Elasticity Control for Latency-Intolerant Mobile Edge Applications

Chanh Nguyen, Cristian Klein, and Erik Elmroth

Department of Computing Science, Umeå University, Sweden
Email: {chanh, cklein, elmroth}@cs.umu.se

2020 IEEE/ACM Symposium on Edge Computing (SEC) | 978-1-7281-5943-0/20/$31.00 ©2020 IEEE | DOI: 10.1109/SEC50012.2020.00013

***Abstract*—**

Elasticity is a fundamental property required for Mobile Edge Clouds (MECs) to become mature computing platforms hosting software applications. However, MECs must cope with several challenges that do not arise in the context of conventional cloud platforms. These include the potentially highly distributed geographical deployment, heterogeneity, and limited resource capacity of Edge Data Centers (EDCs), and end-user mobility.

In this paper, we present an elasticity controller to help MECs overcome these challenges by automatic proactive resource scaling. The controller utilizes information on the physical locations of EDCs and the correlation of workload changes in physically neighboring EDCs to predict request arrival rates at EDCs. These predictions are used as inputs for a queueing theory-driven performance model that estimates the number of resources that should be provisioned to EDCs in order to meet predefined Service Level Objectives (SLOs) while maximizing resource utilization. The controller also incorporates a group-level load balancer that is responsible for redirecting requests among EDCs during runtime so as to minimize the request rejection rate.

We evaluate our approach by performing simulations with an emulated MEC deployed over a metropolitan area and a simulated application workload using a real-world user mobility trace. The results show that our proposed pro-active controller exhibits better scaling behavior than a state-of-the-art re-active controller and increases the efficiency of resource provisioning, thereby helping MECs to sustain resource utilization and rejection rates that satisfy predefined SLOs while maintaining system stability.

***Index Terms*—Resource Provisioning, Elasticity, Auto Scaling, Edge Data Center, Workload Prediction, Location-aware, Machine Learning.**

I. Introduction

The explosion of Internet of Things (IoT) deployments and advances in 5G networking technology have enabled the emergence of many new applications and services in domains as varied as everyday leisure [1], mission critical health [2], and industrial process control [3]. These disruptive applications require shorter response times and greater network bandwidth than can be supported by conventional remote cloud datacenters, and their computing capacity requirements put pressure on the limited computing and storage capacity of end-user devices. To address these challenges, there has recently been a transition towards a new type of computing infrastructure known as a Mobile Edge Cloud (MEC), in which resource capabilities are distributed at the edge of the network, in close proximity to end-users. MECs provide relatively high computational capabilities and processing power at the network edge by exploiting the compute capacity of small servers or datacenters with heterogeneous capacity. These so-called Edge Data Centers (EDCs) are envisioned to be collocated with network base stations. This wide geographical distribution of resources allows MECs to provide services with higher bandwidth and lower latency than current cloud computing platforms can deliver.

The modern centralized cloud platform is a well-established paradigm defined by an on-demand service provisioning model in which *elasticity* is a key feature [4]. In essence, elasticity is the ability of a system to *automatically adapt resource provisioning as required to handle variation in load*. Within each time slot, an elastic controller seeks to provision sufficient resources such that the current demand is matched as closely as possible. With this in mind, we argue that elasticity will also be key to the success of MECs. If anything, MEC controllers must be even more rigorous in terms of speed and precision than those in centralized cloud infrastructures for four reasons: 1) Most applications deployed on MECs will be *latency-intolerant*, i.e., they are extremely sensitive to even very small delays. Sluggishness in resource scale-up or failure to allocate sufficient resources to meet demand (i.e., *under-provisioning*) can cause delays by increasing service waiting time or, worse, by increasing the service rejection rate, which results in a bad user experience; 2) The limited availability and high cost of resources at the network edge mean that allocating resources exceeding the demand (i.e., *over-provisioning*) leads to inefficient operation and costly resource wastage; 3) The stochastic nature of user mobility means that resource demand at the network edge is characterized by frequent transient changes [5]; 4) Resource control actions in cloud datacenters do not take immediate effect, but are bound by *actuation delays* (which can be as long as several minutes) until acquired resources are ready for use [6]. Acceptable levels of service cannot be guaranteed if these actuation delays are neglected, especially when resource usage varies rapidly as is likely in MECs.

These factors make it impractical for a human operator to make scale-up and scale-down decisions, so MECs require autonomous elastic controllers. Scaling decisions must be made quickly enough to capture sudden changes in resource demand so as to avoid Service Level Objective (SLO) violations, but they must also be very accurate to reduce the overhead of

978-1-7281-5943-0/20/$31.00 ©2020 IEEE
DOI 10.1109/SEC50012.2020.00013

the scaling actions themselves and avoid *system oscillation*. To address these challenges, this paper presents an elastic controller for MECs that aims to flexibly provision resources where and when needed, matching dynamic workload changes. The proposed approach helps the system meet SLO targets in terms of request processing time and service rejection rates while minimizing over-provisioning of MEC resources and maximizing system stability.

The authors have previously evaluated the impact of user mobility behavior on resource demand in MECs [6], showing that user mobility gives rise to cross-correlation in workload variation between nearby EDCs. This cross-correlation is a valuable input for improving the accuracy of workload predictions, and was therefore exploited to develop a location-aware workload prediction approach that outperforms previous state-of-the-art methods. The workload predictions generated using this approach are in turn valuable inputs for accurate pro-active estimation of future resource demand, enabling increased resource provisioning efficiency. Here, we propose a new location-aware elastic controller for MECs that applies this predictive approach. In essence, the proposed controller treats all EDCs located in close physical proximity as a group. Each group is managed by a group-level controller, which is responsible for three functions: 1) Predicting workload arrival at EDCs in the group; 2) Proactively determining how many resources to allocate at each EDC; and 3) Configuring load-balancers to direct requests from under-provisioned EDCs to EDCs within the group that have available resources.

The **key contributions** of the paper are summarized as follow:

- We present a location-aware elastic controller for MECs that accounts for correlations in workload changes of EDCs located in close physical proximity to one-another. The controller is responsible for pro-actively scaling up/down resources at EDCs and for redistributing workloads from under-provisioned EDCs to EDCs within the same group that have available resources (see Section III).
- We evaluate the performance of the proposed approach using various core elasticity metrics. First, an MEC topology is emulated with EDCs geographically distributed across the San Francisco metropolitan area. An extremely latency-sensitive application is assumed to be deployed on the MEC to serve end-users in this area; the application's workload is generated using real mobility traces of taxis in San Francisco (see Section IV).

The proposed elastic controller outperforms the state-of-the-art re-active controller and improves the efficiency of resource provisioning. More specifically, in our experiments presented in Section V, the elastic controller achieves an average resource utilization of 85% and a rejection rate of 0.02%, while the corresponding values for the state-of-the-art re-active controller are 69% and 0.04%, respectively.

II. Problem Definition

Figure 1 depicts an MEC platform with resources distributed in EDCs and workload variation at each EDC. In this section, we first describe the goals of an elastic controller for managing MEC resources from the perspectives of both the operator and the end users. We then describe an MEC platform whose resources are distributed across a metropolitan area, and a scenario involving deployment of an extremely latency-sensitive application. Finally, we highlight the challenges of provisioning resources to such an application in order to guarantee acceptable levels of service while attempting to increase resource utilization.

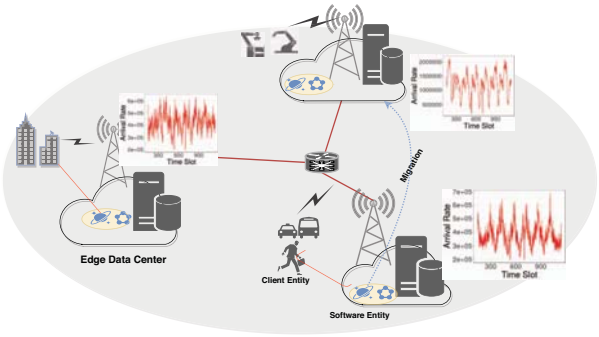


Fig. 1: An MEC platform showing the temporal variations in workload at each EDC.

