A Unified Model for the Mobile-Edge-Cloud Continuum

L. BARESI, D. F. MENDONÇA, M. GARRIGA, S. GUINEA, and G. QUATTROCCHI,
Politecnico di Milano, Italy

Technologies such as mobile, edge, and cloud computing have the potential to form a computing continuum for new, disruptive applications. At runtime, applications can choose to execute parts of their logic on different infrastructures that constitute the continuum, with the goal of minimizing latency and battery consumption and maximizing availability. In this article, we propose A3-E, a unified model for managing the life cycle of continuum applications. In particular, A3-E exploits the Functions-as-a-Service model to bring computation to the continuum in the form of microservices. Furthermore, A3-E selects where to execute a certain function based on the specific context and user requirements. The article also presents a prototype framework that implements the concepts behind A3-E. Results show that A3-E is capable of dynamically deploying microservices and routing the application's requests, reducing latency by up to 90% when using edge instead of cloud resources, and battery consumption by 74% when computation has been offloaded.

Received December 2017; revised April 2018; accepted May 2018

1 INTRODUCTION

Mobile devices and edge and cloud computing have the potential to form a *computing continuum* on which new and disruptive types of applications can be built. This continuum enables the convergence of heterogeneous infrastructures, stretching all the way from cloud to mobile devices, including intermediate steps such as ISP gateways, cellular base stations, and private cloud deployments.

The heterogeneity of the computing continuum is profound and multifaceted. In the cloud, computing resources are typically provided through virtualization and containerization [5, 17], and

This work was supported with grant by *National Council for Scientific and Technological Development (CNPq) - Brazil*, by project *EEB - Edifici A Zero Consumo Energetico In Distretti Urbani Intelligenti (Italian Technology Cluster For Smart Communities) - CTN01_00034_594053* and by the GAUSS national research project (MIUR, PRIN 2015, Contract 2015KWREMX).
Authors' addresses: L. Baresi, D. F. Mendonça, M. Garriga, S. Guinea, and G. Quattrocchi, Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Building 22, Floor 3, Milan, MI, 20133, Italy; emails: {luciano.baresi, danilo.filgueira}@polimi.it, mgarriga87@gmail.com, sam.guinea@icloud.com, giovanni.quattrocchi@polimi.it.

there is an illusion of infinite resource availability thanks to horizontal scaling. In contrast, in edge computing, computational resources are scarce and must be managed very efficiently [6, 27]. This is even truer for mobile devices, as they are strongly constrained by battery and other limitations. Cloud resources are accessible by clients across countries and continents; conversely, in edge computing, a client can only access resources that are under the same network coverage, be it cellular (e.g., 5G) or local (e.g., domestic or office). On the other hand, we can consider the computational resources of a mobile device as always accessible, as long as it has a battery. While the cloud can provide vast computing power through elasticity, accessing these resources may involve multiple hops of network communication, leading to prohibitive latency in the processing of client requests. Indeed, one of the main motivations for introducing edge-based computation is to mitigate network latency [8, 24, 27].

In this article, we propose A3-E, a unified model for the realization of the mobile-edge-cloud continuum. A3-E takes its name from its four main activities: *(A)wareness, (A)cquisition, (A)llocation*, and *(E)ngagement*. The proposed model exploits the Functions-as-a-Service (FaaS) computing paradigm [4, 6, 11] to allow stateless and lightweight functions to be autonomously fetched, deployed, and exposed as microservices by heterogeneous providers. A3-E conciliates providers' goals and client applications' needs with the efficient and scalable management of the life cycles of continuum microservices. Since distinct providers and infrastructures will not be able to autonomously coordinate and decide who should serve a client request, A3-E enables a mutual client-provider awareness that allows for the opportunistic and context-dependent placement of microservices along the continuum.

The feasibility of A3-E has been demonstrated by means of a prototype infrastructure. Also, A3-E has been evaluated in the context of an Augmented Reality application. Thanks to A3-E, the application was able to autonomously proxy its requests to services that were dynamically deployed to a computing continuum. Performed experiments show up to a 90% reduction of latency when edge replaced cloud, and a 74% decrease of battery consumption when computation is offloaded to edge/cloud servers. Moreover, by dynamically selecting what constituents to use in the continuum in different contexts, A3-E was able to maximize availability and prevent service interruptions while reducing the overall execution time and battery consumption. Finally, A3-E reduced deployment time by up to 70%, compared to a similar approach [32] for the resource management of edge nodes.

The rest of this article is organized as follows. Section 2 presents the continuum in terms of infrastructure and application models, formulates the life cycle management problem addressed by A3-E, and motivates the approach with a running example. Section 3 provides a detailed description of the A3-E model, whereas Section 4 introduces the prototype implementation of A3-E. Section 5 reports on the experiments performed to evaluate our proposal. Section 6 surveys related approaches, and Section 7 concludes the article.

2 THE CONTINUUM

2.1 Infrastructure Model

In our formulation, edge nodes are distinguished by the networking technology they use: *mobile edge* hosts (in the context of Multi-Access Edge Computing or MEC [9]) utilize cellular network infrastructures (e.g., 5G base stations), whereas *local edge* hosts integrate with local area network infrastructures (e.g., access points). Also, we assume that cloud and edge providers may be different from one another (e.g., Amazon AWS and telecom operators for MEC) and that the coordination among hosts (e.g., at different base stations) may not always be feasible.

In the continuum, mobile devices play two roles: they are both clients and providers of computational resources. The motivation for including the resources of mobile devices in the model is threefold: (1) the substantial increase in computational capacity exhibited by modern devices, (2) the compatibility of these devices with the computation of continuum microservices (see Section 2.2), and (3) the idea of giving applications zero network latency and highly available alternatives.

In this article, we refer to *cloud*, *edge*, and *mobile domains*. A domain yields a common abstraction of the heterogeneous resources that make up the continuum: e.g., servers, virtual machines, containers, CPU, memory, storage, and network components. According to MEC terminology, the term *domain* extends the ETSI-provided concept of *mobile edge host* [3, 9].

2.2 Application Model

We propose an application model in which stateless components and immutable data form *continuum services*, which are dynamically deployed to mobile, edge, and cloud resources; stateful components, which may still be needed in an application, are deployed to cloud data centers or to the client's device, depending on design-time decisions.

Continuum services can be defined as *microservices*[1] [18], as they are small and modular, communicate through lightweight mechanisms (often through an HTTP RESTful API), and are independently deployable by fully automated machinery. Moreover, continuum μ-services are aligned with the *Function-as-a-Service* (FaaS) execution model [29]. FaaS has been proposed as an alternative cloud paradigm in which business functionality[2] is provided without preallocating computational resources. Instead, shared resources (e.g., containers) are used to provision and execute functions on demand, in only a few milliseconds.

The proposed application model prevents data consistency problems (and the corresponding complexity of potential solutions) that would arise if stateful components such as databases were deployed to finely distributed edge nodes. It also allows multiple service instances to coexist along the continuum. Moreover, service instances may be deployed and undeployed independently without the need for state migration, favoring the seamless transition from one service provider to another. The latter is particularly important to cope with client mobility.

Figure 1 illustrates the architecture of a continuum application. The client-side application consists of *client-side logic*, *local persistence*, and *user interface* components, whereas the cloud infrastructure consists of *server-side logic* and *persistence* components. Continuum μ-services are responsible for stateless computations and are opportunistically deployed to different domains.

2.3 Life Cycle Management

When managing the life cycle of continuum services, there are two conflicting goals: (1) the satisfaction of application requirements and (2) the optimization of the resources consumed by these applications. In this article, we focus on three kinds of application requirements: service latency, battery consumption, and availability; simultaneously, we target the efficient and scalable usage of computational resources—namely, CPU, memory, and storage—from cloud and edge providers.

In the continuum, a multitude of disjoint cloud and edge domains can host the execution of μ-services; i.e., they must be able to handle operational aspects such as *downloading and installing* the μ-services. Also, mobile clients will freely enter and exit geographical areas that are covered by distinct edge domains; even cloud domains may see considerable variations to their aggregate

[1] Hereafter referred to as μ-service.

[2] Note that our approach targets *application-level functions*, whereas Virtualized Network Functions (VNFs [9]) are considered as part of the underlying infrastructure.

