Accepted at: 24th Conference on Emerging Technologies and Factory Automation (ETFA), pp. 1–8, 2019, IEEE

Automatic Application Placement and Adaptation in Cloud-Edge Environments

Sebastian Meixner, Daniel Schall, Fei Li
Corporate Technology
Siemens AG, Austria
{sebastian.a.meixner, daniel.schall, lifei}@siemens.com

Vasileios Karagiannis, Stefan Schulte
Distributed Systems Group
TU Wien, Austria
{v.karagiannis, s.schulte}@infosys.tuwien.ac.at

Konstantinos Plakidas
Software Architecture Research Group
Universität Wien, Austria
konstantinos.plakidas@univie.ac.at

***Abstract*—Edge computing describes a paradigm for combining computational resources at the edge of the network with the cloud. Even though complementing the cloud with these resources provides benefits, e.g., low latency, it also introduces new challenges to the operational staff. Such challenges can be: deciding if the applications should be placed in the cloud or at the edge, and monitoring them at runtime to ensure that all the application requirements are met. This becomes more challenging when using microservices due to the complexity of the resulting placement problem. To mitigate such concerns, we introduce an automatic deployment framework along with a prototype implementation, called D-DAD. This framework provides a transparent (to the operational staff) way to deploy applications with respect to all their requirements—including the non-functional—using mechanisms for monitoring and adapting the deployments to the available resources in a cloud-edge environment. For evaluating our framework, we provide results from a series of experiments which show how the adaptation mechanism meets the application requirements, including a ∼90% reduction of CPU utilization violations, compared to using only the local resources.**

I. INTRODUCTION

Due to the rise of the Internet of Things (IoT) and edge computing, there is a tendency to exploit the computational resources at the edge of the network [1]. In an industrial context, these resources reside on premise, and may include machines that are part of an assembly line (e.g., welding robots), industrial PCs or low-power IoT devices. Apart from the computational resources at the edge, there is also the cloud which offers virtually unlimited computational resources at a remote location, i.e., in a data center [2]. However, the devices at the edge are in the proximity of each other and thus, the response times among them are reduced [3]. Nevertheless, these devices have limited resources compared to the cloud, which restricts the number of applications that can be deployed at the edge. On the other hand, deploying applications in the cloud is an option only if real-time or privacy guarantees are not primary concerns [4]. Therefore, to exploit both the cloud and the edge, some applications should run at the edge (e.g., the latency-sensitive) while others should run in the cloud (e.g., analytics, maintenance checks, etc.) [5].

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785, FORA—Fog Computing for Robotics and Industrial Automation.

In such scenarios, the operational staff needs to cope with the problem of deciding which is the most appropriate deployment location of each application that needs to be executed. This problem aggravates when using microservice architectures [6]. By using microservices, the applications can be comprised by a set of services [7]. Obliging the operational staff to place each service on the available resources becomes a tedious and error-prone task which could be automated to free the staff of this burden. Moreover, the optimal placement of the services may change at runtime due to resource/software requirements and non-functional requirements (NFR), e.g., the delay due to response times. This problem becomes even more evident when multiple services need to be deployed frequently, e.g., in a microservices/DevOps methodology [8].

In this paper, we cope with this problem by designing a framework which handles the deployment of the applications automatically, i.e., the placement decisions are transparent to the operational staff. Moreover, the presented framework allows the operational staff to define rules which correspond to the requirements of the applications, including NFRs. Furthermore, the framework monitors the performance of the applications and adapts the deployments on the available resources at runtime to avoid requirement violations.

The contributions of this work are the following: *i*) We define a reference architecture for a framework that deploys, monitors and adapts the execution of applications on both cloud and edge computational resources. *ii*) We present a method for cloud-edge placement which takes into account NFRs based on user-defined rules. *iii*) We develop a framework named D-DAD (Data-Driven Automatic Deployment), which integrates the cloud-edge placement and we evaluate it based on experiments conducted on a prototype which targets an industrial cloud-edge environment.

The rest of the paper is structured as follows: In Section II, we show how our approach compares to related work on edge computing and automatic application deployment. Section III introduces the D-DAD framework which performs transparent application placement in cloud-edge environments. Afterwards, Section IV presents the experimental results from the evaluation of the proposed approach, which focus on CPU utilization, response times and time required to calculate an optimized placement. Finally, Section V concludes this work and provides an outlook on future research directions.

II. RELATED WORK

There is a lot of recent research aiming at closing the gap between the cloud and the edge. Notably, most related work states that the goal is to extend cloud computing to the edge of the network, rather than to replace the cloud with exclusive computing at the edge [9]. The main reason for doing this is to utilize the resources at the edge of the network for the critical parts of an application which may require low latency, while using the cloud for resource-demanding parts or when the local resources are occupied.

In their seminal work on the topic, Bonomi et al. [3] discuss a conceptual middleware platform for computing using resources that span from the cloud to the edge of the network. This platform orchestrates the individual software components and provides a uniform communication mechanism. Even though this work provides sound theoretical foundation, technical implementation details are not mentioned.