A. Elastic Controller Goals

The overall purpose of the elastic controller is to guarantee that the allocated resources match the current demand as closely as possible. From the end-user's perspective, the controller tries to prevent the system from violating SLOs that are defined (by end-users) in terms of the average request response time and request rejection rate.

From the operator's perspective, the controller should focus on the core elasticity metrics, i.e., average resource utilization and system stability. These metrics may be in conflict under certain conditions. For example, increasing resource utilization may increase the rejection ratio. Our proposed method prioritizes reducing the rejection ratio over reducing system oscillations and maximizing resource utilization.

The metrics considered in this work are:

- **Average Rejection Rate.** To quantify end-user experience, we measure the average rejection rate of EDC_i as

$$Reject_{EDC_i} = \frac{\sum Request_{rejected}}{\sum Request} \quad (1)$$

  A request is considered rejected if the corresponding SLO target is violated (e.g., the estimated response time is greater than the predefined maximum response time).
- **Average Resource Utilization.** We define the average resource utilization as

$$U_{EDC_i} = \sum_{p_k in EDC_i} \frac{T^{p_k}_{busy}}{T^{p_k}_{living}} \quad (2)$$

where $T^{p_k}_{busy}$ is total time that an execution unit p_k of EDC_i (e.g., container, VM) is busy to process requests and $T^{p_k}_{living}$ is total time since p_k was invoked.

- **Average Resource Lifetime.** To quantify system stability, we measure the average resource lifetime, i.e., the difference between the starting and stopping times of an execution unit p_k for the application. Longer lifetimes imply greater stability.

B. MEC Metropolitan Platform

We consider an MEC platform consisting of n EDCs distributed over a metropolitan area. The infrastructure may have tens to thousands of EDCs of various sizes separated by short distances (e.g., 1 km to 10 km). These EDCs are collocated with network base stations or access points dispersed across the area and are connected to one-another via a local network. The network delay is $5\mu s/km$, which is a typical value for fiber optic systems [7].

Unlike "unlimited capacity" cloud datacenters, each EDC is a micro-server or datacenter with a limited resource capacity reserved for providing cloud-based services to nearby end-users. We assume each MEC's resources are allocated using the approach typically applied in modern container orchestration platforms such as Kubernetes[1]: each instance of an application is deployed using an execution unit called a Pod. Each Pod consists of a container that shares physical resources with other containers collocated in the same EDC.

C. Latency-Intolerance Application

The explosion of new IoT applications is making the virtues of proximity increasingly apparent. Jitter is a significant problem for many applications of this sort, and is highly impacted by the variation in latency inherent to any multi-hop network. Clearly, reliance on a centralized cloud datacenter, where resources are located far from the end-users, is inadvisable for applications requiring end-to-end delays to be tightly controlled to less than a few tens of milliseconds.

For example, the quality and effectiveness of Augmented Reality (AR) systems depends strongly on accurate registration, which is negatively impacted by latency. High latency leads to mis-registration and thus breaks the spatial relationships between real and virtual objects, disrupting the illusion that the virtual objects are part of the physical world [8]. To achieve a good user experience for such applications, the *motion-to-display latency* – the total delay from the time a motion occurs to the time the display is updated to reflect its results – must be *below 2.5 ms* [9]. In Section IV we conduct experiments focused on an AR application. However, other envisioned extremely latency-sensitive applications such as car collision warning and emergency stop systems, which require similar latency requirements [10] are applications that benefit from MEC deployment, which also motivate our work.

[1]https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/

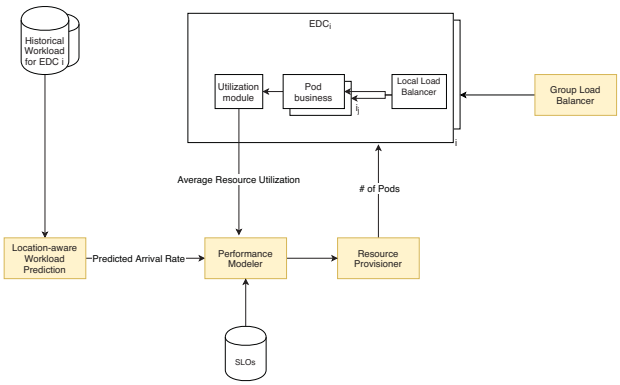


Fig. 2: Components of the proposed group controller.

D. Workload Variations

To provide services with an optimal network delay and a level of jitter compatible with the relevant SLOs, MECs should ideally connect end users to the nearest EDCs, i.e., one-hop-away datacenters. Because end-users may move across a geographic area, individual EDCs will experience temporal workload variation. These variations in workload pose a challenge for management operators because they complicate the process of deciding when and where resources should be allocated, and in what amount.

III. Proactive Elastic Control Framework for Edge Data Centers

This section explains the operating principles of the proposed elasticity controller. As presented in section II, each EDC in an MEC has a limited number of Pods reserved to serve end-user requests. In each EDC, we configure a local load balancer that uses the *shortest queue first technique* to balance requests arriving at the EDC and allocate them to the active Pods that are able to serve requests in the shortest time. EDCs located within close physical proximity to one-another are assigned to the same group, and resource provisioning within each group is controlled by a single group-level controller.

Figure 2 shows the main components of the proposed controller, which are *a location-aware workload predictor, a performance modeler, a resource provisioner*, and *a group load balancer*. In each time interval, the group controller performs 4 sub-tasks: *i)* The controller queries the historical data repository to obtain historical data for a time-window of predefined length (e.g., the workload history of the last 5 days), which is saved as a time series. *ii)* The time series data is forwarded directly to the *location-aware workload predictor*, which then predicts the workload arrival rate for the next interval. *iii)* The predicted workload arrival rate, the current resource utilization, and the predefined SLO thresholds are then sent to the *performance modeler*, which estimates the number of desired Pods to be provisioned in the next interval for each EDC. *iv)* Finally, the current number of Pods on each

EDC and the estimated number of Pods required in the next interval for each EDC are sent to the *resource provisioner*, which determines the final desired number of Pods for each EDC. At running time, the *group load-balancer* is responsible for redirecting requests from under-provisioned EDCs to EDCs with available resources so as to minimize the rejection rate.

Having thus established the flow of data within the controller, the following paragraphs describe the individual components of the proposed framework:

1) ***Location-aware Workload Predictor****:* The workload predictor predicts workload arrival rates at each EDC in the group over a configurable interval. It operates at the beginning of the interval to generate predicted workload arrival rates to the controller. Because of the rapid workload changes that occur in MECs and the delay time due to boosting and starting up provisioned resources before they actually become available to end-users, a fast response and highly accurate forecasting are essential. In [6], the authors developed a location-aware prediction approach using multivariate Long Short-term Memory networks. Experimental investigations showed that this approach is efficient in terms of speed and accuracy. The method leverages the correlation between changes in workload among EDCs located in close physical proximity to predict how the workloads of the EDCs in an MEC will evolve over short periods of time. Given an MEC topology with geographically distributed EDCs, let $Y_t = (y_t^1, y_t^2, ..., y_t^{n+1})'$ be an $((n+1) \times 1)$ dimensional time series vector consisting of $n+1$ individual time series representing the historical workload of an EDC_p under consideration, and its n neighbors. The sampling frequency W_{sample}, i.e., the time interval between two adjacent data points in a single time series, is set based on the fluctuation of the workload. To capture the rapid changes due to the mobility traces used in the experiment presented in the following section, we set the sample time to W_{sample} = *1 minute*. This is sufficient to complete up-/down-scaling actions for the applications we target, as described in Section II.