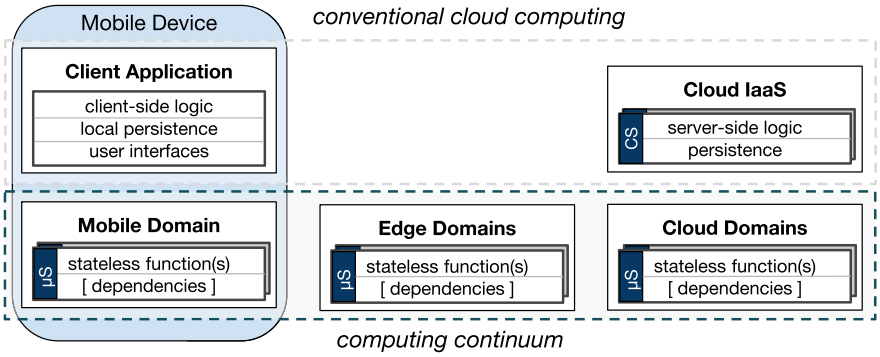


Fig. 1. The high-level architecture of a mobile application exploiting both the computing continuum, by means of μ-services (μS) provided by mobile, edge, and cloud domains, and conventional mobile/cloud computing, by means of local computation and cloud services (CSs).

demand over time. In such a scenario, it may be unfeasible to predict the origin and intensity of the demand that the different μ-services at each domain might see.

From the provider's perspective, an efficient and scalable allocation of the (virtualized) resources in each of its domains must be able to cope with the maximum accepted latency of each provided μ-service. To be efficient, the allocation of resources should be able to mimic the corresponding fluctuations of demand, i.e., be highly responsive. Consequently, it is essential for the mechanisms governing resource allocation in each domain to be *aware* of the actual, and potential, demand for each provided μ-service. To better express this problem, let us consider the set of μ-services $S = \{s_j \mid j = 1, 2, \ldots, J\}$. Vector $\bar{c} = (c_1, \ldots, c_J)$ holds the number of instances for each μ-service. Each instance is bound to a container; the resources allocated to each container (i.e., CPU, storage, and memory) are considered fixed and equal for all μ-services, coherently to the architecture of FaaS frameworks such as OpenWhisk [1]. Now let us assume that each provided μ-service $s_i \in S$ is bound to an SLA specifying its *maximum accepted service latency* Δ_j as well as the minimum/maximum number of service instances $Kmin_j/Kmax_j$. Given the fluctuations in the workload and the response time τ_j (comprising both set-up time and execution time) of each μ-service $s_j \in S$, the aim of a *domain manager* is to find the minimum c_j for each μ-service so that $\tau_j \leq \Delta_j \wedge K_{min_j} \leq c_j \leq K_{max_j}, \forall s_j \in S$.

From the client's viewpoint, the challenges for the materialization of the continuum are mainly twofold: (1) domains may need to be dynamically discovered, and (2) the client may need to choose, at runtime, among different domains that provide similar μ-services with different QoS. For a continuum application, an optimal domain selection is the one that, given a set of requirements (e.g., maximum response time, battery consumption) and the perceived QoS of different services, satisfies the multiattribute decision of which provider to use for each μ-service. Moreover, since continuum μ-services are modular and independently deployed, it consists of individual decisions regarding each μ-service consumed by the application.

To better express the client-side allocation problem, let us extend the previous formulation by considering a continuum application C_a that relies on the set of μ-services $S_a = \{s_{a,i} \mid i = 1, 2, \ldots, I_a\}$ and the set of disjoint continuum domains $D = \{d_p \mid p = 1, 2, \ldots, P\}$ perceived by the client. Each domain $d_p \in D$ provides a set of μ-services $S_p = \{s_{p,j} \mid j = 1, 2, \ldots, J_p\}$. For each $s_{a,i} \in S_a$, there is at least one domain providing that μ-service, that is, $\forall s_{a,i} \in S_a, \exists\ s_{p,j} \in \bigcup_{p=1}^{P} S_p \wedge s_{p,j} = s_{a,i}$. Now let us consider that each $s_{a,i} \in S_a$ is bound to a set of QoS requirements $QoS_a = \{q_{a,u} \mid u = 1, 2, \ldots, U_a\}$ and that each $q_{a,u} \in QoS_a$ is represented by a tuple $(ct_{a,u}, w_{a,u})$

respectively defining a *constraint* (e.g., response time $\leq$ 300ms) and a *weight* for that attribute, with $w_u \in 0 \leq w_u \leq 1 \wedge w_u \in \mathbb{R}$. For each $q_{a,u} \in QoS_a$, $actual_{p,a,u}$ defines the *value* for that QoS attribute as perceived by the client for domain dp. It follows that, for each $s_{a,i} \in S_a$, the goal is to select the domain $d_p \in D$ that maximizes the utility function $U_a(p) = \sum_{u=1}^{U_a} w_{a,u} * actual_{p,a,u}$, provided that $actual_{p,a,u} \vdash ct_{a,u}, \forall q_{a,u} \in QoS_a$, that is, that QoS constraints are satisfied.

2.4 Running Example

We illustrate the continuum with an example scenario that involves multiple applications that rely on computations executed in the cloud, the edge, and the user's device.

First, let us consider an Augmented Reality (AR) [6] application, employed by our user to explore the points of interest (POI) in the city she is currently visiting. This real-time application involves heavyweight image processing for the *extraction of features* from captured scenes, as well as a trained neural network model to *match features* from an extensive object catalog. This is a computation-intensive and latency-sensitive kind of application; as such, it can definitely benefit from adopting the continuum [7]. Indeed, despite the capacity of mobile devices, with these kinds of applications, it is common for users to experience functional and nonfunctional degradation (e.g., reduced object catalog, battery drain). By offloading the extraction and matching tasks to a server, the user can enjoy an improved Quality of Experience.

Two continuum μ-services are modeled for this application: one relies on an image processing library and a trained model, the other on a feature-based object catalog. Client-side logic is only responsible for capturing scenes from the device's camera and for updating the scene rendered by the display with POI information. To support this application, mobile-edge servers have been deployed to the base stations covering touristic areas. Their additional storage allows a broader set of objects to be recognized, thanks to a more extensive trained model. Variations in the workload are handled by means of a fast instantiation of AR μ-services, which must consider other applications that may be relying on the same servers.

After her tour, our user calls for an autonomous vehicle (AV) to drive her back to the hotel. During the drive back, she starts editing the pictures taken during the day; this activity is only interrupted by notifications from the AR application providing nearby POI information. To reduce the processing time and avoid battery drain, the vehicle features a local-edge server that provides a dynamic catalog of μ-services, including those needed by the AR application. Also, the vehicle uses a *route planning* μ-service to calculate the best plan to reach the destination. Since latency is not a first-class requirement, the latter is served by a cloud domain.

Once at her hotel room, our user continues to edit the images and videos that she shot during the day. Later on, she decides to enjoy a Mobile Game (MG). The MG consists of client-side logic and user interfaces, as well as complex calculations that pose a burden to the CPU of her device. To support guest applications, the hotel provides a local-edge server. The server identifies the image editing application and the MG and, after downloading and installing the necessary software, starts providing the required μ-services. Last but not least, a conventional cloud infrastructure provides stateful services (e.g., multimedia storage, authentication, and persistence).

3 A3-E

To realize the mobile-edge-cloud continuum, we propose A3-E, a model supporting the self-management of the life cycles of continuum μ-services. A3-E inherits its name from its four main activities, namely, *(**A**)wareness, (**A**)cquisition, (**A**)llocation, and (**E**)ngagement.*

A3-E targets the efficient and scalable placement of μ-services along the continuum and the satisfaction of application requirements such as maximum response time, battery consumption,

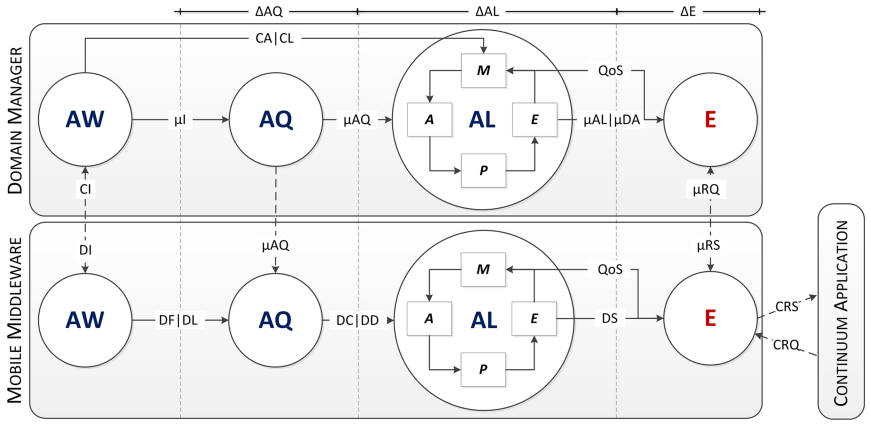


Fig. 2. A3-E overview. A3-E's activities—(AW), (AQ), (AL), (E)—are carried out by a *domain manager* and a *mobile middleware* and coordinate through asynchronous events depicted by arrows and labeled as follows: *domain identification* (DI), *client identification* (CI), *client arrived* (CA), *client left* (CL), *μ-service identified* (μI), *μ-service acquired* (μAQ), *μ-service allocated* (μAL), *μ-service deallocated* (μDA), *domain found* (DF), *domain lost* (DL), *domain confirmed* (DC), *domain denied* (DD), *domain selected* (DS), *μ-service request* (μRQ), *μ-service response* (μRS), *C-request* (CRQ), and *C-response* (CRS).

and availability. To achieve it, clients and heterogeneous domains take part in the automated and opportunistic decision of which continuum resources—among those of mobile, edge, and cloud domains—should be employed in provisioning each of the *μ*-services required by continuum applications. Figure 2 illustrates each activity in A3-E. Activities are refined by procedures that coordinate asynchronously through events and are carried out by a *domain manager* and by a *mobile middleware*. To address the intrinsic heterogeneity of the continuum, A3-E is flexible with respect to how each of its activities is actually implemented.