Hong et al. [10] present a programming model for applications in the IoT. According to this model, an application is divided into mobile processes which are organized hierarchically. These processes are mapped on a hierarchy of distributed resources that reside in the cloud and the edge. This approach presents a solution for deploying applications in a cloud-edge environment. The D-DAD framework presented in the work at hand, also considers the adaptation of already deployed applications at runtime. This is done to ensure that the NFRs remain satisfied even when the system state changes.

Another approach for deploying applications on available computational resources is using Disnix [11]. Disnix is a toolset that allows users to declare applications and available resources, and then to map these applications on the resources. However, in this approach the placement decisions are still left to the user whereas in our approach, this is done automatically by the proposed framework.

Matougui and Leriche [12] present a constraint-based architecture for autonomic software deployment. This architecture enables users to declare constraints and attach them to applications. Potential deployment locations are discovered in the network by a dedicated service. The placement decisions are made according to the solution of a CSP (Constraint Satisfaction Problem). However, this approach does not consider NFRs. In the D-DAD framework, we provide the users with a mechanism to define applications, resources and NFRs, which infer the constraints automatically.

Skarlat et al. [13] provide a formal approach for optimizing resource allocation in cloud and edge resources. The goal of this work is to distribute the applications on the available resources in such way that the latency and the cost are minimized. However, the presented framework does not allow the users to define rules for meeting other functional and non-functional requirements whereas, the D-DAD framework allows the operational staff to model various application requirements.

Huber et al. [14] present an approach for dynamic runtime adaptation of software systems. The authors propose a modeling language for describing adaptation properties in a component-based architecture. By using this modeling language, the adaptations are modeled in an intuitive and machine-readable manner which makes the adaptation decisions reusable by different autonomic systems. The scope of the work by Huber et al. is to propose a modeling language and thus, no logic for automatic placement is discussed.

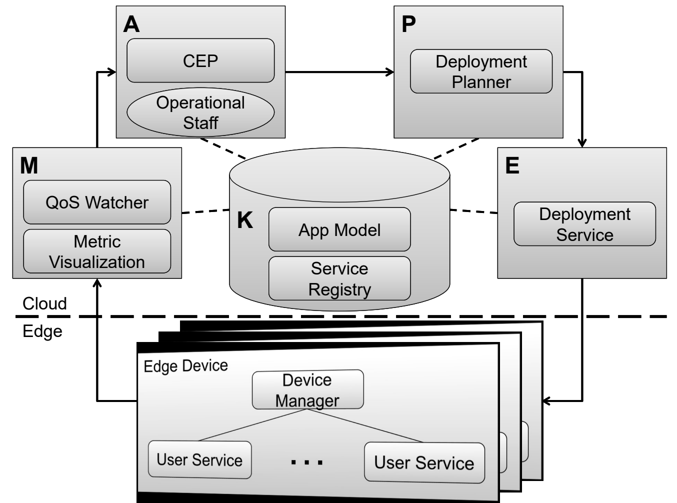


Fig. 1: Architecture of the D-DAD framework.

The D-DAD framework presented in the work at hand, provides a solution that enables deploying applications on cloud-edge resources in an automatic manner, while also monitoring and adapting the system based on user-defined rules, thus minimizing human intervention. Furthermore, D-DAD provides a holistic solution which comprises all the functionality into a single framework.

III. THE D-DAD FRAMEWORK

This section introduces the D-DAD framework for automatic application deployment in cloud-edge environments. First, we present an overview of the architecture in Section III-A and we analyze the functionality of the basic components in Section III-B. Then, we describe the main component interactions in Section III-C and afterwards, we explain the deployment process in Section III-D and the adaptation process in Section III-E.

A. High-level Architecture

The D-DAD framework follows the MAPE-K (*Monitor-Analyze-Plan-Execute* on a shared *Knowledge Base*) model (which is commonly applied in autonomic computing) because of the aim to provide automatic application deployment and minimize the human intervention [15]. MAPE-K models a system that executes a loop with four stages which all have access to a shared knowledge base. By implementing this model, the system is able to manage itself in a cycle.

Fig. 1 shows the high-level architecture of D-DAD, which follows the MAPE-K model. In this architecture, D-DAD assumes that the operational staff and/or the CEP (Complex

Event Processing) component realize the *Analyze* part of the cycle by observing the current state of the system and by taking actions accordingly. The state of the system is given by the *Monitor* part which monitors the system at runtime. The *Planning* part is realized by the Deployment Planner which queries the knowledge base for useful information and calculates an optimized plan for placing the applications on the available resources. Lastly, the Deployment Service implements the *Execute* part which performs the actual deployment of the applications. The shared *Knowledge Base* is realized by a registry service.

B. Basic Components

The basic components in the proposed architecture, also shown in Fig 1, are the following:

1) App Model: This component provides a model that aims at capturing the diversity of the applications and their requirements. An application is considered to consist of one or more services which may communicate with each other. To fully model the functionality of an application, the App Model allows users to define not only the services but also all their dependencies. Regarding the services, the modeled parameters include the NFRs (e.g., response times) and the resource requirements (e.g., CPU, memory). Regarding the dependencies, modeled parameters include the properties of the environment (e.g., offer elastic scalability), installed programming languages needed for the runtime of a service, permissions (e.g., to make a service publicly available) and licenses (e.g., to prevent the deployment on certain hosts). Moreover, services can have dependencies among each other, which is the case since many services together form one more complex application. We allow users to reuse existing services in order to build complex systems because this is common in microservice architectures which is the target of our work.