Taking Y_t as input, the workload arrival rate at EDC_p is predicted in the k-step-ahead interval $t+k, k \geq 1$. As described above, our aim is to quickly obtain a response prediction for every interval and also to prevent error accumulation when making multi-step-ahead predictions. We therefore set $k = 5$, meaning that the prediction interval is equal to *5 minutes*.

2) ***Performance Modeler****:* The performance modeler is responsible for estimating the number of resources required at each EDC to meet the SLOs. We assume that all Pods have the same hardware and host the same software application, and therefore deliver the same performance. At any given time, there is only one request being served by a Pod, the subsequent requests are queued. Each EDC can thus be modeled as a set of m $M/M/1/k/FIFO$ queues with a local load balancer. Here, m is the total number of Pods. Let k be the maximum number of requests that a Pod can accommodate (either in the queue or under service) to guarantee that admitted requests are served with an acceptable service time. The value of k can be computed as:

$$k = \left\lfloor \frac{T_{max}}{T} \right\rfloor$$

where T_{max} is the negotiated maximum response time defined by the SLOs and T is the average time required to serve a single request in a Pod.

The SLOs also define the target rejection rate $max\_rejectionRate\_thresh$. MEC must guarantee that its average resource utilization remains above a predefined $avg\_utilization\_thresh$ and that its rejection rate remains below $max\_rejectionRate\_thresh$. Using these values together with the predicted workload arrival rate $\hat{\lambda}$ and the monitored average request response time $T_{current}$, the controller can start to estimate the quantities of resources m that should be provisioned in each time interval using Algorithm 1. In line 2, m_{new} is initialized using equation 3. This guarantees that the new quantity of resources is close to that needed to meet the SLOs, which reduces the computing time for the following loop. Then, given m_{new} and the predicted workload arrival rate $\hat{\lambda}$, the system is modeled as a network of queues and the expected response time $\hat{T}_{average}$ and number of jobs in the queue $AveJobInQueue$ are estimated using the equations for an $M/M/1/k$ queue from [11]. If the SLOs are not met, m is recalculated and the process moves once again through the loop between lines 3 and 14.

$$m_{init} = \left\lceil \hat{\lambda} \frac{(1 - max\_rejectionRate\_thresh)}{avg\_utilization\_thresh} T_{current} \right\rceil \quad (3)$$

3) ***Resource Provisioner****:* The resource provisioner is responsible for determining the number of Pods to be scaled up/down at each EDC in a group. Unfortunately, it is impossible to predict workloads with complete accuracy, so some EDCs may be under-provisioned while others are over-provisioned. The idea is that requests arriving at an overloaded EDC can be redirected to other EDCs in the same group if doing so will not violate any SLOs. To improve system stability (i.e., to minimize the number of scale up/down actions while avoiding SLO violations), the final resource provisioning decision is made on the basis of the total resource demand at the group level. In other words, the resource provisioner is responsible for cross-evaluating the resource requirements of all EDCs in a group and setting a final number of desired resources at each EDC based on the current aggregated number of resources in that particular group.

The method used for group resource provisioning is outlined in Algorithm 2. It takes as inputs the value of m_i, the estimated number of Pods required at each EDC E_i in the group returned by Algorithm 1, and $m_{icurrent}$, the current number of Pods at each EDC in the group. First, it calculates the total required Pods and the total current Pods of the group, and the difference between the group's current and required Pods. If there is a requirement to scale down ($\delta < 0$), the algorithm scales down Pods, starting with EDCs having the lowest Pod requirements

Algorithm 1 Estimating the desired number of Pods at each EDC

1: **Input:** SLO metrics: T_{max} maximum response time, $max\_rejectionRate\_thresh$ maximum rejection rate
Input: $T_{current}, k, \hat{\lambda}$ ▷ $T_{current}$: average response time of current system. $\hat{\lambda}$: predicted arrival rate, $m_{current}$: current number of Pods, and k: queue length
Input: m_{max} ▷ maximum number of Pods in EDC
2: **Initialize:** m_{new}= equation(3)
3: **do**
4: $temp = m_{new}$
5: $\lambda_i = \frac{\hat{\lambda}}{m_{new}}$
6: estimate expected response time $\hat{T}_{average}$ in $M/M/1/k$ queue with $\lambda_i, T_{current}$
7: estimate $AveJobInPod$, the average job in an $M/M/1/k$ queue with $\lambda_i, T_{current}$
8: **if** $(AveJobInPod > (k-1))or(\hat{T}_{average} > T_{max})$ **then**
9: $m_{new} = m_{new} + \frac{m_{new}}{2}$
10: **if** $m_{new} > m_{max}$ **then**
11: $m_{new} = m_{max}$
12: **end if**
13: **end if**
14: **while** $temp! = m_{new}$
15: **return** m_{new}

(lines 4 to 15). Conversely, if there is a requirement to scale up ($\delta > 0$), the algorithm scales up Pods, starting with EDCs having the highest Pod requirements in the group (lines 16 to 27).

*4) **Group Load-balancer**:* The load-balancer is responsible for real-time balancing of requests from under-provisioned EDCs to over-provisioned EDCs in the same group. It does this using a *weighted round-robin* approach. Algorithm 1 computes the number of Pods required at each EDC in the group before load-balancing, while Algorithm 2 computes the actual number of Pods that should be allocated in each EDC to dampen scaling up and scaling down actions. Algorithm 3 then uses the outputs of these two algorithms to configure the weight vector W in which each W_i is the weight assigned to an over-provisioned EDC EDC_i. The weight W_i represents the proportion of requests redirected to EDC_i from each under-provisioned EDC in the group.

IV. Evaluation Setup and Methodology

This section presents our simulation-based evaluation process. We first explain how we simulate an MEC infrastructure in a metropolitan area. This is followed by descriptions of the approaches used to model the application of interest and its workload. Finally, we describe the implementation and setup of the tested controller, including both the proposed location-aware pro-active controller and a baseline re-active controller that is widely used in modern commercial clouds.

Algorithm 2 Group resource provisioning

1: **Input:** m_i ▷ number of Pods required at each EDC for the next time step, returned by algorithm 1
Input: $m_{icurrent}$ ▷ current number of Pods at each EDC
2: $requiredPod = \sum_{i=1}^{N} m_i$ ▷ N is the number of EDCs in the group
3: $currentPod = \sum_{i=1}^{N} m_{icurrent}$
4: **if** $requiredPod < currentPod$ **then**
5: sort EDCs in the group in increasing order of number Pods required.
6: $\delta = currentPod - requiredPod$
7: i = 0
8: **do**
9: $k = min(m_i, \delta)$
10: remove k Pods from EDC_i
11: $\delta = \delta - k$
12: **if** $\delta > 0$ **then**
13: $i = i + 1$
14: **end if**
15: **while** $\delta! = 0$
16: **else if** $requiredPod > currentPod$ **then**
17: sort EDCs in the group in decreasing order of number Pods required.
18: $\delta = requiredPod - currentPod$
19: i = 0
20: **do**
21: $k = min(m_i, \delta)$
22: add k Pods to EDC_i
23: $\delta = \delta - k$
24: **if** $\delta > 0$ **then**
25: $i = i + 1$
26: **end if**
27: **while** $\delta! = 0$
28: **end if**

Algorithm 3 Calculating weights for over-provisioned EDCs

1: **Input:** m_{1i} ▷ number of Pods required at each EDC for the next time step, returned by Algorithm 1
Input: m_{2i} ▷ number of Pods required at each EDC for the next time step, returned by Algorithm 2
2: $\Delta = \sum_{i=1}^{K} m_{2i} - m_{1i}$ ▷ K is number of EDCs needing to be scaled down in the next time interval according to Algorithm 1
3: **for** $i \leftarrow 1$ to K **do**
4: $W_i = \frac{m_{2i} - m_{1i}}{\Delta}$ ▷ W_i is the weight assigned to EDC_i
5: **end for**
6: return W_i

