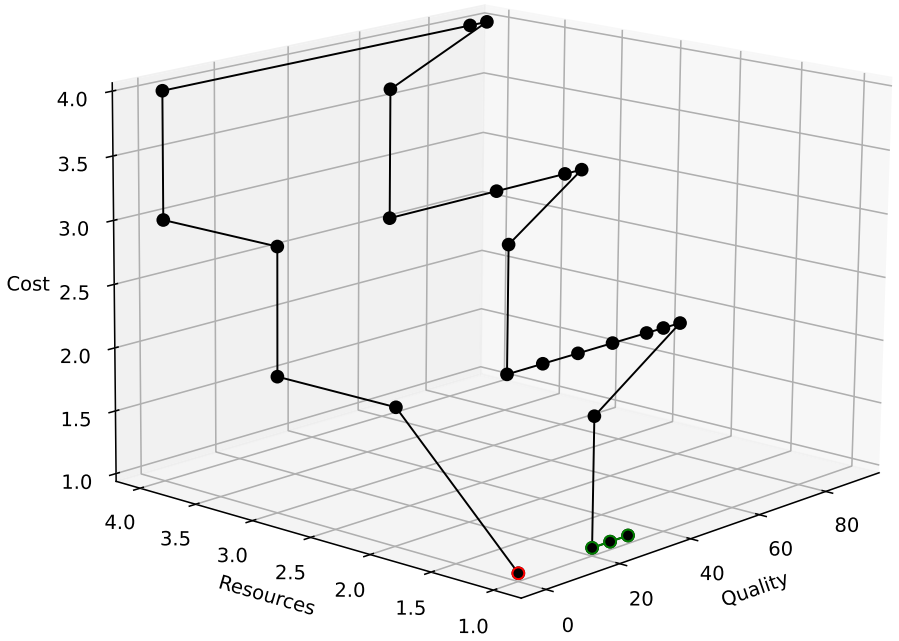

Fig. 10. Evolution of front-end component in the elasticity space.

front-end component scales by increasing the number of resources used and the cost is increased. Furthermore, the edge application scales down when the service is not used (i.e., red dot). To that end, such an approach is guaranteed to meet edge application resource demands at runtime. Other edge application components evolve in the elasticity space based on their load during runtime. Similarly, the configurator device monitors elastic requirements specified at the edge application level. Thus, the configurator device considers the overall resource consumption of edge application components. For instance, the user may specify that a particular edge application cannot use more than 50% of available resources at the edge.

6.3 Discussion

We have demonstrated through a running example that automatic scaling of edge applications is easily achieved in an edge infrastructure with low-powered devices by using DECENT. Furthermore, we showed that our approach helps avoid highly undesirable situations, such as resource over-provisioning. This ensures that the available resources are used whenever edge applications need them. Nevertheless, elasticity features are crucial in avoiding edge device failures due to resource over-utilization. For instance, a low-powered edge device can quickly fail when an edge application or a software component fully utilizes device resources. Thus, specifying elastic requirements and the DECENT mechanism helps to avoid the overloading of edge devices.

Several assumptions inherent in our approach must be further investigated. In the current prototype, elastic constraints do not conflict with each other. We focus on developing novel constraints and enforcement strategies related to these applications. Simplifying the development process of elastic specifications is among the future works we plan to do. Thus, we plan to integrate the language into an IDE such as c-Eclipse and extend it with further functionalities. The c-Eclipse framework provides a user-centric interface through which developers can describe their applications for deployment over edge and cloud. The language integrated into a development

environment will make it easy for users to develop elastic specifications and specify correct values to avoid the wrong configuration. Nevertheless, the following tool will also help to detect conflicting constraints that the user may select. Nevertheless, we acknowledge that the user may specify conflicting elastic constraints, and thus, we plan to investigate various techniques that would help identify and avoid such situations.