Regarding the resources, we consider a combination of cloud and edge hosts. Each service can be associated with a host, which indicates the place that the service runs. When a service is deployed on a host, a metadata file is created. This file contains the necessary information to communicate with this service, i.e., an identifier (e.g., IP address and port number) which can be used for initiating contact and the utilized communication protocol which can be HTTP, MQTT or similar [16]. Furthermore, each service is accompanied by an artifact identifier which can be used for downloading and executing the respective service.

2) Device Manager: The Device Manager is a lightweight service that runs on every participating device at the edge of the network. This is similar to what Bonomi et al. refer to as a *Foglet* [3]. This component is responsible for starting and stopping the services as well as monitoring the resource utilization (e.g., CPU utilization) and pushing it to the cloud. This is done in order to reduce the overhead of sending the monitoring information to the cloud, compared to having each service reporting to the cloud separately. To reduce the communication overhead even more, each service pushes individual monitoring information to the Device Manager through a socket using a JSON syntax which prevents the services from having to implement a message broker (e.g., using MQTT or similar).

Notably, a separate instance of the Device Manager runs on all the hosts, which provides a level of abstraction between the available resources and the system. Since the Device Manager exposes the same API from all the hosts, we do not need to be concerned with individual device settings or parameters.

3) Deployment Planner: The Deployment Planner calculates an optimized plan for placing a set of services on the available resources. To optimize the placement of the services, we formulate a CSP and we fill it with the services' dependencies from the App Model. Moreover, we consider the monetary cost of resource utilization (e.g., for using cloud resources) and the resource cost of the invoked migrations. After acquiring this information, the Deployment Planner builds the optimization model and forwards this model to a CSP solver (e.g., the Choco solver) which integrates the necessary algorithms to produce an optimized solution. This solution corresponds to the optimized placement plan for distributing the services on the available resources. After the optimized placement has been produced, the Deployment Planner translates the output of the solver to a format compatible with the internal algorithms of our system.

4) Deployment Service: The Deployment Service is responsible for initiating new deployments. This can be done manually when the operational staff triggers the deployment of a service on a host, semi-automatically using a Continuous Integration server after a new version of a service has been committed, built and passed all tests, or automatically by another component of the system. Other components that can trigger the execution of the services on the devices are: the Deployment Planner (i.e., according to an optimized placement) and the CEP Engine (see below), according to user-defined rules. Specifically, in such cases these components send a list of services to each affected Device Manager which follow the list to start/stop services. This list is also sent to the cloud for starting/stopping the services there.

Notably, the deployment actions are decoupled from the operations required to calculate an optimized placement plan. This is done so that each one of the components can be replaced without compromising the functionality of the system. This is especially useful for performing updates and maintenance tasks.

5) Monitoring Component: This component consists of two parts: the QoS (Quality of Service) Watcher and the CEP Engine. The QoS Watcher receives monitoring information from the Device Manager and forwards this information to the CEP Engine. The CEP Engine analyzes the monitoring information and decides if an alert has to be raised based on the user-defined rules, i.e., compares the monitoring information with the rules. If an alert is raised, the QoS Watcher is notified to take an action according to the alert. This component is used for creating events, i.e., if the CPU utilization of a host exceeds a certain threshold, to send a notification to the operational staff or to deploy more instances of the service.