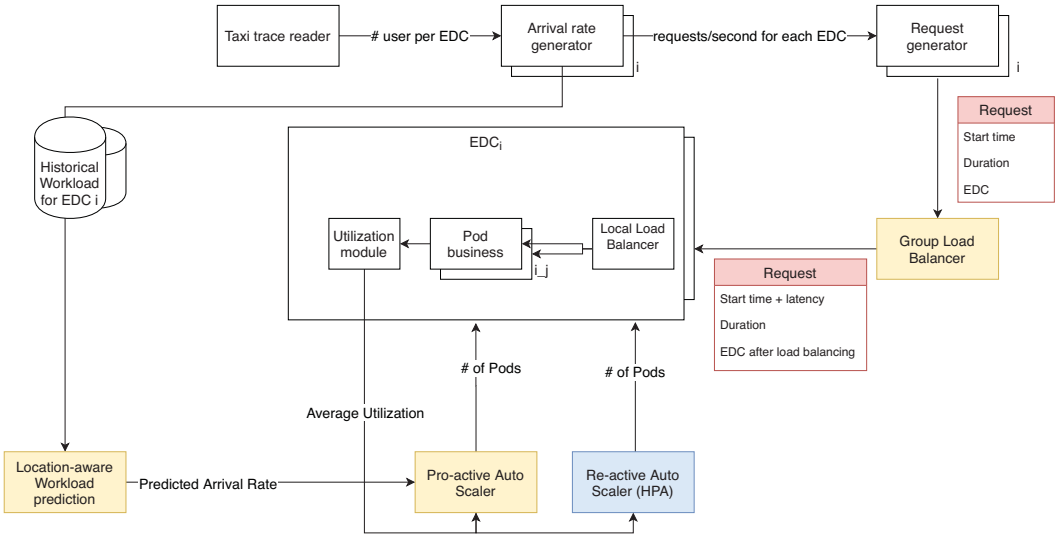


Fig. 3: The experimental simulation.

A. Infrastructure: An MEC Distributed Over a Metropolitan Area

We simulate a single MEC distributed over a metropolitan area. To model this MEC, we captured real geo-coordinate information on cellular towers distributed across San Francisco, US[2]. Duplicated towers (i.e., towers with geographical coordinates identical to another tower) were removed, leaving 15 unique cellular towers. The MEC is modeled by supposing that one EDC is collocated with each unique tower to provide coverage of the studied area. Figure 4 shows the distribution of these 15 EDCs in San Francisco. For simplicity, we assume that the EDCs are connected to a metropolitan area network with unlimited bandwidth and no delay in the middle layers (i.e., the routers, switches, and network cards). The network delay between each pair of EDCs E_p and E_q is thus a function of the distance between them:

$$delay(E_p, E_q) = D * distance(EDC_p, EDC_q)$$

Here, D is set to 5 $\mu s/km$ [7] with a positive variation of 10%, uniformly distributed; and $distance(EDC_p, EDC_q)$ is the distance between the two EDCs, which is computed from their geographical coordinates (i.e., longitude and latitude) using the *Haversine formula* [12].

We assume that the resources of each EDC are virtualized according to the Kubernetes model, under which resources are provisioned and shared flexibly. Each individual instance of a given application is encapsulated in a manageable disposable entity called a Pod, which wraps the container and storage resource together. Horizontal scaling of an application thus entails increasing or decreasing the number of replicated Pods.

[2]http://www.city-data.com/towers/cell-San-Francisco-California.html

B. Application: An Extremely Latency-Intolerant AR Application

We assume that an AR application is deployed on the MEC to provide services to end-users in the area. Each end-user's request, if admitted, is served by an application instance running on a Pod allocated by EDCs. Each application instance serves several requests on a first come first served basis to avoid unnecessary startup times. The aggregate time of the Pod startup time, the waiting time, the service time, and the network communication delay constitutes the final application's response time. As noted in section II, it is important to maintain a low response time in order to guarantee an acceptable end-user experience that does not induce motion sickness or dizziness. Some AR applications (e.g., optical head-mounted display) require a latency less than 2.5 ms of motion-to-display lag [9]. We therefore model the application with the service time is 1 ms with a positive variation of 10%, uniformly distributed; and the maximum response time of the application is set to 2.5 ms. Since the main objective is to evaluate the efficiency of the proposed elastic control, we conduct experiments with a single application. However, it is worth noting that the experiment can be easily extended to multiple applications, each of which is modeled with a different predefined response time requirement. Due to the large amount of data that must be copied in when creating a new Pod, we assume that the actuation delay, i.e., the amount of time a new Pod requires to become ready, is 1 minute with a positive variation of 10%, uniformly distributed.

C. Workload: Real Taxi Mobility Traces

To simulate workload arrival at EDCs, we use real mobility traces for taxis in San Francisco [13]. The main reason we chose these traces is that they cover the same geographic area

as the data used to generate the geographic distribution of the MEC.

The dataset contains information about the mobility behavior of 536 taxi cabs in the San Francisco Bay area over 25 days starting on May 17th, 2008. The original dataset contains 4 attributes: longitude, latitude, datetime, and number of people in the car. We preprocess the trace to remove all invalid records by setting a maximum velocity of $V_{max} = 115km/h$ based on the relevant urban traffic regulations. We then eliminate records reflecting a velocity greater than V_{max}. Since there are only 536 cabs in the dataset, the generated workload would be rather sparse and small, and would not represent a typical throughput for an AR application of the type under consideration (i.e. the number of requests would be too low to keep the application busy during the experimental period). We therefore increase the magnitude of the workload using the function below to generate a more dense workload for the EDCs:

$$newWorkload = (originalWorkload + t) * k$$

Here, t, and k are predefined constants. For the experiment, t is set to 25 and k is set to 160 000. It is worth noting that this function is linear, so it increases the workload size without impacting the intrinsic characteristics of the vehicles' original mobility behavior.

D. Auto-scaler Setup

To conduct representative experiments, the proposed controller method is configured as follows: We divide the MEC into 4 groups, with each group consisting of a set of EDCs located in close proximity to one-another, as shown in Table I. The system periodically monitors and records request arrival rate data in the historical dataset for each EDC, aggregated over *1-minute* sampling intervals. The first 17 days of the historical dataset (starting from May 17th, 2008) are used as a training set for the predictive model. The trained model is then used to predict the request arrival rate at each EDC for the last 7 days covered by the dataset (June 3rd, 2008 to June 9th, 2008) using a *5-minute* prediction interval (i.e., the look-ahead step is set to 5). At runtime, the model is constantly updated by incorporating newly observed data into the input dataset and removing the same quantity of old observation data, with the oldest data being removed first. The training is then repeated using the updated training set. The predicted arrival rate, the target SLOs, and the average utilization are used as inputs for the Pro-active Auto scaler (i.e., the Performance modeler, and the Resource Provisioner as detailed in Figure 2), which is used to estimate the number of Pods required in the next time window. The target average resource utilization at each EDC is set to $avg\_utilization\_thresh = 80\%$, and the upper limit for the rejection rate is set to $max\_rejectionRate\_thresh = 1\%$.

To characterize the performance of the proposed controller and the impact of its individual components, we perform experiments using two controller settings:

TABLE I: Group settings.

GroupID	EDCs
#1	#1, #2, #3, #5, #10
#2	#8, #12, #15
#3	#11, #14
#4	#4, #6, #7, #9, #13

Pro-active AS, in which the controller uses only the pro-active auto-scaler (i.e., the group load balancer is disabled); and

Pro-active AS + LB, in which the controller uses both the pro-active auto-scaler and the group load-balancer.

E. Baseline: Re-active Auto-Scaler

We implement a baseline auto-scaler that mimics the Kubernetes Horizontal Pod Auto-Scaler - a widely-used, battle-tested, production-ready, re-active auto-scaling tool that Kubernetes deploys for horizontal scaling of its replicas [14]. In this case, the resource utilization at each EDC_i in time window w is:

$$Utilization(EDC_i)_w = \sum_{i}^{n} \frac{Utilization(Pod_i)_w}{m}$$

Here, $Utilization(Pod_i)_w$ is the function used to calculate the utilization of Pod i, which is defined by the percentage of the total busy time that Pod i spends serving requests over time window w. m is the total current number of Pods (including both "*ready*" and "*not-yet-ready*" Pods) in all EDCs at timestamp t.