Within this article, our primary focus resided in enabling elasticity features at edge infrastructures; thus, we consider only edge applications where all components are deployed on the edge (i.e., everything on the edge model). Accordingly, adding cloud or fog devices will expand overall available resources and allow executing application components (i.e., containers) in these environments when insufficient resources are at the edge. In future work, we will investigate performance aspects when moving edge application components in large-scale Edge-Cloud infrastructures and controlling their elasticity from the edge.

7 CONCLUSION

Satisfying dynamic and stringent requirements of edge applications has become challenging for resource-constrained edge networks. Even though edge applications can be modeled as multi-components, dynamic workloads may cause unexpected latencies that are higher than the expected response time between application components, IoT devices, and end-users. We proposed an efficient solution that simplifies the deployment process and enables elasticity controlling in edge applications deployed in the Edge-Cloud infrastructure to overcome such challenges. The developer and user can characterize edge applications by specifying elasticity requirements that are captured and interpreted by DECENT. The DECENT runtime mechanism then performs complex elasticity controls at the edge of a network.

Edge networks can be different in size and setting; thus, the proposed system is configurable by the system designer. In this article, we consider edge networks as resource-constrained environments composed of low-powered edge devices. The experiments conducted in a realistic testbed showed the feasibility of executing elastic features on low-powered edge devices and adapting edge application components at runtime at the edge. Furthermore, edge applications are executed in a runtime that considers the heterogeneity of edge resources. The proposed framework automatically reconfigures edge applications to meet their specified elastic requirements. In future work, we plan to perform an extensive evaluation of the approach by considering distributed cloud entities in the system. Furthermore, we plan to develop a user-centric interface through which developers and users can easily describe their edge applications for deployment over edge and cloud environments.

REFERENCES

[1] Arif Ahmed and Guillaume Pierre. 2018. Docker container deployment in fog computing infrastructures. In *2018 IEEE International Conference on Edge Computing (EDGE'18)*. IEEE, 1–8.

[2] Jean-Paul Arcangeli, Raja Boujbel, and Sébastien Leriche. 2015. Automatic deployment of distributed software systems: Definitions and state of the art. *Journal of Systems and Software* 103 (2015), 198–218.

[3] Majid Ashouri, Paul Davidsson, and Romina Spalazzese. 2018. Cloud, edge, or both? Towards decision support for designing IoT applications. In *2018 5th International Conference on Internet of Things: Systems, Management and Security*. IEEE, 155–162.

[4] Antonio Brogi and Stefano Forti. 2017. QoS-aware deployment of IoT applications through the fog. *IEEE Internet of Things Journal* 4, 5 (2017), 1185–1192.

[5] Antonio Brogi, Stefano Forti, and Marco Gaglianese. 2019. Measuring the fog, gently. In *International Conference on Service-Oriented Computing*. Springer, 523–538.

[6] Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, and Schahram Dustdar. 2013. SYBL: An extensible language for controlling elasticity in cloud applications. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 112–119.