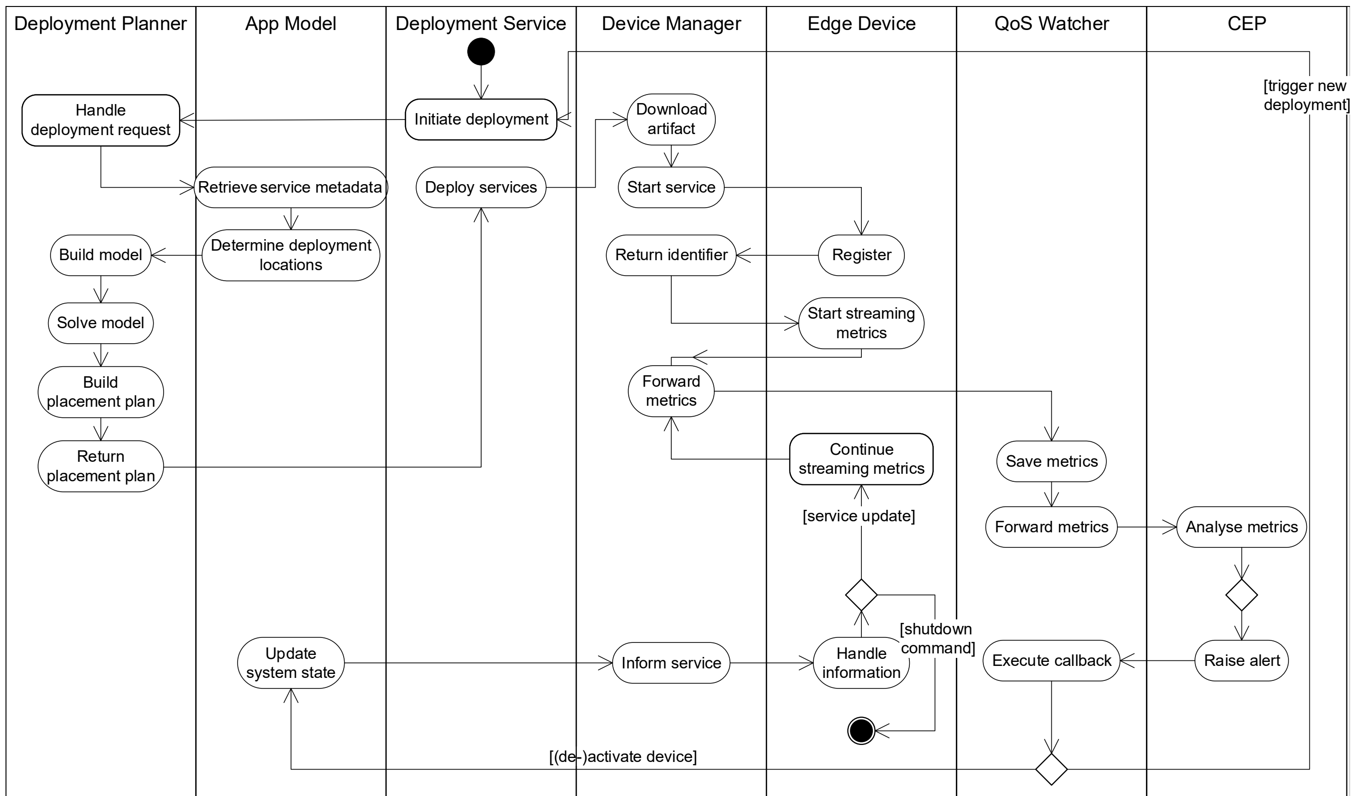


Fig. 2: Interactions among the components of D-DAD for deploying, monitoring and adapting the services.

In addition to analyzing the monitoring information and taking actions, the Monitoring Component also provides the feature to visualize the metrics. This allows the operational staff to acquire an overview of the state of the system and manually decide if adapting the deployments, the service dependencies or the rules is necessary. By enabling users to interfere with the system in this manner, we aim at providing the operational staff with the flexibility to handle undesirable system states in the way they see fit.

C. Interactions among the Components

Fig. 2 shows the main interactions among the components of the D-DAD framework, which take place for deploying, monitoring and adapting the services. First, the Deployment Service is invoked either manually or automatically, as explained in Section III-B4. Then, the Deployment Service sends a request to the Deployment Planner which contacts the App Model. The App Model gathers information about the current state of the system and the metadata of the services. Based on the metadata, the App Model also decides which hosts can be considered as potential deployment locations (based on the available resources) for the services and forwards all this information to the Deployment Planner. Upon receiving the required information, the Deployment Planner builds and solves a CSP in order to produce an optimized placement plan which contains the mapping of the services on the hosts. This plan is sent to the Deployment Service which deploys the services by instructing the Device Managers (of the affected devices) to download the respective artifacts and to start the services.

Upon initialization, each service registers with the Device Manager and starts streaming monitoring metrics (regarding resource utilization). The Device Manager sends these metrics to the QoS Watcher which stores them for further analysis and also forwards them to the CEP Engine. The CEP Engine examines and compares the metrics with the user-defined rules. If one or more of the rules are broken, the CEP Engine raises the respective alerts and notifies the QoS Watcher which stores the callbacks associated with these alerts. The QoS Watcher executes the callbacks, which usually result in an automatic update of the system state. An update can mean that, e.g., a new deployment is triggered or that some services should stop in order to reduce the load on the hosts. Instead of an automatic update, the callback can also result in sending a notification to the operational staff, if this is preferred.

D. Cloud-Edge Deployment

During the cloud-edge deployment, the Deployment Planner solves a CSP and produces a placement plan which meets all the requirements of the services. Before accepting a placement plan is valid, we make sure that four prime constraints are met: *i*) Each service is associated with one host. *ii*) Each service has the necessary permissions and licenses to be deployed at the host that the placement plan indicates. *iii*) The resource demands of a service do not exceed the available resources of the host. *iv*) Hosts can provide the required software–

considering also the compatibility with the the versions of the required software–to run the services.