The other parameter settings used in this case are presented in Table II. These settings are the defaults for Kubernetes. In essence, for each period ($periodSecond = 15\ seconds$), the controller periodically queries the resource utilization of each EDC in the preceding time window ($w = 30\ seconds$) and calculates the *scale ratio* using the logic presented in [14]. If the ratio is within the *tolerance* (set to 10% by default), no scaling is performed. Otherwise, if there is a scaling up signal, the controller will immediately add the number of replicas desired to make the observed resource utilization reach or exceed the target specified by the end-users. The *scaleUpLimitFactor* and *scaleUpLimitMinimum* are set to 2.0 and 4.0 respectively to specify how quickly EDCs can scale up. To smooth out the impact of rapidly fluctuating metric values (i.e., to prevent rapid fluctuations in the number of replicas), scale down actions are tuned to occur gradually; by default, the *downscale-stabilization* is set to 5 minutes, meaning that the controller will select the highest desired number of Pods recommended within the last 5 minutes.

F. Performance Metrics for Comparing Auto Scalers

To meaningfully and quantitatively compare the performance of different auto scaling methods, we use the three

TABLE II: Parameter settings used in the experiments.

MECs	
number of EDC	15
maximum Pods at each EDC	300
minimum Pods at each EDC	5
Pod readiness delay	1 minute
Application	
average response time	1 ms
maximum response time	2.5 ms
Targeted threshold	
avg_utilization_thresh	80%
max_rejectionRate_thresh	1%
Proposed Pro-active auto scaler settings	
number groups	4
look-ahead arrival rate prediction	5 minutes
Re-active auto scaler setting	
downscale-stabilization	5 minutes
periodSeconds	15 seconds
scaleUpLimitFactor	2.0
scaleUpLimitMinimum	4.0
tolerance	10%

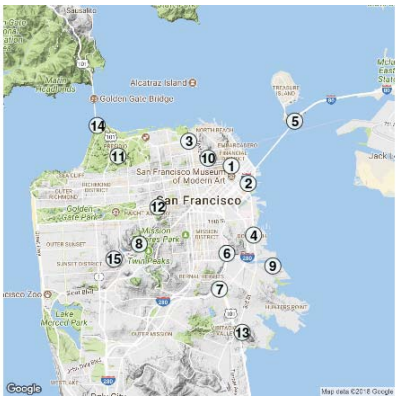


Fig. 4: Distribution of EDCs in San Francisco.

main elasticity metrics mentioned in Section II together with a set of system- and user-oriented metrics recommended by the research group of the Standard Performance Evaluation Corporation (SPEC) [15]. Given an experiment duration T, we use s_t and d_t to denote the resource supply and resource demand at each timestamp $t \in [0, T]$ respectively. Below, we summarize the metrics used to measure aspects of elasticity suggested in [15].

– Two accuracy metrics θ_U, and θ_O are introduced to measure the deviation (in resource units) between resource demand and the resource supply based on the auto-scaler's decisions:

Under-Provisioning accuracy, θ_U: The number of resource units that must be added for s_t to match d_t, normalized against the duration of the experiment, T. In other words, the metric θ_U is calculated as the sum of the areas where the resource demand exceeds the supply.

$$\theta_U[\text{resource units}] = \sum_{t=1}^{T} \frac{\max(d_t - s_t, 0)}{T}$$

Over-provisioning accuracy, θ_O: Analogously, the over-provisioning accuracy metric θ_O is the number of resource units supplied by an auto-scaler in excess of the demand, normalized against the experiment's duration.

$$\theta_O[\text{resource units}] = \sum_{t=1}^{T} \frac{\max(s_t - d_t, 0)}{T}$$

θ_U and θ_O both take values in $[0, +\infty]$; for both metrics, values closer to 0 are better (values of 0 mean that no under-provisioning or over-provisioning occurs during the experiment).

– To measure the proportion of the total experimental time during which the system is under- or over-provisioned, two *timeshare* metrics, τ_U and τ_O are introduced:

Under-Provisioning timeshare τ_U: The total amount of time that the system spends in an under-provisioned state, normalized against the overall experiment duration T.

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

Over-Provisioning timeshare τ_O: The total amount of time that the system spends in an over-provisioned state, normalized against the experiment duration T.

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

Both τ_U and τ_O take percentage values within $[0, 100]$, with values closer to 0 being better; values of 0 indicate that no under- or over-provisioning are detected during the experiment duration T.

– To determine whether the supply curves generated by an auto-scaler change in the same direction as the demand curve, we use the *instability* υ metric:

$$\upsilon[\%] = \frac{100}{T - t_1} \sum_{t=2}^{T} \min(|sgn(\Delta_{s_t}) - sng(\Delta_{d_t})|, 1)\Delta_t$$

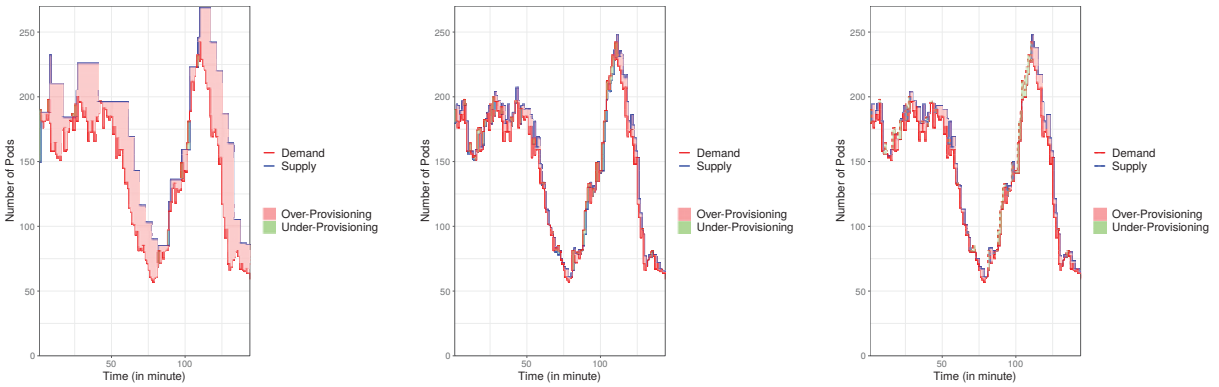
υ takes values in the interval $[0, 100]$, with values closer to 0 being better; a value of 0 indicates that the demand and supply curves always move in the same direction.

V. Experimental Result and Discussion

In this section, we present the experimental results obtained with the two configurations of the proposed pro-active elasticity controller and the baseline re-active controller.

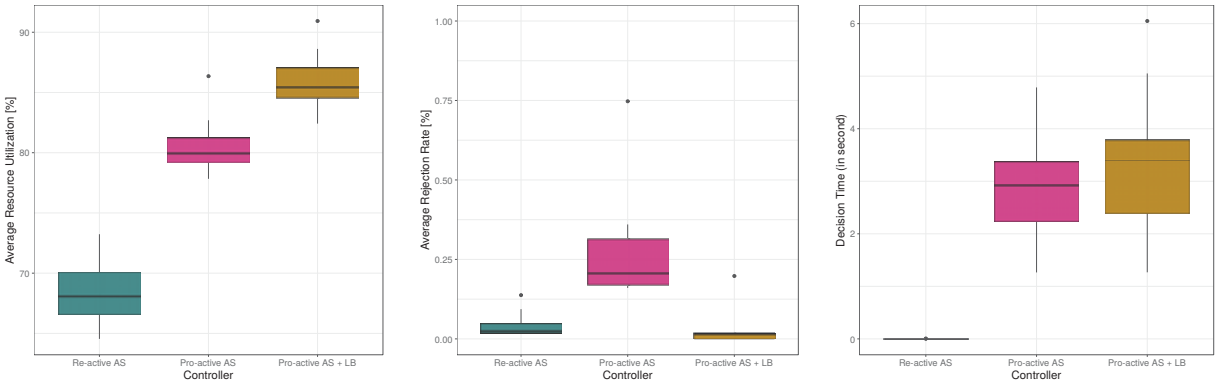
A. *How does the proposed pro-active controller perform when compared to the re-active controller?*

Our objective is to quantify the efficacy of the proposed controller and compare its scaling behavior to that of the currently popular re-active controller. To this end, we conduct experiments with three controller configurations as described in the preceding section. Table III summarizes the quantitative results for each controller configuration with respect to the elasticity metrics discussed in the previous section.