3.1 Awareness

The latency of continuum *μ*-services can be decomposed into *network latency* and *service latency*. The latter can be further decomposed into three parts (see Figure 2): *acquisition delay* (ΔAQ), *allocation delay* (ΔAL), and *execution delay* (ΔE). FaaS platforms like AWS Lambda [2] and OpenWhisk [1] adopt a *cold start* policy in which *μ*-services are allocated after the first call and kept for a while even if they are idle. The occasional ΔAL, however, may be disruptive for some applications.

Awareness has two benefits: (1) it alleviates ΔAL (cold start) by proactively allocating *μ*-services just before they are needed, and (2) it reduces ΔAQ and enables *μ*-service acquisition to be opportunistic (i.e., on demand) by triggering the download and installation of new *μ*-services as soon as the client becomes active in a specific domain (e.g., by starting the application or by entering the area of an edge domain).

While cloud domains are not likely to change and may be set up statically by clients, edge domains must advertise their existence (*domain identification*) with metadata (network address, static performance indicators) using protocols compatible with their network infrastructures: for example, through IP broadcasting in local-edge domains, and through Evolved Multimedia Broadcast/Multicast Service (eMBMS) in mobile-edge domains [3, 16]. In particular, edge domains exploit locality by triggering a *client left* event once mobile devices leave the coverage areas, while cloud domains rely on adjustable *timeouts*.

From the client's perspective, *Awareness* corresponds to the discovery of domains (source of *domain found* and *domain lost* events) and to the advertisement of required μ-services through a *client identification* event that contains their metadata (name, repository URL), upon which the domain manager triggers two *client arrived* and *μ-service identified* events. In this case, special-purpose HTTP endpoints can be used for cloud domains; local area network protocols (e.g., IP unicast over UDP) are used for local-edge domains; eMBMS protocols (e.g., FLUTE [16]), which are carried over traditional UDP and IP multicast toward end-user devices, are used for mobile-edge domains; and system-level events (e.g., intent broadcasting in Android) are used in mobile domains.

3.2 Acquisition

Acquisition models the automated download and installation of continuum μ-service artifacts and the confirmation that the domain can provide that μ-service. Its ultimate goal is to mitigate the use of domain resources before the μ-service is actually needed, while also facilitating IT operations (Ops) for developers and administrators. Ops mitigation is particularly important in (finely distributed) edge domains, since the manual administration of a large number of μ-services can prove cumbersome and expensive. Nevertheless, this can also prove useful for cloud domains. Indeed, to the best of our knowledge, current FaaS platforms only support uploading (pushing) functions through public interfaces.

Acquisition is autonomously managed; therefore, it allows μ-services to be downloaded and installed on demand. A domain manager fetches the artifacts (e.g., compiled classes and dependencies) from a repository upon the arrival of a *μ-service identified* event. Note that mobile domains are exempt from performing *Acquisition* as local μ-services are assumed to be downloaded and installed along with the client application.

From the client's perspective, *Acquisition* corresponds to the confirmation (or denial) of the capability of a domain in providing the μ-services required by the application. After a *domain found* event, the mobile middleware listens for a *μ-service acquired* event—to be handled together with the update of a list of capable domains and the subsequent triggering of a *domain confirmed* (or *denied*) event.

In our running example, the assets that compose the μ-services of the AR, Image Editing, and MG applications are once fetched and installed by the two local-edge domains upon the arrival of a first *μ-service identified* event. While this is achieved, the applications momentarily continue to rely on their mobile domain, or on any other domain in which the μ-services have already been acquired and allocated due to a previous interaction (e.g., the mobile-edge domains in our example skip the acquisition of AR artifacts due to previous contacts with tourist devices).

3.3 Allocation

Allocation models the deployment of continuum μ-services on a pool of resources provided by the domain. It also captures the client-side selection of the domain(s) for each of the μ-services employed by the application.

The scope of provider-side *Allocation* is limited by its domain boundaries. Cloud domains allocate μ-service instances to containers in resourceful data centers covering a large area. On the other hand, edge domains rely on containers from one or more (virtual) machines serving an office or a building (local-edge) or a 5G base station area (mobile-edge). Finally, the allocation of μ-services to mobile domains is platform-specific (e.g., based on the Android service life cycle).

Existing FaaS platforms handle *Allocation* with the on-demand instantiation of containers after a first request, which may be kept *warm* before being deallocated after a period of idleness [1, 2]. A3-E generalizes this mechanism as a self-management loop [15] in which μ-service instances are

allocated to cloud and edge domains to guarantee that the latency and availability of each provided μ-service are in line with the desired SLA. To achieve this goal, the self-management *monitor* (M in Figure 2) measures the number of μ-service invocations and their execution times (*QoS* in Figure 2) and detects *client arrived* and *client left* events from *Awareness* (*CA* and *CL*, respectively, in Figure 2). The *analyzer* (A in Figure 2) aggregates monitored data over a predefined time window and computes the *arrival rate* (α_j), *response time* (τ_j), and *number of clients* for each μ-service.

The *planner* (P in Figure 2) exploits these analyses and calculates the number of instances c_j so that $\tau_j \leq \Delta_j \wedge K_{min_j} \leq c_j \leq K_{max_j}, \forall s_j \in S$. To mitigate ΔAL (cold start), the planner reacts to a *client arrived* event by anticipating the allocation of containers, if resources are available, and by keeping them allocated (*warm*) after each *μ-service request*. Given the resource limitations of edge domains, contentions between different μ-services may exist. Thus, the *planner* is also in charge of managing such situations by prioritizing applications. Back to our running example, critical applications, like the AR application for tourists, should have a higher priority with respect to the others. In such cases, some μ-services might become unavailable or available with a slower response time. Finally, the *executor* (E in Figure 2) carries out the new allocation scheme by means of commands to the container platform (e.g., Docker).

The fluctuations in the availability and latency are handled by clients by means of the dynamic selection of the domain that best satisfies their requirements. Analogously to the provider-side *Allocation*, the mobile middleware realizes this activity by means of a self-management loop [15] for each μ-service consumed by the client application. At each loop iteration, the *monitor* (M in Figure 2) measures the *battery level* and *network latency* of all capable domains $D_j \subset D$ and listens to *QoS* events from *Engagement* to keep track of the actual *execution times*. The *analyzer* (A in Figure 2) aggregates acquired data to compute the overall *battery consumption* and *service latency* of each *capable domain*. The *analyzer* may also consider the static performance indicated by each domain during *Awareness* (e.g., the computational power of the distinct edge and cloud domains). The *planner* (P in Figure 2) uses these data to run a multiattribute rating algorithm [22] to compute the best domain for future invocations. Finally, the *executor* (E in Figure 2) enacts the selection if needed (i.e., the computed domain differs from the one in use) and eventually triggers a *domain selected* event. Meanwhile, the middleware handles *domain acquired (lost)* events by including (disregarding) such a domain. A detailed description of this self-management loop can be found in Section 4.

3.4 Engagement

Engagement models the actual provisioning of a continuum μ-service by a domain after its successful acquisition and allocation. Throughout *Engagement*, and as long as the client-domain interaction persists, the client is able to engage with that domain by means of invocations to provided μ-services. Remote domains (i.e., cloud and edge) are engaged through distributed protocols (e.g., HTTP requests or WebSockets). To enforce a standard interface between the mobile middleware and heterogeneous domains, the mobile domain is engaged by means of system-level events.