To ensure *i*), we use an array of variables μ with size $|S|$, whereby S is the set of services and we restrict the domain of variables from 1 to $|H|$, whereby H is the set of potential (cloud and edge) hosts. The value h at index s means that the service indicated by s is deployed at host h. This way, each service is associated with exactly one host. To ensure *ii*), we restrict μ_s to certain hosts such that:

$$\forall s \in S : l_{\mu_s} \in \alpha_s$$

whereby α_s is the set of locations of allowed hosts which can be used for the deployment of the service and l_{μ_s} is the location of the selected host. To ensure *iii*), we define the variables δ, ρ and ϱ. δ_h is the set of services deployed at host h, ρ_h^r indicates the available resources of type $r \in R$ (e.g., CPU), whereby R is the set of the overall resources at host h. ϱ indicates the resource demand of resource r for service s. For each host h with deployed services d_h this constraint applies:

$$\forall r \in R : (\sum_{s \in \delta_h} \varrho_s^r) \leq \rho_h^r$$

Finally, to ensure *iv*) we define the variables σ and ς. σ_h^r indicates the set of available versions of software $sw \in SW$, whereby SW is the set of all the available software at host h. ς indicates the compatible versions of software sw with service s using the following constraint for each service s which is deployed on a host h:

$$\forall sw \in SW : (\sigma_h^{sw} \cap \varsigma_s^{sw}) \subseteq \varsigma_s^{sw}$$

When defining the cost function of a placement plan in the CSP, we model cloud and edge resources differently with regard to cost. This is done because the resources of the cloud incur additional (leasing) cost compared to the resources at the edge, which reside on premise and thus, require no additional cost. Moreover, the cost function takes into account the number of migrations resulting from an optimized deployment plan, since migrations also incur additional (resource- and overhead-related) cost. To allow the operational staff to decide how the cost affects the objective function, we provide the possibility to assign weights on the different parts of the objective function.

If the optimization process does not produce a valid placement plan, i.e., there is no possible mapping of the services on the hosts, which meets all the requirements, the operational staff is informed. In this case, the staff should reconsider the requirements and the available resources.

E. Monitoring and Adaptation

When running applications in a production environment, it is very important to monitor the runtime behavior. There is plenty of motivation for doing this, e.g., to ensure that all the applications operate properly, to make sure that all the requirements are met or to observe if additional resources are necessary. Apart from providing this kind of insights into the applications' behavior, monitoring can also be used for adapting the applications on the available resources at runtime.

To achieve this, we use the Device Manager which is installed on every host of our system. Each running service shares its process ID with the Device Manager. Through this ID, the device manager collects information about the utilized resources of each service, e.g., CPU and memory utilization. However, there are additional service-specific metrics that each service sends to the Device Manager, e.g., RTT (Round Trip Time) of requests, capacity/availability of queues, and task execution times. To achieve this, when a service registers with the Device Manager upon initialization, the Device Manager assigns an identifier (e.g., IP address and port number) to the service, through which the service streams the application-specific metrics. These metrics are stored locally but also, they are forwarded to the cloud for further processing and analytics.

The monitoring information of the Device Manager is then used for raising alerts at runtime, according to the user-defined rules. These rules refer to conditions that automatically trigger certain events. Such conditions can be related to resource utilization (e.g., the CPU utilization is above a specific threshold for a certain amount of time), response times (e.g., the communication delay between services which are hosted on different locations), etc. The aim of these rules is to adapt the execution of the services to the available resources and to ensure that all the services' requirements remain satisfied at runtime. The outcome of these rules can be: the notification of the person in charge (e.g., via e-mails or pop-up alerts), the reduction of the number of services on a specific host, the scaling of a service, etc. Moreover, if the performance of the system is not satisfactory, a user-defined rule can trigger the redeployment of certain/all the services, which may lead to better placement decisions. Thus, the goal of this runtime adaptation mechanism is to prevent the violation of the services' requirements by taking appropriate actions to refine the system execution.

IV. EVALUATION

In this section, we present the evaluation of the proposed framework. First, we describe the application scenario which we consider for this evaluation in Section IV-A and then, we present a prototype implementation along with technical details, which is used for running the experiments, in Section IV-B. Finally, we analyze the results of the evaluation in Section IV-C. Specifically, these results are related to the runtime adaptation of the services (cf. Section IV-C1) and the service placement process (cf. Section IV-C2).

A. Evaluation Scenario

For the evaluation scenario, we consider a basic analytics use case prominently featured in industrial contexts. In this scenario, the goal is to predict the state of the machines and devices, based on sensor readings and by using machine learning techniques [17]. We assume that there is a service in place, which sends the sensor readings to the cloud in order

to train a machine learning model which classifies the state of a machine. Hereinafter, we refer to this process as *scoring*.

Traditionally, the process of training the machine learning model occurs in the cloud due to the required (intense) processing. However, the placement of scoring is decided by the operational staff, i.e., either in the cloud or locally at the edge. The former case incurs communication delay and additional monetary cost. The latter bears no long-distance communication delay and no additional cost (see Section III-D). However, the primary tasks of the edge devices in the industry, have stringent requirements, e.g., the operation of a welding robot. Therefore, it is very important to make sure that such critical tasks are not affected by secondary processes like scoring. To this end, if the resources at the edge operate at full capacity executing critical tasks, the scoring should be performed in the cloud. To avoid a slow and error-prone human intervention, switching between edge and cloud should be done dynamically by the framework according to resource utilization and the requirements of the deployed services.