(a) Re-active AS (b) Pro-active AS (c) Pro-active AS + LB (the red dashed curve: the demand before load balancing; the red curve: the new demand after load balancing)

Fig. 5: The scaling behavior of three controllers on EDC#1 (*avg_utilization_thresh*= 80%, *max_rejectionRate_thresh* = 1%).



(a) Average resource utilization. (b) Average rejection rate. (c) Average decision time.

Fig. 6: Performance of the three controllers based on the main elasticity metrics (*avg_utilization_thresh*= 80%, *max_rejectionRate_thresh* = 1%).

When using the *Pro-active AS + LB* controller configuration, the system maintains an average resource utilization of 85.9%, while the average rejection rate is 0.02%. The *Pro-active AS* configuration also helps the system avoid breaching the predefined thresholds: the average resource utilization in this case is above 80% and the rejection rate is below 1%. However, the *Pro-active AS* configuration creates a total of 4405 Pods with an average lifetime of approximately 35.2 minutes, whereas the *Pro-active AS + LB* configuration creates fewer Pods (3154 Pods in total) with a longer average lifetime (73.3 minutes). Of the three studied controller configurations, *Re-active AS* exhibits the worst scaling behavior. Although its rejection rate does not breach the user-specified threshold, it yields a low level of resource utilization (68.4%) and creates more Pods than either of the pro-active controller configurations. These results together with the elasticity metrics $(\theta_U, \theta_O, \tau_U, \tau_O)$ presented in Table III reflect the intrinsic behavior of the Kubernetes horizontal Pod auto-scaler, which is designed to compensate for a lack of predictive capacity by provisioning more resources than currently demanded.

This conclusion is supported by the results presented in Figure 5, which shows the scaling behavior of the three controller configurations in EDC#1. The *Re-active AS* configuration clearly causes EDC#1 to be over-provisioned for most of the experimental period, whereas the two *Pro-active AS* configurations yield a better match between resource demand and supply. The supply curve plotted in Figure 5c also exhibits fewer oscillations than that in Figure 5b. Additionally, at time points when EDC#1 is short of resource capacity, the group load balancer redirects requests arriving at this EDC to other EDCs (i.e., #2, #3, #5, #10) in the same group. Two demand curves are plotted in Figure 5c (red dashed curve: the demand before load balancing; red curve: the new demand after load balancing) showing that the demand curve obtained with load

TABLE III: The performance of the three controllers based on the elasticity metrics (*avg_utilization_thresh*= 80%, *max_rejectionRate_thresh* = 1%).

Metric	Pro-active AS + LB	Pro-active AS	Re-active AS
θ_U	13.6	41.2	**5.4**
θ_O	**14.2**	39.5	305.6
τ_U	**4%**	43%	5.3%
τ_O	**2.5%**	46.7%	94.1%
υ	**2.44%**	2.8%	3.9%
Avg. resource utilization	**85.9%**	80.5%	68.4%
Rejection rate	**0.02%**	0.26%	0.04%
total Pods	**3154**	4405	5337
Avg. Pod lifetime (minute)	**73.3**	35.2	29.6

balancing matches the supply curve even at time points when EDC#1 would otherwise be short of capacity.

The admitted requests are guaranteed to meet the response time requirement. Indeed, Figure 7 presents the cumulative density of the response time of the admitted requests in experiments with the three different elasticity control settings. It is easy to observe that, for the system with the *Re-active AS*, most admitted requests are served with response times around 1 ms. This is due to the *Re-active AS* generously allocating more resources than the demand. Whereas, due to maximizing resource utilization, the system with *Pro-active AS* and *Pro-active AS + LB* controllers allocate a number of Pods which are close to the demand, resulting in some admitted requests requiring some waiting time before being served. This leads to a slight increase in the final response times. In the system employed *Pro-active AS + LB*, some admitted requests are redirected among EDCs in a group, hence the aggregated response time is further increased by the network delay between the connecting EDC and the serving EDC. As a result, the average response time of the admitted requests in this setting is larger than those observed in the system with the *Re-active AS* and the *Pro-active AS* settings. Nonetheless, with the response time for the admitted requests satisfy the SLOs, and the low rejection rate, the proposed controller helps MECs guarantee the user's experience for the deployed AR application.

To summarize, of the three studied controller configurations, *Pro-active AS + LB* exhibits the best behavior in general, giving the highest average resource utilization, a low rejection rate, and high stability.

B. To what degree does location-awareness improve scaling behavior?

The proposed controller estimates the number of Pods desired at each EDC based on the request arrival rate predicted by the workload prediction model. Its performance thus depends strongly on the accuracy of the predictive model. At every time point, a more accurate arrival rate prediction will lead to a better estimation of the required number of Pods. The finding in [6] revealed that the variation in the workloads of

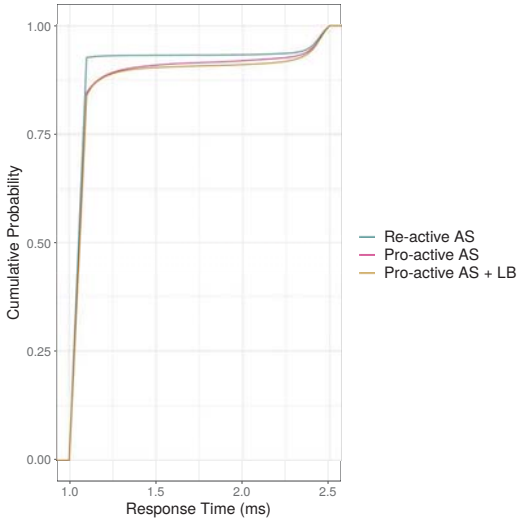


Fig. 7: Cumulative density of response times of the application in three elastic control settings.

neighboring EDCs is correlated, which can be exploited to improve the accuracy of workload prediction in MECs. However, the increase in predictive accuracy becomes saturated once the number of neighboring EDCs exceeds a certain threshold. To verify the impact of varying the number of neighbors on the scaling behavior of the pro-active controller, we conduct another experiment in which we vary the group size, k. Two values of k are considered:

k = 1: one pro-active controller is deployed on each EDC, and each controller is only responsible for estimating the number of Pods for the EDC on which it is deployed. Additionally, the only input used by the request arrival rate prediction model is the historical workload of that EDC. The controllers in this scenario can be considered **location-unaware** because they cannot use information on the correlation of workload variation between neighboring EDCs. The results obtained using this controller configuration serve as a baseline for

comparison to the performance achieved with pro-active auto scalers of the type used in current cloud computing platforms.

k = 15: all 15 EDCs of the emulated MEC are assigned to a single group. The prediction model takes historical data for all 15 EDCs as its input, and the group load balancer is responsible for balancing loads from overloaded EDCs to other EDCs in the group. The elastic controller in this case is therefore referred to as a **location-aware** controller.