During *Engagement*, the domain manager handles *μ-service request* events by associating them with μ-service instances, given the constraints set by the used *load balancing* strategy. The decisions taken during *Allocation* guide the manager to react to *μ-service deallocated* events by indicating that the μ-services have become unavailable and by queuing subsequent requests to them. In case of too high latency, the manager sends a *μ-service response* event along with a specific error code. Conversely, a *μ-service allocated* event indicates the recovery of a given μ-service; queued and subsequent requests are then processed accordingly.

From the client's viewpoint, a *C-request arrived* indicates the client application has sent a new request to a continuum μ-service. The event contains the name of the target μ-service along with

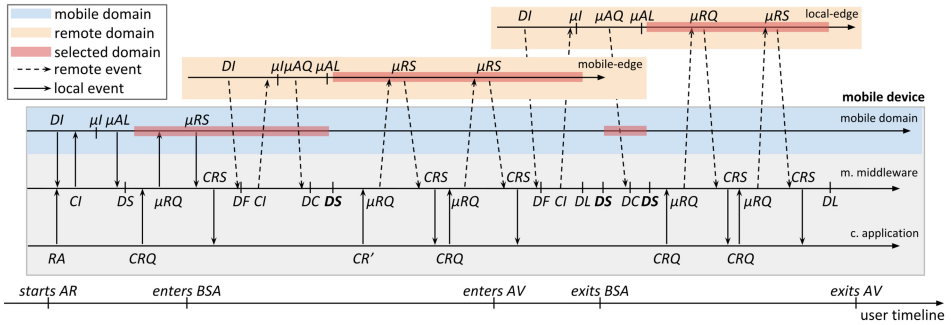


Fig. 3. A timeline of events from the running example scenario.

required parameters; the middleware handles it by invoking (*μ-service request*) the μ-service from the currently selected domain. Upon arrival of a *μ-service response*, the middleware triggers a *C-request reply* event, which is then handled by the client application. During *Engagement*, the mobile middleware listens for *domain selected* events that indicate the domain to interact. If no domain is available for that specific μ-service, the middleware reacts by queuing subsequent C-requests until a new *domain selected* event confirms a new domain or, in case of timeout, by triggering a *C-request reply* with an error.

Figure 3 depicts a timeline of events from our Running Example scenario. The timeline starts with our user initializing the AR application after entering the touristic area. The mobile middleware receives a *domain identification* event (*DI*) from its mobile domain and replies with a *client identification* (*CI*). Following the μ-service identification (*μI*), the middleware triggers a *μ-service allocated* (*μAL*) once the corresponding functions have been registered. After selecting this domain, the middleware triggers a *domain selected* (*DS*), which allows subsequent *C-request* (*CRQ*) events to be handled locally. As our user enters the area covered by a mobile-edge domain provided by a base station (BSA), the middleware receives a *domain identification* (*DI*) and repeats the previous handshake procedure. To prevent battery drain, the self-management loop decides for the mobile-edge domain and triggers a *DC* event once the *μAQ* event arrives. After a long period of engagement with the mobile-edge domain, our user enters the vehicle and its local-edge domain. Due to a change of network, the connection with the mobile-edge domain is lost (*DL*). To prevent service interruption, the middleware momentarily switches back to its mobile domain (*DS*). Meanwhile, as this is the first contact with the AR application, the edge domain goes through *Acquisition*, which takes some time (Δ*AQ*) to complete (*μAQ*). Upon a *domain confirmed* event (*DC*), the local-edge domain is selected (*DS*).

4 IMPLEMENTATION

To demonstrate the capabilities of A3-E in managing the life cycle of continuum applications, we have implemented prototype versions of domain managers, for local-edge and mobile domains, and of the mobile middleware, for Android devices. These prototypes have then been employed in the evaluation of our model. In this article, we rely on existing FaaS platforms [1, 2] for handling the *Allocation* of μ-services to a pool of dynamically allocated containers.

4.1 Domain Managers

The first domain manager[3] manages a local-edge domain and encompasses *Awareness* and *Acquisition*; *Allocation* and *Engagement* are delegated to the FaaS platform (OpenWhisk), which allocates

[3]Documentation and source code available at https://github.com/deib-polimi/A3-E-DSM-local-edge/.

μ-services to its pool of containers and handles client application requests (fired by the middleware) by means of RESTful endpoints.

The domain manager implements *Awareness* by broadcasting *domain identification* UDP events at a constant interval. A client device that enters the network replies—by means of a UDP unicast—with the *client identification* event that contains the μ-services the application requires, along with the respective repository from which μ-service artifacts can be fetched during *Acquisition* (as described in Section 3.1). For each identified μ-service, the manager proceeds with *Acquisition*, and among downloaded files, a descriptor provides installation instructions (e.g., compilation of Java classes and required dependencies). In particular, the prototype relies on Gradle,[4] a state-of-the-art build tool commonly employed in projects ranging from mobile applications to μ-services.

Once artifacts have been downloaded and built, *Acquisition* finishes with the deployment of μ-service function(s) and dependencies to OpenWhisk by means of its command line interface and with the generation of a *μ-service acquired* event with the same UDP unicast channel. In case of failure, the mobile middleware is informed with a *μ-service denied* event (as described in Section 3.2).

The second domain manager considers Android devices as particular mobile domains, and the resulting implementation was packaged as a module within the mobile middleware (described in Section 4.2). The prototype implements *Awareness* by triggering a system-level *domain identification* event once it has been loaded by the middleware and by listening to a *client identification* reply event. In contrast to cloud and edge domains, each *μ-service identified* event contains the qualified name (e.g., the system path of a Java class that implements the static function), which is added to a *service registry* that implements *Acquisition*. This domain manager supports two types of μ-service functions: Java methods, which are natively supported, and JavaScript functions, which require a JNI wrapper for their execution on Android devices. Note that existing FaaS platforms support a variety of other languages and runtimes. More comprehensive implementations of the mobile domain manager may either use additional wrappers or require developers to provide native implementations of needed μ-service functions.

Once a μ-service function is registered, the mobile domain manager triggers a *μ-service acquired* event, which enables its selection by the middleware. During *Engagement* and upon the selection of this domain, *μ-service request* events are handled by looking up for the corresponding functions. Once found, the function is called with the parameters in the original *C-request* and produces a *μ-service response* to return the result.

4.2 Mobile Middleware

This middleware[5] targets Android devices, but it does not use any Android-specific feature and can be generalized to other mobile platforms.

To implement *Awareness*, the middleware listens for *domain identification* events triggered by its mobile domain and broadcasted by edge domains through UDP. To avoid battery drain, the middleware limits discovery to a short time period after the mobile middleware is first launched or the mobile device changes network (e.g., from a local area WiFi to a 5G cellular network). For every domain found, the middleware proceeds by sending a *client identification* event containing the address of the repository from which μ-service functions and dependencies can be downloaded, for remote domains, or the qualified name of Java/JavaScript classes, for its local (mobile) domain. Each corresponding *μ-service acquired (denied)* event is handled by means of a system-level *domain confirmed (denied)* event. Since cloud domains have been evaluated in this article by exploiting an

[4]https://gradle.org/.

[5]Documentation and source code available at https://github.com/deib-polimi/A3-E-CSM.

existing FaaS platform, which lacks *Awareness* and *Acquisition*, these domains have been set up programmatically at startup.

The middleware also implements the self-managing loop required by *Allocation* (see Section 3.3). The loop takes into account three types of requirements: *location* constrains the μ-service to be *local*, on a *local edge*, on a *mobile edge*, or in the *cloud* (or any combination of the above), while service *latency* and *battery consumption* can be set to *any*, *low*, or *very low*. For each μ-service, the middleware monitors network latency for each of the corresponding capable domains; it then adds the execution time to the aggregated network latency and plans for the changes in the selected domain by means of a multiattribute rating algorithm; finally, it enacts the change by triggering a *domain selected* event.

To estimate the execution time for different domains, the middleware relies on scores that range from 1 to 5 and defines the computational power of the domains.[6] By default, the score of mobile domains is 1 and the score of cloud domains is 5 to reflect their shortage/abundance of computational resources. In particular, the score of edge domains is determined by the static performance indicator part of their *domain identification*s (see Section 3.1). Battery consumption follows a similar, dual path: 5 for mobile domains, while 1 and 3 are the default values used to characterize edge and cloud domains, respectively. These values can be updated at runtime based on the service latency of each domain: the longer the connection stays open waiting for a response, the more battery is consumed.