Considering this scenario, in this evaluation we employ the D-DAD framework to solve this problem by automatically deciding if the scoring service should be deployed at the edge or in the cloud. Moreover, once the services are deployed, the framework monitors and adapts the deployments at runtime in order to satisfy all the requirements, e.g. latency delays, resource utilization. The aim of considering this scenario is to demonstrate that the proposed framework can unify cloud and edge resources in a transparent manner and exploit their complementary characteristics to ensure that all the requirements (including NFRs) are met.

B. Evaluation Environment

The edge devices are emulated using Amazon EC2 instances. The selected instances we use have a single 2.4 GHz CPU core and 0.5 GB of RAM. Since the resources at the edge of the network are expected to be limited, we assume that these capacities resemble the resources of an edge device. On such an emulated edge device, we install a Device Manager and a data acquisition software which provides the sensor readings. Moreover, we install a service which sends the readings to the cloud in order to train the scoring process. The scoring process is also loaded on a device at the edge for performing local scoring (although training occurs only in the cloud).

For this evaluation, we monitor the state of 45 machines and thus, every second there are 45 requests to the scoring services. To enable the D-DAD framework to scale in order to cope with the potential load from managing an industrial environment, we deploy all the framework-internal services, apart from the Device Manager, in the cloud (as shown in Fig. 1). Finally, in order to simulate the primary tasks (i.e., the critical services) of the edge devices, which must not be affected by the scoring, we use a script that occupies 30–60% of the CPU (randomly using a uniform distribution).

The local scoring service is implemented in C# and includes bindings for the R programming language in order to use additional R libraries. By default, the scoring takes place locally due to the benefits of utilizing the resources at the edge. This continues until the CEP Engine instructs the framework to deactivate local scoring, which forces the service to forward the data to the cloud and execute the scoring there. The CEP Engine is configured to raise the alert which causes this behavior when the CPU utilization of an edge device exceeds 75%. This aims at preventing a device from becoming overloaded so that the primary tasks of the edge devices are not affected.

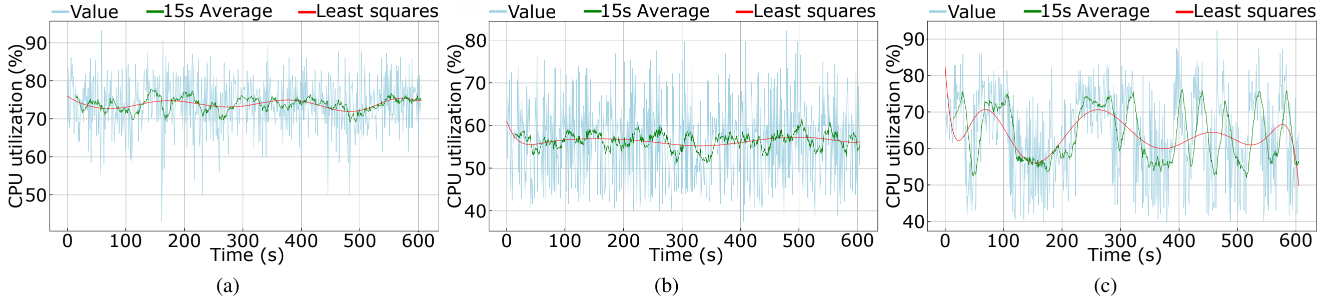
Moreover, we configure an alert when the RTT of the scoring service takes longer than 600 milliseconds. This specific value is selected because as long as an action responds within a second, the framework is still estimated to be responsive to the operational staff [18]. To avoid being affected by performance spikes, we measure these values based on 15-second averages.

After intense experimentation with our framework, we also define the following additional conditions in order to adhere even better to the aforementioned NFRs (i.e., the CPU utilization and the RTT): *i*) If the CPU utilization in the last 15 seconds has an average value which is greater than 70% and a maximum value which is greater than 90%, move the scoring to the cloud. *ii*) If the RTT of the requests to the scoring service in the last 15 seconds has an average value of above 550 milliseconds and a maximum value of above 900 milliseconds, move the scoring to the edge.