We use three metrics: the rejection rate, the resource utilization, and the lifetime of invoked Pods to measure the system's scaling behavior when using the proposed controller under varying conditions and to compare it to that achieved with the re-active controller. Figure 8 plots the performance of the three studied controller configurations in the $k = 1$ and $k = 15$ cases as well as the original case where the EDC groups are defined based on physical proximity. The performance data are plotted using radar charts with three axes corresponding to the three elasticity metrics. As shown in Figure 8a, in the $k = 1$ case, the system with the pro-active controller only reaches an average resource utilization of 74%, which is below the predefined resource utilization threshold. This is because the predictive model in this case is learned based on the historical workload of the predicted EDC only, hence it cannot accurately predict the request arrival rate, leading to inaccurate estimation of the required number of Pods. For the group including all 15 EDCs ($k = 15$), the *Pro-active + LB* controller achieves the average measured resource utilization (80.2%) exceeds the target resource utilization, and the rejection rate (0.3%) less than the maximum rejection rate threshold, as shown in Figure 8c. However, the performance with respect to all three metrics is worse than in the case where EDCs are grouped based on physical proximity. This is because the workload variation of physically distant EDCs is uncorrelated, causing "noise" in the training data for learning the predictive model, which reduces the accuracy of the workload predictions and thus reduces the accuracy of the estimates of the number of Pods required to meet future demand.

To summarize, the correlation in workload variation among EDCs in close physical proximity improves the accuracy of the location-aware prediction model, which in turn enables more accurate estimation of the number of Pods required to meet performance objectives.

C. *What is the decision time of the elastic controller?*

To verify the applicability of the proposed controller, we measure its running time on each occasion it is invoked. These experiments were performed on a PC with an Intel i7-4790 CPU and 32GB RAM using a single thread.

Figure 6c presents decision times for the three controllers for every 5-minute time window during the experiment's duration. *Re-active AS* is the fastest of the three controller configurations, with an average decision time of approximately 0.05 seconds for each scaling action. This is understandable because the re-active controller chooses scaling actions based on records of the system's current resource utilization. Conversely, the two pro-active configurations require some time to predict the future workload and estimate the current system performance. Therefore, the average decision time of the *Pro-active AS* controller is approximately 2.84 seconds. The decision time of the *Pro-active AS + LB* controller (3.13 seconds) is slightly higher still because it must perform the additional task of configuring load balancing. However, recalling that the scaling actions are implemented at 5-minute intervals, one can reasonably conclude that the pro-active controllers make decisions with acceptable speed, leaving ample time for the newly provisioned resources to be started-up and made ready to use.

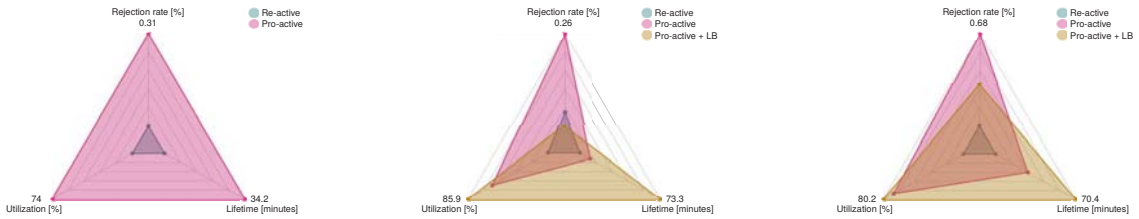
D. *What is the impact of the two predefined thresholds on the controller's scaling behavior?*

Our proposed controller estimates the number of Pods required in the coming interval based on two predefined thresholds: the target resource utilization and the target rejection rate. To explore the impact of these thresholds on the final scaling behavior, we conduct experiments in which their values are varied. In the first experiment, we keep the target rejection rate constant (i.e., *max_rejectionRate_thresh = 1%*) and only change the target resource utilization (gradually increasing it from 70% to 95%). Conversely, in the second experiment the target resource utilization is constant (i.e., *avg_utilization_thresh = 80%*), while the target rejection rate is decreased from 10% to 1%. Figure 9 shows the controller's scaling behavior based on two main metrics: the measured resource utilization, and the measured rejection rate. Table IV presents the measured values of the resource utilization, the rejection rate, and the total number of Pods invoked, which describes the scaling behavior of the proposed controller. When the controller tries to improve the average resource utilization, its scaling decisions are made in part with the aim of dampening the rate of change in resource supply relative to the current demand. This makes the system unable to keep the rejection rate below the target level, as indicated by the bolded values in Table IV. At the other extreme, when the controller aims to minimize the rejection rate, it generously provides more resource than are needed to meet demand, causing a decline in resource utilization.

VI. Related Work

Over the last decade, considerable efforts have been invested into resource management research with the aim of developing reliable, robust, and cost- and energy-efficient distributed computing environments. Diverse mechanisms and techniques have been used to implement elasticity systems, ranging from reactive statistical approaches to pro-active methods based on machine learning [16], [17].

The *reactive rule-based approach* is a resource scaling method that has become popular among commercial cloud providers because of its straightforward deployment. This method requires cloud providers to offer a user interface that lets clients define performance metric thresholds (based on, e.g., resource utilization, rejection rates, or service response times) and scaling plans. Resources are then automatically

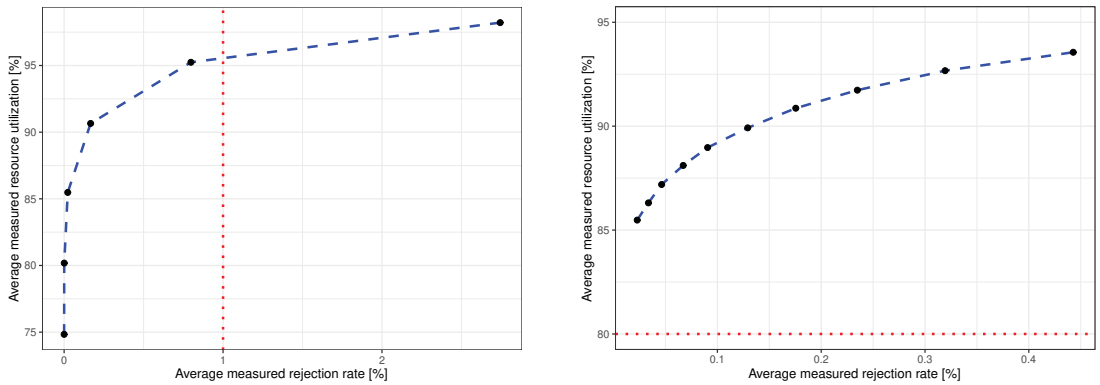


(a) Groups consisting of 1 EDC only ($k = 1$).

(b) Groups with neighboring EDCs as specified in table I.

(c) Single group consisting of all 15 EDCs ($k = 15$).

Fig. 8: Performance of the three studied controller configurations based on the three major elasticity metrics when the number of neighboring EDCs is varied ($avg\_utilization\_thresh = 80\%$, $max\_rejectionRate\_thresh = 1\%$).



(a) The targeted resource utilization is changed, while the targeted rejection rate is held constant at 1%.

(b) The targeted rejection rate is changed, while the targeted resource utilization is held constant at 80%.

Fig. 9: The controller's scaling behavior when varying the threshold settings.

added or removed based on these predefined conditions. Rule-based auto-scalers of this type are used by Kubernetes and Amazon AWS [14], [18]. The limitation of these rule-based approaches is that they are reactive: they only scale resources *after* a change in performance metrics has been detected. Therefore, end-users may experience poor performance until the extra resources become available.

In contrast to reactive methods, pro-active self-adaptive mechanisms use various prediction techniques to improve efficiency. Ali-Eldin et al. [19] proposed a horizontal auto-scaler that applies *control theory* and *queueing theory* to model system performance and estimate future resource demand. Their method uses both a reactive and a pro-active controller that dynamically change the number of virtual machines allocated to services based on the current performance and the predicted future demand. Similarly, Bauer et al. [20] introduce a hybrid pro-active auto-scaler, called Chameleon, with two sub-functions (one pro-active and one reactive). Chameleon uses *time series analysis* to forecast the arriving load intensity and *queue theory* to estimate service demand. Extensive experiments using both private and public clouds revealed that this approach achieves efficient scaling performance.