ALGORITHM 1: A3-E Selection Algorithm

```
1:  function selectDomain(A3EService service, A3EDomain[] capableDomains)
2:      scoreRange ← 5
3:      maxLatency ← computeMaximumLatency(capableDomains)
4:      maxBattery ← computeMaximumBattery(capableDomains)
5:      latencyWeight ← service.getLatencyRequirement()
6:      batteryWeight ← service.getBatteryRequirement()
7:      maxScore ← 0
8:      selectedDomain ← null
9:      for all domain ∈ capableDomains do
10:         serviceLatency ← computeServiceLatency(domain.getNetworkLatency(), domain.
             getCptPower())
11:         batteryConsumption ← computeBatteryConsumption(domain, service)
12:         latencyScore ← latencyWeight ∗ ((scoreRange − 1) ∗ (1 − latency/maxLatency) + 1)
13:         batteryScore ← batteryWeight ∗ ((scoreRange − 1) ∗ (1 − batteryConsumption/
             maxBattery) + 1)
14:         score ← (latencyScore + batteryScore)/(latencyWeight + batteryWeight)
15:         if score ≥ maxScore then
16:             maxScore ← score
17:             selectedDomain ← domain
18:         end if
19:     end for
20:     return selectedDomain
21: end function
```

Algorithm 1 describes the procedure used by the *planner*. The algorithm computes a score that ranges from 0 to 5 (line 2). First, it retrieves the maximum network latency and computational

[6]Labeling computational power is also common in the cloud where different tiers of virtual machines are available: e.g., https://aws.amazon.com/ec2/instance-types/.

power from available domains (lines 3 and 4). Then, it retrieves the weights assigned to each QoS metric (lines 5 and 6), which correspond to the values required for service *latency* and *battery consumption*: *any* corresponds to a weight of 0, *low* corresponds to 1, and *very low* to 2.

For each domain, the algorithm computes the overall score (lines 9 to 14). For each QoS attribute, it normalizes the actual value with the previously computed maximum. Since a higher score should mean lower service latency/battery consumption, the algorithm computes each score by means of the complement of the normalized value and adds 1 to avoid 0 scores. The overall score is then calculated as the weighted average between the scores obtained by the domains for service latency and battery consumption. The loop concludes (lines 15 to 18) with the selection of that domain if its overall score is greater than the previous (maximum) one. In particular, Algorithm 1 is an instantiation of SMART [22], in which multiple competing QoS attributes are taken into account using the following formula:

$$Smart(p) = \frac{\sum_{u=1}^{U} value_u(p) * weight_u}{\sum_{u=1}^{U} weight_u}, \tag{1}$$

where p is a domain, the considered QoS attributes are service latency and battery consumption (thus $U = 2$), and their weights are represented as explained above.

To implement *Engagement*, the middleware handles *C-request*s triggered by client applications by invoking the services provided by the previously selected domain. Domains are bound to invocation resolvers: those for edge and cloud domains fire HTTP requests, while those for mobile domains broadcast Android events that contain the requests. Analogously, each resolver handles invocation responses by triggering the corresponding *C-response* events, which are then handled by client applications.

5 EXPERIMENTAL EVALUATION

To assess the proposed computing continuum, we performed different experiments with four distinct domains. The first experiment studies how latency changes and how the remote domains scale with a varying workload. The second experiment evaluates all domains from a client's perspective, in terms of both battery consumption and execution time. The third experiment evaluates the capabilities of selecting domains dynamically and targets availability. Finally, the last experiment evaluates the performance of *Acquisition* and *Allocation*. When possible, we compared A3-E against Enorm, a framework for edge resource management.

5.1 Setup of Experiments

Table 1 summarizes the four domains used in the evaluation. The Mobile domain was deployed on an Android smartphone that hosted both the A3-E middleware and a continuum, Augmented Reality application that invokes μ-services[7] responsible for *feature extraction* and *object detection*, which are placed along the continuum. The application is a typical use case for the continuum due to its demanding requirements in terms of low latency and high computational power [6, 7].

The prototype domain manager (see Section 4.1) was deployed on two local-edge domains. *Local-edge-1* represented a situation in which latency is ultra low but the computational resources are more constrained and scaling up is not possible due to the inherent physical restrictions of the underlying infrastructure (e.g., a lightweight, office-wide server). In turn, *Local-edge-2* had more computational resources and, again, low latency could be achieved due to physical proximity (e.g., a robust edge server that is supposed to cover an entire floor of a building).

[7]Implementation available at https://github.com/deib-polimi/A3-E-image-recognition.

Table 1. Domains Setup in the Continuum for the Experimental Evaluation

Domain	Machine Resources	Execution Environment
Mobile	Samsung Galaxy S6 SM-G90, 3Gb RAM, 8x Cortex CPU 2Ghz	Android 5.0.2 + Java Functions + OpenCV
Local-edge-1	ubuntu/trusty64-2, 4x vCPUs, 4Gb RAM	OpenWhisk, 256Mb/Action, Python 2.7 + OpenCV
Local-edge-2	ubuntu/trusty64-2, 8x vCPUs, 16Gb RAM	OpenWhisk, 256Mb/Action, Python 2.7 + OpenCV
Cloud-FaaS	N/A	AWS Lambda, 256Mb/Function, Python 2.7 + OpenCV
Cloud-IaaS	Auto Scaling Group with t2.micro instances + Amazon Linux AMI 2017	NodeJs 6.11 server + Python 2.7 + OpenCV

As for cloud domains, we used AWS Lambda [2] (*Cloud-FaaS*), the most mature FaaS solution on the market. To be consistent with our formulation of the computing continuum, functions and associated dependencies were deployed to the AWS Lambda data center in Europe. Additionally, we also deployed the functionality onto a traditional setup using the IaaS provided by AWS (*Cloud-IaaS*). The main goal of this setup was not to compare traditional cloud services against an FaaS solution, but to demonstrate that the proposed continuum could outperform the cloud under the tested circumstances and requirements.

5.2 Service Latency and Scalability

The first experiment assessed different domains in terms of service latency and scalability when subjected to a varying workload. We simulated an increasing number of clients, each one making 100 requests for the μ-services required by the application at a rate of two per second. This setup was conceived by taking into account the default limit of concurrent executions in AWS Lambda [2] and Openwhisk [1].

This experiment excluded the mobile domain and focused on the remote domains, that is, the edge- and cloud-based ones. In this experiment, requests were emulated using Postman,[8] an open-source application designed to perform load testing. The payload used for this experiment was an example image of approximately 65KB, which is a reasonable size for this use case when considering the requirements related to low latency and computation time. To mitigate the cost of cloud-based providers, the execution time was profiled once for each domain, and then we focused on network latency, which is subject to higher fluctuations; the results on network latency were averaged through five executions.

Figure 4 shows the average latency for each increment in used clients. Note that the computation time (light gray) is different from the overhead (dark gray). The latter includes network communication (routing and forwarding) and queuing time (when no resources are available to process the request). If we use *Cloud-FaaS* as baseline, latency reduction was up to 90% for *Local-edge-1* and up to 82% for *Local-edge-2*. Interestingly, with *Cloud-FaaS,* the latency decreases when the number of simultaneous clients increases. This is due to the extra infrastructure provided under the hood by AWS Lambda to compensate for the initialization overhead of cold requests: higher reuse rates correspond to higher stress levels of requests [20]. For a few service invocations, the load distribution adopted by AWS is uneven across hosts. This uneven use of the infrastructure may

[8]https://www.getpostman.com/.

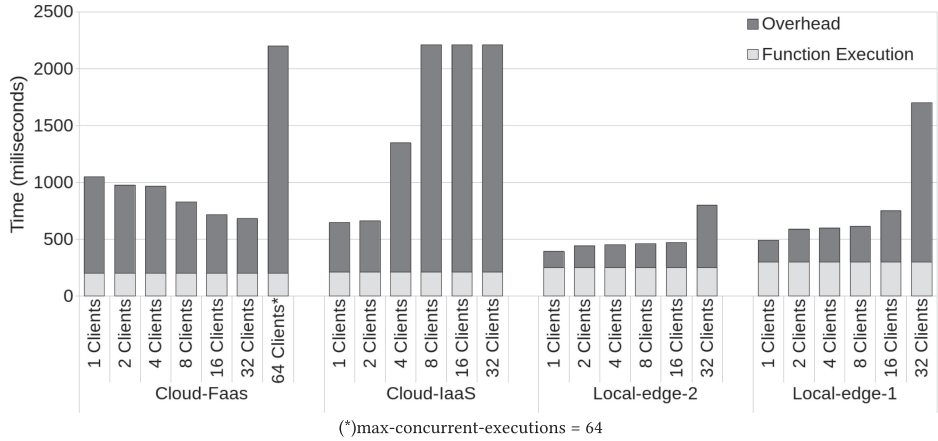


Fig. 4. Latency and scalability for each domain and different number of clients.

lead to the early deallocation of some containers (incurring in cold starts) if client workloads do not utilize all hosts. Thus, the decrease in the execution time with a higher level of workload is justified by the fact that more containers are kept warm.