C. Evaluation Results

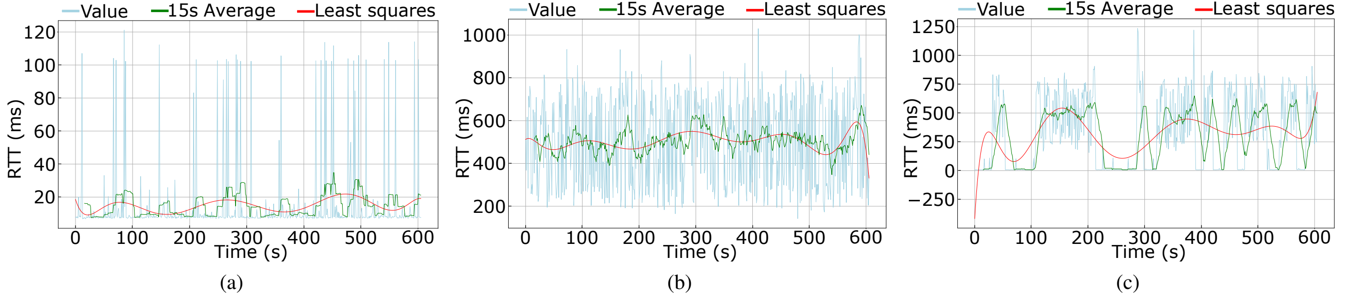
1) Runtime Adaptation: The goal of this experiment is to show that by using the proposed framework, we also adhere to the NFRs of a service (as much as possible). To this end, we define two NFRs as explained in Section IV-B, i.e., the CPU utilization of the edge devices (per 15-second averages) should remain under 75% and the RTT of the requests to the scoring service should take less than a second. For these two metrics, we plot figures (see Fig. 3 and 4) which show the values of the measurements per second (blue lines) and per 15-second average (green lines). Also, since we determine the state of multiple devices, i.e., 45 machines per second, we compute the exponential weighted average using the least squares method with a smoothing factor of 0.5 and we plot it per second (red lines), to show the general tendency of the framework. The alerts of violating the NFRs are: to perform the scoring locally if the RTT takes a long time and to perform the scoring in the cloud if the CPU utilization is high.

Fig. 3 shows the CPU utilization throughout 10 minutes of execution time. Fig. 3a shows the CPU utilization of performing the scoring exclusively at the edge, which remains mostly between 65% and 85%. The 15-second average of the CPU utilization exceeds the threshold 158 times. Fig. 3b shows the CPU utilization when scoring occurs exclusively in the cloud, which remains mostly between 40% and 80%. However, the 15-second average is substantially lower than when using local scoring (i.e. at the edge) and never exceeds the threshold. Fig. 3c shows the CPU utilization when we enable runtime adaptation using the user-defined rules, as described in Section IV-B. The peaks and the valleys of the



(a) (b) (c)

Fig. 3: CPU utilization for local scoring (a), cloud scoring (b) and with runtime adaptation (c).



(a) (b) (c)

Fig. 4: RTT of requests for local scoring (a), cloud scoring (b) and with runtime adaptation (c).

figure show when scoring occurs at the edge or in the cloud, respectively. The 75% threshold is exceeded 13 times, which indicates a reduction of nearly 90%, compared to local scoring.

Fig. 4 shows the RTT of a request to the scoring service, i.e., the period from the time a sensor reading is received from the data acquisition service until the response is received and processed. Fig. 4a shows the RTT of using only local scoring, which has a maximum value of nearly 120 ms since no data needs to be transferred to the cloud. The RTT threshold of one second is never exceeded. Fig. 4b shows the RTT of scoring in the cloud which takes substantially longer since the requests are sent over the Internet. The RTT threshold of one second is exceeded 2 times. Fig. 4c shows the RTT of scoring when we enable runtime adaptation using the user-defined rules, as described in Section IV-B. The peaks and the valleys of the figure correspond to scoring in the cloud or the edge, respectively. The RTT threshold of one second is exceeded 3 times.

Notably, this experiment shows that for the reduction of 90% of NFR violations with regard to the CPU utilization, there are only 3 violations with regards to the RTT. It should also be noted that the cloud is utilized for a smaller amount of time since the scoring occurs also at the edge, which lowers the monetary cost of using cloud resources.

Finally, we show Table I to summarize the observed NFR violations. For these results, we repeat the experiment 3 times for each method (i.e., cloud scoring, local scoring, or using adaptation). This table shows that when using the runtime adaptation, the NFR violations related to the CPU utilization drop significantly, compared to local scoring. At the same time, the NFR violations related to RTT are only increased slightly. Compared to cloud scoring, at a first glance it seems that the runtime adaptation causes slightly more NFR violations. However, when looking closely at Fig. 3 and 4, we note that the runtime adaptation is able to provide the low CPU utilization of the cloud scoring with the low RTT of the local scoring. For this reason, we consider these results satisfactory since we combine cloud and edge resources and we enable the operational staff to define rules for using these resources according to the requirements of the services, in a transparent manner.

2) Service Placement: In this experiment, we examine how much time is required to solve a CSP and produce an initial placement plan. To do this, we generate a multitude of fictional services and hosts. Regarding the hosts, we define four types of edge devices, each one integrating different software and resource capacities. Regarding the services, we define five different types with different software and resource dependencies. Moreover, we define the placement options of the services, i.e., only at the edge, only in the cloud and either

TABLE I: NFR violations when using different methods.

Metric	Method	Average	σ (std. deviation)
Latency	Local	0%	0%
	Cloud	0.39%	0.08%
	Mixed	0.94%	0.34%
CPU Load	Local	34.28%	15%
	Cloud	0%	0%
	Mixed	1.94%	0.21%

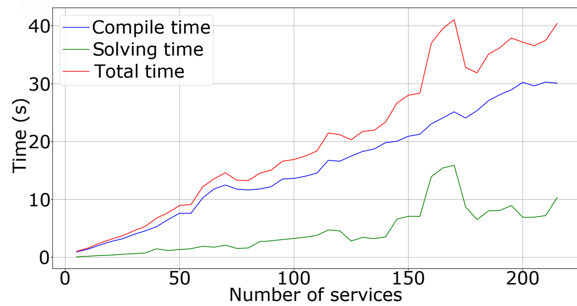


Fig. 5: Compile and solving time of a CSP with 60 hosts.

at the edge or the cloud. Commonly, the resource-demanding applications are deployed in the cloud while the applications with low resource demands run at the edge. Therefore, we define the cloud-only services with high resource demands, the edge-only services with low resource demands and the services which can be deployed in both types of resources, with moderate resource demands.