[7] Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong-Linh Truong. 2011. Principles of elastic processes. *IEEE Internet Computing* 15, 5 (2011), 66–71.
[8] Schahram Dustdar and Ilir Murturi. 2020. Towards distributed edge-based systems. In *2020 2nd IEEE International Conference on Cognitive Machine Intelligence*. IEEE, 1–9.
[9] Schahram Dustdar and Ilir Murturi. 2021. Towards IoT processes on the edge. In *Next-Gen Digital Services. A Retrospective and Roadmap for Service Computing of the Future*. Springer, 167–178.
[10] Rafael Fayos-Jordan, Santiago Felici-Castell, Jaume Segura-Garcia, Adolfo Pastor-Aparicio, and Jesus Lopez-Ballester. 2019. Elastic computing in the fog on Internet of Things to improve the performance of low cost nodes. *Electronics* 8, 12 (2019), 1489.
[11] Stefano Forti, Marco Gaglianese, and Antonio Brogi. 2020. Lightweight self-organising distributed monitoring of fog infrastructures. *Future Generation Computer Systems* 114 (2020), 605–618. https://doi.org/10.1016/j.future.2020.08.011
[12] Jonathan Fürst, Mauricio Fadel Argerich, Bin Cheng, and Apostolos Papageorgiou. 2018. Elastic services for edge computing. In *2018 14th International Conference on Network and Service Management (CNSM'18)*. IEEE, 358–362.
[13] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. 2013. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC'13)*. 23–27.
[14] Cheol-Ho Hong and Blesson Varghese. 2019. Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms. *ACM Computing Surveys (CSUR)* 52, 5 (2019), 1–37.
[15] Gian Paolo Jesi, Alberto Montresor, and Ozalp Babaoglu. 2006. Proximity-aware superpeer overlay topologies. In *IEEE International Workshop on Self-managed Networks, Systems, and Services*. Springer, 43–57.
[16] Ivan Lujic and Hong-Linh Truong. 2019. Architecturing elastic edge storage services for data-driven decision making. In *European Conference on Software Architecture*. Springer, 97–105.
[17] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. 2018. Fog computing: A taxonomy, survey and future directions. In *Internet of Everything*. Springer, 103–130.
[18] Sunilkumar S. Manvi and Gopal Krishna Shyam. 2014. Resource management for infrastructure as a service (IAAS) in cloud computing: A survey. *Journal of Network and Computer Applications* 41 (2014), 424–440.
[19] Petar Maymounkov and David Mazieres. 2002. Kademlia: A peer-to-peer information system based on the XOR metric. In *International Workshop on Peer-to-Peer Systems*. Springer, 53–65.
[20] Ilir Murturi, Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. 2019. Edge-to-edge resource discovery using metadata replication. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC'19)*. IEEE, 1–6.
[21] Ilir Murturi and Schahram Dustdar. 2021. A decentralized approach for resource discovery using metadata replication in edge networks. *IEEE Transactions on Services Computing* PP, 99 (2021), 1–11. DOI: 10.1109/TSC.2021.3082305
[22] Stefan Nastic, Andrea Morichetta, Thomas Pusztai, Schahram Dustdar, Xiaoning Ding, Deepak Vij, and Ying Xiong. 2020. SLOC: Service level objectives for next generation cloud computing. *IEEE Internet Computing* 24, 3 (2020), 39–50.
[23] Nguyen Dinh Nguyen, Linh-An Phan, Dae-Heon Park, Sehan Kim, and Taehong Kim. 2020. ElasticFog: Elastic resource provisioning in container-based fog computing. *IEEE Access* 8 (2020), 183879–183890.
[24] Yuvraj Sahni, Jiannong Cao, Shigeng Zhang, and Lei Yang. 2017. Edge mesh: A new paradigm to enable distributed intelligence in internet of things. *IEEE Access* 5 (2017), 16441–16458.
[25] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.
[26] Olena Skarlat, Vasileios Karagiannis, Thomas Rausch, Kevin Bachmann, and Stefan Schulte. 2018. A framework for optimization, service placement, and runtime operation in the fog. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC'18)*. IEEE, 164–173.
[27] Fan-Hsun Tseng, Ming-Shiun Tsai, Chia-Wei Tseng, Yao-Tsung Yang, Chien-Chang Liu, and Li-Der Chou. 2018. A lightweight autoscaling mechanism for fog computing in industrial applications. *IEEE Transactions on Industrial Informatics* 14, 10 (2018), 4529–4537.
[28] Shanhe Yi, Zijiang Hao, Qingyang Zhang, Quan Zhang, Weisong Shi, and Qun Li. 2017. LAVEA: Latency-aware video analytics on edge computing platform. In *Proceedings of the 2nd ACM/IEEE Symposium on Edge Computing*. ACM, 15.
[29] Alessandro Zanni, Stefan Forsstrom, Ulf Jennehag, and Paolo Bellavista. 2018. Elastic provisioning of Internet of Things services using fog computing: An experience report. In *2018 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud'18)*. IEEE, 17–22.

Received March 2021; revised January 2022; accepted April 2022