Despite its virtually unlimited resources, the *Cloud-FaaS* has tunable limits for the maximum number of concurrent executions (default is 1,000) due to budget constraints [31]. This is reflected in the substantial increase in latency when we consider the *Cloud-FaaS* domain and 64 clients.

If we use the *Cloud-IaaS* domain as baseline, the reductions when using *Cloud-FaaS* are up to 77% and 58% with respect to *Local-edge-1* and *Local-edge-2*, respectively. Interestingly, *Cloud-IaaS* outperformed *Cloud-FaaS* (46% less overhead) for light workloads (up to two simultaneous clients). This can be due to the additional steps performed by the API Gateway to forward RESTful calls to AWS lambda functions in *Cloud-FaaS*.[9] Nevertheless, this advantage is mitigated by the fact that *Cloud-FaaS* can better react to workload bursts, given its faster horizontal scaling [11, 31].

For light to medium workloads (up to 16 simultaneous clients), the overhead added by the local-edge domains is less than in both cloud alternatives. Under heavier workloads (from 32 simultaneous clients onward), these domains present restricted availability and degraded performance—as one can observe with *Local-edge-1*, the most resource-constrained domain. One may argue that a local-edge domain is intended to cope with light workloads (e.g., a single office or a floor in a building). When edge domains must cope with hundreds of simultaneous clients (e.g., a mobile-edge domain), the computation, storage, and memory capabilities are expected to be orders of magnitude higher.

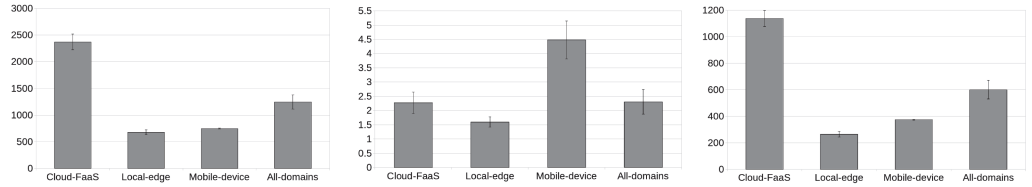
5.3 Battery Consumption and Execution Time

The main goal of this experiment was to evaluate the compute continuum from the client's viewpoint in terms of battery consumption and execution time. Differently from the previous experiment, we set up a mobile device with the mobile middleware (see Section 4.2).

This experiment featured four different scenarios: the first three consider one domain each (*Cloud-FaaS*, *Local-edge-2*, and *Mobile-device*), and the last one (*all-domains*) combines the previous ones to form the continuum.[10] The experiment consisted of cascading 2,000 sequential requests

[9]http://docs.aws.amazon.com/lambda/latest/dg/with-on-demand-https.html.

[10]Details on the availability of each domain in the *all-domains* scenario are discussed in Section 5.4.



(a) Total execution time (sec) (b) Battery consumption (%) (c) Execution time per call (ms)

Fig. 5. A3-E experimental evaluation results.

for *feature extraction* and *matching* of an example image (with a size of 65KB). We measured the total execution time, that is, the time between a request and its response, the battery consumption, and the average time per call. Figure 5 shows the obtained results, averaged on five executions for each scenario.

If we consider the total execution time (Figure 5(a)) and *Cloud-FaaS* domain as baseline, *Local-edge* reduced it up to 72%, while *Mobile-device* and *All-domains* reduced it up to 69% and 49%, respectively. In turn, we measured the battery consumption (Figure 5(b)) in the *Mobile-device* domain and noticed a drop of 4.5% after 750 seconds (i.e., 12.5 minutes) of execution. Starting from this baseline, the savings with *Cloud-FaaS*, *Local-edge,* and *All-domains* were 49%, 35%, and 49%, respectively. The time per call (Figure 5(c)), with *Cloud-FaaS* as baseline (1, 137 milliseconds per call), was improved by 76%, 68% and 47% for *Local-edge*, *Mobile-device,* and *All-domains*, respectively.

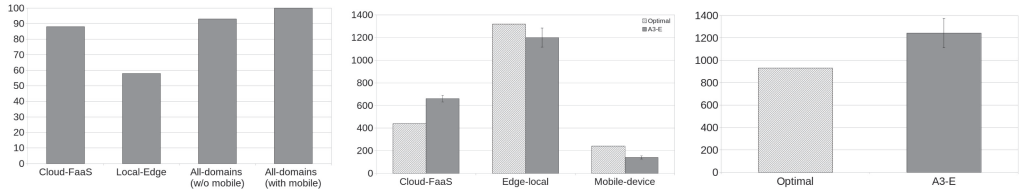
These experiments tell us that the total execution time, when using only the cloud, was two times higher than when using the continuum (*All-domains*). Since the requests were performed in cascade and given the higher latency per call in the cloud, the total time increases accordingly. Clearly, the use of A3-E to switch to edge domains when possible would substantially reduce latency and would improve the perceived QoS.

Battery consumption was substantially lower when offloading computation rather than performing it on the device. The *Mobile-device* domain lasted half the time but used twice as much battery (a prohibitive 20% of battery drain per hour) than the *All-domains* one, given that it was performing CPU-intensive operations. This recalls the importance of computation offloading to preserve the resources of mobile devices.

5.4 Domain Selection and Availability

We evaluated the capability of A3-E to select the best domain given the requirements on service latency and battery consumption. In this experiment, we used the three aforementioned domains together to form the continuum and simulated their availabilities using a probability distribution. The mobile middleware was configured to ping for domain availability and network latency every 2 seconds. We considered that the cloud domain could be unavailable mainly due to the absence of mobile network coverage since downtimes of cloud services are minimal [10]. To simulate this, the average network unavailability was set to once every 15 minutes, while the average time for it to become available again was 2 minutes, which resulted in an availability of 88%.

The rationale for the edge domain was analogous, yet with a higher probability of being unavailable (e.g., due to the lack of memory or CPU). In this case, the edge domain was unavailable once every 10 minutes, and it needed an average of 5 minutes to become available again, resulting in an availability of 66%. If we consider that edge nodes are only reachable within network coverage, the resulting availability is calculated by the product of the two availabilities, that is, $0.88 * 0.66 = 58\%$. Finally, the mobile device is considered as always available, as we aimed to stress the tradeoff



(a) Average availability (%) (b) Number of requests served (c) Total Execution Time (sec)

Fig. 6. All-domains scenario results.

between battery consumption (when the mobile domain is employed) and latency (when remote domains are used).

To set an *optimal* baseline for this experiment, we calculated the theoretical number of calls to be served per domain, given the probabilities explained above. Upon this, we calculated the *optimal* execution by assuming no overhead for domain switching and by always using the best domain available. Figure 6 shows the results for the *All-domains* scenario, with respect to availability. The experiment showed (see Figure 6(a)) an average perceived availability of 93% without considering the *Mobile-device* domain (5% and 35% improvement with respect to the cloud and edge domains, respectively) and 100% availability when also considering the *Mobile-device* domain (which is always available).

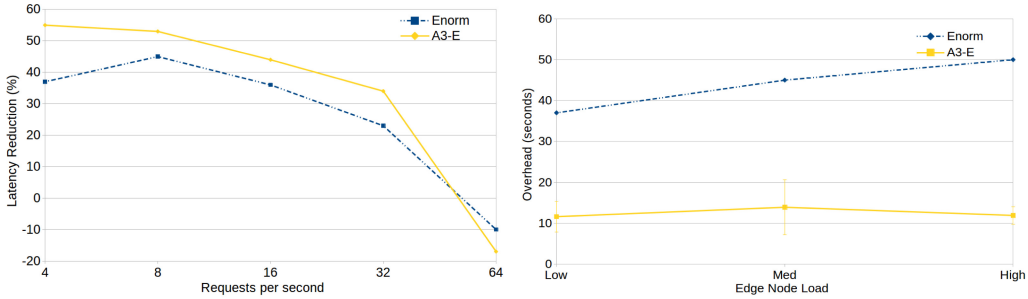
Figure 6(b) shows the distribution of calls per domain in the continuum: 63% served by *Local-edge*, followed by *Cloud-FaaS* (30%) and *Mobile-device* (7%)—with an optimal of 66%, 22%, and 12% respectively. In terms of consistency, given that all requests were served, on average 70% of the requests perceived low latency, while the 30% that relied on the cloud had a degradation in QoS but still were processed successfully. Finally, Figure 6(c) shows an increase of 33% in the total execution time using A3-E w.r.t. the optimal, where the former includes the overhead of domain selection and switching.