Fig. 5 shows the required time for solving the CSP when using 60 hosts and an increasing number of services. At the end of the experiment, 215 services are deployed. For this number of services whereby ∼25% of them are edge-only, ∼25% of them are cloud-only, and ∼50% of them can be deployed on both resource types, solving the CSP takes approximately 10 seconds, while the compilation of the model takes approximately 30 seconds.

Due to the complexity of the problem, a large number of hosts and services require the respective amount of time for producing a solution. However, for the intended use, i.e., the operational staff deploying services on premise with occasional use of the cloud, the performance is satisfactory. Notably, we stop the solving process after the first solution is produced. Even though this does not guarantee an optimal solution, it does guarantee the fastest feasible solution. Presumably, allowing the solving time to execute for longer time periods, may produce better placements.

V. Conclusion

In this paper, we present the D-DAD framework which unifies cloud and edge resources and provides a way to perform application placement in an automatic manner. Moreover, the framework is able to integrate user-defined rules which are taken into account when producing a placement plan. These rules are also considered during the adaptation process of the framework which occurs at runtime and ensures that all the requirements of the applications remain satisfied. To evaluate this approach, we build a prototype and we perform a series of experiments which focus on CPU utilization and RTT of the requests. The results show that our framework can provide low CPU utilization due to using the resources in the cloud combined with low RTT of the requests due to using the resources at the edge.

Future plans in this line of work include designing a service which can automatically detect computational resources that join and leave the system in order to integrate such resources without user intervention. Another promising research direction would be to improve the optimization process of the application placement. Currently, the D-DAD framework performs application placement based on a CSP which may take a long time to solve, if very large numbers of hosts and applications are used. Therefore, researching ways to reduce the complexity of this problem or working on heuristics for approximating the optimal solution in a timely manner, can be useful.

References

[1] R. Mahmud, R. Kotagiri, and R. Buyya, "Fog computing: A taxonomy, survey and future directions," in *Internet of everything*, pp. 103–130, Springer, 2018.

[2] O. Skarlat, V. Karagiannis, T. Rausch, K. Bachmann, and S. Schulte, "A framework for optimization, service placement, and runtime operation in the fog," in *11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pp. 164–173, IEEE, 2018.

[3] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for internet of things and analytics," in *Big Data and Internet of Things: A Roadmap for Smart Environments*, pp. 169–186, Springer, 2014.

[4] S. Yi, C. Li, and Q. Li, "A survey of fog computing: concepts, applications and issues," in *Workshop on Mobile Big Data*, pp. 37–42, ACM, 2015.

[5] A. V. Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh, and R. Buyya, "Fog Computing: Principles, Architectures, and Applications," in *Internet of Things: Principles and Paradigms*, ch. 4, pp. 61–75, Morgan Kaufmann, 2016.

[6] M. Fowler and J. Lewis, "Microservices," *http://martinfowler.com/ articles/microservices.html [cited June 19, 2017]*, 2014.

[7] J. Bosch, "Speed, data, and ecosystems: the future of software engineering," *IEEE Software*, vol. 33, no. 1, pp. 82–88, 2016.

[8] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[9] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 27–32, 2014.

[10] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: A programming model for large-scale applications on the internet of things," in *2nd ACM SIGCOMM Workshop on Mobile Cloud Computing*, pp. 15–20, ACM, 2013.

[11] S. Van Der Burg and E. Dolstra, "Disnix: A toolset for distributed deployment," *Science of Computer Programming*, vol. 79, pp. 52–69, 2014.

[12] M. E. A. Matougui and S. Leriche, "A middleware architecture for autonomic software deployment," in *7th International Conference on Systems and Networks Communications*, pp. 13–20, XPS, 2012.

[13] O. Skarlat, S. Schulte, M. Borkowski, and P. Leitner, "Resource Provisioning for IoT Services in the Fog," in *9th IEEE International Conference on Service Oriented Computing and Applications*, pp. 32–39, IEEE, 2016.

[14] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, and S. Kounev, "Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments," *Service Oriented Computing and Applications*, vol. 8, no. 1, pp. 73–89, 2014.

[15] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[16] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate, "A survey on application layer protocols for the internet of things," *Transaction on IoT and Cloud Computing*, vol. 3, no. 1, pp. 11–17, 2015.

[17] G. A. Susto, A. Schirru, S. Pampuri, S. McLoone, and A. Beghi, "Machine learning for predictive maintenance: A multiple classifier approach," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 3, pp. 812–820, 2014.

[18] J. Nielsen, *Usability engineering*. Elsevier, 1994.