At the other extreme are approaches that use trial and error to converge on an optimal policy [21], [22] for making scaling decisions in specific states. Such approaches require no a priori knowledge or explicit models of either the system or its current workload. Grimadi et al. [21] introduced a feedback control strategy to adaptively allocate cloud resources to users of public clouds in a way that guarantees that Service Level Objectives are met. Their approach uses *gain scheduling* in which adaptive control is optimized using an automatic optimal tuning procedure. Similarly, Rao et al. [22] developed VCONF, a reinforcement learning-based approach for virtual machine auto-scaling. Experiments on a Xen-based testbed showed that

TABLE IV: The scaling behavior of the proposed controller with different predefined threshold settings.

Targeted rejection rate[%]	Targeted resource utilization[%]	Measured resource utilization[%]	Measured rejection rate[%]	Total Pods
1	70	74.8	0	3812
	75	80.2	7e-4	3484
	80	85.9	0.02	3154
	85	90.6	0.16	2890
	90	95.2	0.8	2653
	95	**98.2**	**2.7**	**1995**
10	80	93.5	0.44	2753
...		...	...	...
3		87.2	0.05	3065
2		86.3	0.03	3113
1		85.9	0.02	3154

this approach offers good adaptability and scalability. For fine-grained container-level cloud resource management, Nguyen et al. [23] proposed MONAD, a self-adaptive micro-service based framework for scientific workloads. MONAD employs neural network to predict the performance of the scientific workflow deployed on cloud resources. A feedback control mechanism helps the cloud system dynamically scale its resources, securing the QoS requirement without any advanced knowledge of workflow structures.

While such approaches have achieved remarkable efficiency gains when applied to the resource provisioning challenge in modern clouds, none of them was specifically designed to take advantage of user mobility-driven workload correlation across the EDCs of an MEC. Our proposed elasticity controller takes into consideration the impact of user mobility on the workload variation in physically neighboring EDCs to build a location-aware elastic controller for MECs. The controller has two main functions: a pro-active auto-scaler to determine how many resources to allocate to each EDC, and a group-level load-balancer that balances workloads on the fly, reallocating them from underprovisioned EDCs to nearby EDCs with available resources.

VII. Conclusion and Future Work

Because of user mobility and the resulting heterogeneity of resource distribution, it is very difficult to control resource allocation in Mobile Edge Clouds in a way that optimizes resource use efficiency while also delivering expected end-user QoS. In this paper, we introduce a local-aware pro-active controller for dynamic provisioning of resources to services running in MECs. The proposed controller takes advantage of the correlation of workload variation in physically neighboring EDCs to predict the request arrival rate at EDCs. These predictions are then used as inputs to estimate service demand and the number of resources that will be desired at each EDC. Additionally, to minimize the request rejection rate, we develop a group-level load balancer to redirect requests among EDCs during runtime. Experiments using an emulated MEC over a metropolitan area, and simulated application workloads from a real mobility trace, we showed that our proposed controller delivers significantly better scaling behavior than a state-of-the-art re-active controller and also improves the efficiency of resource provisioning. This in turn helps MECs maintain resource utilization and request rejection rates that satisfy predefined requirements while maintaining system stability.

In future, we plan to test our approach on an emulated MEC using a diverse dataset obtained from mobile network operators. We see the potential to extend our proposed controller in different ways: we will work on designing a hybrid controller that incorporates both pro-active and re-active control together with an applied control theory technique to help the system dynamically switch between the controllers so as to further improve scaling behavior.

Acknowledgements

The authors would like to thank Dr. Utsav Drolia (from NEC Labs America) and the anonymous reviewers for their helpful and constructive suggestions that greatly contributed to improving the final version of the paper. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] Jianli Pan and James McElhannon. Future edge cloud and edge computing for internet of things applications. *IEEE Internet of Things Journal*, 5(1):439–449, 2017.

[2] SM Riazul Islam, Daehan Kwak, MD Humaun Kabir, Mahmud Hossain, and Kyung-Sup Kwak. The internet of things for health care: a comprehensive survey. *IEEE Access*, 3:678–708, 2015.

[3] Sabina Jeschke, Christian Brecher, Tobias Meisen, Denis Özdemir, and Tim Eschert. Industrial internet of things and cyber manufacturing systems. In *Industrial Internet of Things*, pages 3–19. Springer, 2017.

[4] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[5] Chanh Nguyen. Autonomous resource management for mobile edge clouds, 2019.

[6] Chanh Nguyen, Cristian Klein, and Erik Elmroth. Multivariate lstm-based location-aware workload prediction for edge data centers. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 341–350, 2019.

[7] Vjaceslavs Bobrovs, Sandis Spolitis, and Girts Ivanovs. Latency causes and reduction in optical metro networks. In *Optical Metro Networks and Short-Haul Systems VI*, volume 9008, page 90080C. International Society for Optics and Photonics, 2014.

[8] Mahdi Nabiyouni, Siroberto Scerbo, Doug A Bowman, and Tobias Höllerer. Relative effects of real-world and virtual-world latency on an augmented reality training task: an ar simulation experiment. *Frontiers in ICT*, 3:34, 2017.

[9] Randall E Bailey, Jarvis James Arthur III, and Steven P Williams. Latency requirements for head-worn display s/evs applications. In *Enhanced and Synthetic Vision 2004*, volume 5424, pages 98–109. International Society for Optics and Photonics, 2004.

[10] Philipp Schulz, Maximilian Matthe, Henrik Klessig, Meryem Simsek, Gerhard Fettweis, Junaid Ansari, Shehzad Ali Ashraf, Bjoern Almeroth, Jens Voigt, Ines Riedel, et al. Latency critical iot applications in 5g: Perspective on the design of radio interface and network architecture. *IEEE Communications Magazine*, 55(2):70–78, 2017.

[11] Leonard Kleinrock. *Queueing systems, volume 2: Computer applications*, volume 66. Wiley New York, 1976.

[12] C Carl Robusto. The cosine-haversine formula. *The American Mathematical Monthly*, 64(1):38–40, 1957.

[13] Michal Piorkowski, Natasa Sarafijanovic-Djukic, and Matthias Grossglauser. CRAWDAD dataset epfl/mobility (v. 2009-02-24). Downloaded from https://crawdad.org/epfl/mobility/20090224, February 2009.

[14] Horizontal Pod Autoscaler. https://v1-17.docs.kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, Jan. 2020. Accessed: 2020-01-18.

[15] Nikolas Herbst, Rouven Krebs, Giorgos Oikonomou, George Kousiouris, Athanasia Evangelinou, Alexandru Iosup, and Samuel Kounev. Ready for rain? a view from spec research on the future of cloud metrics. *arXiv preprint arXiv:1604.03470*, 2016.

[16] Guilherme Galante and Luis Carlos E de Bona. A survey on cloud computing elasticity. In *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, pages 263–270. IEEE, 2012.

[17] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[18] Amazon Web serivce Auto Scaling. https://aws.amazon.com/autoscaling/, 2018. Accessed: 2020-04-01.

[19] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *2012 IEEE Network Operations and Management Symposium*, pages 204–212, April 2012.

[20] André Bauer, Nikolas Herbst, Simon Spinner, Ahmed Ali-Eldin, and Samuel Kounev. Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):800–813, 2018.

[21] Domenico Grimaldi, Valerio Persico, Antonio Pescapé, Alessandro Salvi, and Stefania Santini. A feedback-control approach for resource management in public clouds. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7, Dec 2015.

[22] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. Vconf: a reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of the 6th international conference on Autonomic computing*, pages 137–146, 2009.

[23] Phuong Nguyen and Klara Nahrstedt. Monad: Self-adaptive microservice infrastructure for heterogeneous scientific workflows. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 187–196. IEEE, 2017.