The experiment showed that A3-E is capable of performing domain selection and computation distribution at runtime. We achieved 100% availability, given that the mobile domain is always available. The *All-domains* scenario reflects the underlying rationale of the computing continuum: one should exploit edge domains as much as possible (by giving them high scores in Formula 1), leading to a better balance among computation time, latency, and resource consumption. Note that the computational power of the mobile domain has the lowest possible score (see Section 4.2), and thus the cloud was often selected as the first alternative in case of unavailability of the edge domain.

5.5 Enorm

Finally, we compared *Local-edge-2* against Enorm [32], which assumes a similar, resource-constrained configuration for edge nodes. Figure 7 (a) shows the latency reduction when using both A3-E and Enorm, that is, an edge alternative, instead of a cloud-based solution. Both edge-based solutions are better than a cloud alternative for up to 50 simultaneous clients (from 35% to 55% latency reduction), and A3-E is always better (higher reduction) than Enorm. Both approaches perform worse than the cloud solution with heavier workloads (from some 50 to 64 simultaneous clients on).

The last experiment evaluated the performance of *Acquisition* and *Allocation*. Given the resource limitations of edge domains and the potential benefits of exploiting the client arrival/exit awareness provided by edge locality, we targeted the evaluation of *Local-edge-2* as a domain. The



(a) Latency Reduction (%) (b) Provisioning Overhead (sec)

Fig. 7. Comparison of A3-E (*Local-edge-2*) and Enorm against a cloud-based solution.

experiment focused on the scenario in which no edge resources are preallocated, that is, a worst-case cold-start scenario ($\Delta AQ + \Delta AL$).

The μ-service we used was similar to the ones used in the previous experiments. Along with the metadata related to *client identification*, the client informed the server of the repository to fetch required assets (some 30Mb including the μ-service function and dependencies). Also, the experiment only measured the domain-side performance, as the service latency perceived by clients was evaluated in the previous experiments. After each successful *Acquisition* and *Allocation*, we uninstalled and deleted all the μ-service-related assets to allow subsequent measurements to capture a worst-case cold start.

We executed cascading sequential requests for periods of 5 minutes, with different utilization levels of the edge node: low (10% server load and low network traffic, equivalent to eight clients), medium (55% server load and 16 clients), and high (85% server load and 32 clients). Again, we were able to compare A3-E against the Enorm framework, given that our acquisition and awareness are analogous to the provisioning phase in Enorm, which consists of deploying application server partitions from the cloud to containers' edge nodes.

Figure 7(b) shows obtained results. The average time between the detection of a *client identification* event and a successful *Allocation* was 12.5 and 44 seconds for A3-E and Enorm, respectively, without considerable variations regarding the current load of the edge node. Such a reduction of provisioning overhead (up to 70%) is one of the main advantages of adopting an FaaS model for the continuum. The underlying FaaS solution (OpenWhisk in this experiment) reduces the burden of downloading and installing new functionality: thanks to the highly shared platform, μ-service functions can be created and deleted in a fraction of the time needed to do it with complete containers (as in Enorm and most of the state-of-the-art solutions).

5.6 Threats to Validity

The experiments only targeted one example application and, in the case of the experiments of Sections 5.3 and 5.4, one single client device. Further tests are needed to also consider multiple clients that use several applications composed of μ-services with conflicting requirements, whose corresponding functions are deployed along the continuum. Additionally, it is currently not possible to test with real mobile-edge domains, that is, to provide computational capabilities on base stations. This could be approximated either by simulation or by deploying edge domains by following the current specifications of *mobile edge computing* in terms of computational power and latency, to better capture the heterogeneity of the continuum. Nonetheless, the fact that specifications and

technologies are still under development limits the accuracy with which mobile-edge domains can be evaluated.

6 RELATED WORK

The first notion of a computing continuum was based on cloudlets (trusted, resource-rich computers, or clusters of computers, connected to the Internet and available for nearby mobile devices [24]) for a mobile-cloudlet-cloud architecture that aimed to reduce response time [28]. Cloudlets are homogeneous and connected to coordinate the offloading of computationally intensive tasks. Static data (e.g., template images for face recognition) are stored in the cloudlets beforehand.

A number of works have tackled the problem of placement among the set of computational entities from edge to cloud computing. Some (e.g., [30]) focus on the placement problem of single components/applications, whereas others (e.g., [34, 36]) consider multiple components/applications: the problem in both cases proved to be NP-hard [36].

With respect to these works, our formulation allows multiple instances of (static and lightweight) μ-services to coexist across the computing continuum and, from the viewpoint of cloud and edge providers, A3-E focuses on the opportunistic allocation of instances within disjoint *domains*. On one hand, domain autonomy copes with scenarios in which cloud and edge resources are managed by distinct providers and interdomain coordination is not feasible. On the other hand, the cooperation among nearby edge nodes (as part of an extended domain) may further increase the robustness and scalability of the edge layer and is considered for future work.

More closely related to our service model, Jia et al. [14] tackled the QoS-aware offloading of tasks to distributed cloudlets with Virtual Network Functions (NVFs), while [33] proposed an online approximation algorithm for the placement of service instances. In our model, we rely on state-of-the-art load balancing mechanisms to distribute requests to existing instances of services, which are opportunistically allocated to autonomous domains in collaboration with clients.

From a different perspective, several works [19, 23, 35, 37] have focused on the decision of whether to offload computation from/to mobile, edge, or cloud nodes. In this category, Orsini et al. [23] proposed CloudAware, a comprehensive context-aware mobile middleware that handles offloading to edge and cloud nodes. Similarly to A3-E, clients select the alternative that best suits their requirements, including the option of performing computations locally. CloudAware deals with stateful components, which renders the offloading decision—controlled by the client—more critical. Instead, A3-E influences, but does not dictate, the opportunistic allocation of stateless and lightweight μ-services onto edge and cloud providers.

The Enorm framework [32] for edge node resource management exhibits similarities with A3-E. In Enorm, cloud managers are responsible for the allocation of server-side application partitions on edge nodes with the aim of reducing service latency and the volume of data sent to the cloud. Server allocation follows a handshake between cloud and edge managers. If successful, servers are deployed, and clients are bound to specific ports following a network-level reconfiguration. In contrast, in A3-E, μ-services are dynamically deployed after a direct handshake between clients and autonomous domains, including mobile, edge, and cloud domains. Also, clients decide on the best domain given their requirements and perceived QoS. As in Enorm, our model takes into account both priority and latency; nonetheless, in A3-E, requests from different clients are not bound to specific instances but decided by a load balancer, favoring seamless mobility. Accordingly, allocation in A3-E can be based on a controller able to deal with a high workload churn and a large number of concurrent requests.

As for the Network Function Virtualization (NFV) field, several frameworks, mainly based on the ETSI MANO standard [13], have been proposed to cope with the fluctuating demand of

network infrastructure[11] [26]. These frameworks provide an abstraction layer over a mobile edge infrastructure, making the shift among the different parts of the continuum utterly transparent to applications. However, A3-E focuses on the opportunistic placement of Application-Level Functions (following the FaaS model) rather than on VNF elements. We see NFV/VNF as part of the underlying infrastructure, and thus the work on them complementary to ours.

7 CONCLUSIONS AND FUTURE WORK

This article proposes A3-E, a unified model for creating and managing the computing continuum formed by mobile, edge, and cloud resources. Due to its *self-management* capabilities and *efficiency*, which complement the functionality provided by FaaS platforms, A3-E provides a suitable approach for the realization of the convergence among mobile, edge, and cloud nodes. It overcomes the heterogeneity of the different domains thanks to stateless functions exposed as μ-services and tackles the opportunistic placement of computation along the continuum through mutual client-provider awareness. A3-E has been assessed through experiments that demonstrated its ability to support applications with the opportunistic placement and selection of μ-services. The experiments also showed a substantial reduction of both latency and battery drain when edge services are used instead of cloud or local ones, while availability was maximized by the mobile domain.

Ongoing and future work include advanced resource management along the continuum. Several approaches in the literature tackle the dynamic resource allocation problem by using heuristics [25], artificial intelligence [21], or queue theory [12]. Resource provisioning for continuum applications poses new challenges: one has to consider their low-latency nature and their highly dynamic workload (e.g., from users that quickly enter and leave a particular edge area). To tackle this problem, we are working on a self-management loop based on lightweight control theoretical algorithms (as proposed in [5]) that could handle the fast allocation of μ-service instances.

To further improve robustness and scalability, we would also like to work on a decentralized placement approach that considers the coordination among surrogate edge and cloud domains from the same provider. In the same direction, we would like to consider mobility to anticipate acquisition and allocation. Finally, we would like to explore scenarios related to computation offloading in which devices that provide more computational resources can be seen as mobile domains for constrained devices in case nearby edge domains are not available and the latency to the cloud is prohibitive.

REFERENCES

[1] 2018. Apache OpenWhisk. Retrieved from https://openwhisk.apache.org.

[2] 2018. AWS Lambda. Retrieved from https://docs.aws.amazon.com/lambda.

[3] Several authors. 2016. *Mobile Edge Computing (MEC); Framework and Reference Architecture*. Technical Report. ETSI GS MEC. Retrieved from http://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/01.01.01_60/gs_MEC003v010101p.pdf.

[4] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. 2017. Serverless computing: Current trends and open problems. *arXiv preprint arXiv:1706.03178* (2017).

[5] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi. 2016. A discrete-time feedback controller for containerized cloud applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, 217–228. DOI: https://doi.org/10.1145/2950290.2950328

[6] L. Baresi, D. F. Mendonça, and M. Garriga. 2017. Empowering low-latency applications through a serverless edge computing architecture. In *Proceedings of the 6th European Conf. on Service-Oriented and Cloud Computing*. Springer International Publishing, Cham, 196–210.

[7] M. T. Beck, M. Werner, S. Feld, and S. Schimper. 2014. Mobile edge computing: A taxonomy. In *Proceedings of the 6th International Conference on Advances in Future Internet*. Citeseer, 48–54.

[11] For example, OpenBaton: https://openbaton.github.io/.

[8] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu. 2014. *Fog Computing: A Platform for Internet of Things and Analytics*. Springer International Publishing, Cham, 169–186. DOI: https://doi.org/10.1007/978-3-319-05029-4_7

[9] ETSI Group. 2016. *Mobile Edge Computing (MEC) Terminology*. Technical Report. European Telecommunications Standards Institute (ETSI). Retrieved from http://www.etsi.org/deliver/etsi_gs/MEC/001_099/001/01.01.01_60/gs_MEC001v010101p.pdf.

[10] J. L. Garcia-Dorado. 2017. Bandwidth measurements within the cloud: Characterizing regular behaviors and correlating downtimes. *ACM Transactions on Internet Technology* 17, 4, Article 39 (2017), 25 pages. DOI: https://doi.org/10.1145/3093893

[11] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2016. Serverless computation with openLambda. In *Proceedings of the 8th USENIX Conf. on Hot Topics in Cloud Computing*. USENIX Association, Berkeley, CA, 33–39. Retrieved from http://dl.acm.org/citation.cfm?id=3027041.3027047.

[12] Y. Hu, J. Wong, G. Iszlai, and M. Litoiu. 2009. Resource provisioning for cloud computing. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., Riverton, NJ, 101–111. DOI: https://doi.org/10.1145/1723028.1723041

[13] A. Israel, A. Hoban, A. Tierno Sepulveda, F. Salguero, G. Garcia de Blase, and K. Kashalkar. 2017. *Open Source MANO Release Three – ETSI White Paper*. Technical Report. ETSI OSM Consortium. Retrieved from https://osm.etsi.org/images/OSM-Whitepaper-TechContent-ReleaseTHREE-FINAL.PDF.

[14] M. Jia, W. Liang, and Z. Xu. 2017. QoS-aware task offloading in distributed cloudlets with virtual network function services. In *Proceedings of the 20th ACM International Conf. on Modelling, Analysis and Simulation of Wireless and Mobile Systems*. ACM, New York, NY, 109–116.

[15] J. O Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (Jan. 2003), 41–50. DOI: https://doi.org/10.1109/MC.2003.1160055

[16] D. Lecompte and F. Gabin. 2012. Evolved multimedia broadcast/multicast service (eMBMS) in LTE-advanced: Overview and Rel-11 enhancements. *IEEE Communications Magazine* 50, 11 (2012), 68–74.

[17] P. Leitner and J. Cito. 2016. Patterns in the Chaos – A study of performance variation and predictability in public IaaS clouds. *ACM Transactions on Internet Technology* 16, 3 (2016), 15.

[18] James Lewis and Martin Fowler. 2014. Microservices: A definition for this new architectural term. Retrieved from http://martinfowler.com/articles/microservices.html.

[19] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief. 2016. Delay-optimal computation task scheduling for mobile-edge computing systems. *ArXiv e-prints* (April 2016). arxiv:cs.IT/1604.07525

[20] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. 2018. Serverless computing: An investigation of factors influencing microservice performance. In *Proceedings of the 6th IEEE International Conf. on Cloud Engineering (IC2E'18)*.

[21] A. Y. Nikravesh, S. A. Ajila, and C.-H. Lung. 2015. Towards an autonomic auto-scaling prediction system for cloud resource provisioning. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press, 35–45.

[22] D. L. Olson. 1996. *Smart*. Springer New York, New York, 34–48.

[23] G. Orsini, D. Bade, and W. Lamersdorf. 2016. CloudAware: A context-adaptive middleware for mobile edge and cloud computing applications. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W'16)*. 216–221. DOI: https://doi.org/10.1109/FAS-W.2016.54

[24] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. 2009. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing* 8, 4 (Oct. 2009), 14–23.

[25] S. Schulte, D. Schuller, P. Hoenisch, U. Lampe, S. Dustdar, and R. Steinmetz. 2013. Cost-driven optimization of cloud resource allocation for elastic processes. *International Journal of Cloud Computing* 1, 2 (2013), 1–14.

[26] N. Shalom, Y. Parasol, S. Naeh, and W. Yoram. 2014. *NFV and What It Means to You: From ETSI to MANO to YANG –Cloudify White Paper*. Technical Report. GigaSpaces Research, Cloudify Team.

[27] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (Oct. 2016), 637–646.

[28] T. Soyata, R. Muraleedharan, C. Funai, M. Kwon, and W. Heinzelman. 2012. Cloud-vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture. In *Proceedings of the 17th IEEE Symposium on Computers and Communications*. 59–66. DOI: https://doi.org/10.1109/ISCC.2012.6249269

[29] J. Spillner, C. Mateos, and D. A. Monge. 2018. FaaSter, better, cheaper: The prospect of serverless scientific computing and HPC. In *High Performance Computing*, Esteban Mocskos and Sergio Nesmachnow (Eds.). Springer International Publishing, Cham, 154–168.

[30] W. Tarneberg, A. Mehta, E. Wadbro, J. Tordsson, J. Eker, M. Kihl, and E. Elmroth. 2017. Dynamic application placement in the mobile cloud network. *Future Generation Computer Systems* 70 (2017), 163–177.

[31] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang. 2017. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS lambda architectures. *Service Oriented Computing and Applications* 11, 2 (2017), 233–247.

[32] N. Wang, B. Varghese, M. Matthaiou, and D. S. Nikolopoulos. 2017. ENORM: A framework for edge NOde resource management. *IEEE Transactions on Services Computing* abs/1709.04061 (2017), 1–1. DOI: https://doi.org/10.1109/TSC.2017.2753775

[33] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. K. Leung. 2017. Dynamic service placement for mobile micro-clouds with predicted future costs. *IEEE Transactions on Parallel and Distributed Systems* 28, 4 (April 2017), 1002–1016. DOI: https://doi.org/10.1109/TPDS.2016.2604814

[34] S. Wang, M. Zafer, and K. K. Leung. 2017. Online placement of multi-component applications in edge computing environments. *IEEE Access* 5 (2017), 2514–2533.

[35] J. Xu, L. Chen, and P. Zhou. 2018. Joint service caching and task offloading for mobile edge computing in dense networks. *CoRR* abs/1801.05868 (2018). arxiv:1801.05868. Retrieved from http://arxiv.org/abs/1801.05868.

[36] R. Yu, G. Xue, and X. Zhang. 2018. Application provisioning in fog computing-enabled internet-of-things: A network perspective. In *Proceedings of the 13th IEEE International Conference on Computer Communications (INFOCOM'18)*. IEEE.

[37] T. Zhao, S. Zhou, X. Guo, Y. Zhao, and Z. Niu. 2015. A cooperative scheduling scheme of local cloud and internet cloud for delay-aware mobile cloud computing. *CoRR* abs/1511.08540 (2015). arxiv:1511.08540