A Goal-driven Approach for Deploying Self-adaptive IoT Systems*

Fahed Alkhabbas[†], Ilir Murturi[*], Romina Spalazzese[†], Paul Davidsson[†], and Schahram Dustdar[*]
[†]Internet of Things and People Research Center, Malmö University, Sweden
[†]Department of Computer Science and Media Technology, Malmö University, Sweden
{fahed.alkhabbas, romina.spalazzese, paul.davidsson}@mau.se
[*]Distributed Systems Group, TU Wien, Austria
{imurturi, dustdar}@dsg.tuwien.ac.at

***Abstract*—Engineering Internet of Things (IoT) systems is a challenging task partly due to the dynamicity and uncertainty of the environment including the involvement of the human in the loop. Users should be able to achieve their goals seamlessly in different environments, and IoT systems should be able to cope with dynamic changes. Several approaches have been proposed to enable the automated formation, enactment, and self-adaptation of goal-driven IoT systems. However, they do not address deployment issues. In this paper, we propose a goal-driven approach for deploying self-adaptive IoT systems in the Edge-Cloud continuum. Our approach supports the systems to cope with the dynamicity and uncertainty of the environment including changes in their deployment topologies, i.e., the deployment nodes and their interconnections. We describe the architecture and processes of the approach and the simulations that we conducted to validate its feasibility. The results of the simulations show that the approach scales well when generating and adapting the deployment topologies of goal-driven IoT systems in smart homes and smart buildings.**

***Index Terms*—Deploying Self-adaptive IoT Systems; Goal-driven IoT Systems; Edge-Cloud Continuum; Software Architecture.**

I. Introduction

The Internet of Things (IoT) has enabled objects and devices, such as sensors, actuators, and appliances to connect and collaborate to achieve user goals. This has opened up for the development of novel types of applications in different domains, such as building automation, transportation, logistics, and health-care [4], [16]. Engineering IoT systems is a challenging task partly due to the *dynamicity* and *uncertainty* of the environment including *involvement of the human in the loop*. Users should be able to achieve their goals seamlessly in different environments. Also, IoT systems should be able to cope with dynamic changes, such as the sudden unavailability of devices and changes in their deployment topologies, i.e., the deployment nodes and their interconnections [7].

A *Goal-Driven* IoT System (GDS) is composed of a set of devices with individual functionalities that connect and cooperate temporally to achieve the user goal. In previous studies, we referred to GDS as Emergent Configurations [1], [10]. For instance, a GDS could be dynamically formed by connecting a light sensor and two connected lamps to achieve the goal "adjust the light level" in a smart meeting room. Another GDS could be dynamically formed in a hotel room by exploiting an available light sensor, a connected lamp, and connected curtains to achieve the same goal of adjusting the light level. A more complex example is to dynamically form IoT systems to support users to evacuate a building during an emergency (e.g., fire) based on available devices and data.

Several approaches have been proposed to enable the dynamic formation, enactment, and self-adaptation of GDS. However, they do not address deployment issues, such as the automated placement of the GDS functionalities within a hardware infrastructure. Additionally, among the works that focus on the deployment of IoT systems, very few aim at enabling the systems to self-adapt to dynamic changes in their deployment topologies, such as, the sudden unavailability of the deployment nodes and the degradation of the quality of their services or interconnections [7], [28].

In general, IoT systems can be deployed according to the following models [3], [30]:

1) *Everything in the Cloud model*: In this model, the software components of IoT systems are placed in the Cloud. It is suitable when the systems require significant elastic processing and storage capabilities or when their constituents are scattered in various areas.
2) *Everything in the Edge model*: The software components of IoT systems are placed in networks of more constrained devices with respect to the Cloud (e.g., local servers and gateways) that are at the Edge of the network. In this model, the computational capabilities are lower than those within the Cloud-based model.
3) *Hybrid Edge-Cloud model*: The software components of IoT systems are distributed across the Cloud and the Edge of the network. Thus, it enables exploiting the advantages of the other two models. For instance, it supports deploying the components that should perform processes with low latency in the Edge of the network, and deploying those that perform resource-demanding processes in the Cloud.

These deployment models have different properties in terms of response time, availability, privacy, and other quality characteristics [3].

The deployment of GDS is a complex process partly due to the following reasons. The number and types of the things that constitute a GDS formed to achieve a goal can be different

978-1-7281-4659-1/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSA47634.2020.00022

from an environment to another based on the available devices. Moreover, the dynamically formed GDS may have different performance requirements based on the requested goals. Additionally, the hardware infrastructures in different environments are heterogeneous with respect to their processing and storage capabilities and interconnections. Furthermore, changes in the status of the systems constituents and/or the deployment topologies can happen suddenly.

To address the aforementioned challenges, we propose a *goal-driven* approach for deploying GDS in the Edge-Cloud model. Our approach consists of an *architecture* and *processes* that enable the deployment of GDS and also support them to *self-adapt* to dynamic changes in their deployment topologies and in the status of the IoT things that constitute them. The proposed approach is goal-driven as the placement of the functionalities of the GDS on the dedicated hardware resources should take into consideration the goals. For instance, more powerful resources should be dedicated to supporting the evacuation of a building due to fire compared to those dedicated to supporting a user to give a presentation. To validate the feasibility of the approach, we implemented a prototype and simulated the deployment of GDS in a smart home and a smart building environments.

The remainder of this paper is organized as follows. Section II discusses related work. Section III introduces the approach. Section IV presents the prototype implementation. Section V presents the simulations performed to validate the feasibility of the approach. Section VI discusses the approach. Finally, Section VII concludes the paper and outlines future work directions.

II. Related Work

Several approaches have been proposed to enable the dynamic formation and adaptation of goal-driven IoT systems. Mayer et al. [25] proposed an approach where the capabilities of IoT devices are modeled as semantically annotated services to enable the dynamic composition of goal-driven IoT mashups. The approach also supports the automated adaptation of the mashups apropos the availability of the services. In [2], we proposed an approach for enabling the automated formation and adaptation of goal-driven IoT systems by exploiting context-awareness and AI-planning techniques. De Sanctis et al. [12] proposed a service-based approach for dynamically forming goal-driven IoT systems taking into consideration the quality aspects of available things (e.g., energy consumption and response time). Tsigkanos et al. [34] proposed a goal-driven approach for engineering resource coordination at runtime, tailored for the decentralized and pervasive systems. The approach considers dependencies among IoT things in order to achieve a particular goal. None of the existing approaches that enable the dynamic formation and/or adaptation of goal-driven IoT systems addresses deployment issues.

The challenges introduced in the Edge-Cloud continuum [13], such as heterogeneity, resource-constrained, and volatile environments have grabbed the attention of many researchers, resulting in proposing various techniques for addressing the resource management problem (i.e., application placement [24] or resource discovery [27]). In Edge Computing, the application placement problem has been widely studied by considering various factors such as computation and communication time [21], [36], data size [32], cost [18], [20], and user-application context [11], [20]. Skarlat et al. [33] introduced a novel approach on how to place IoT services on Edge resources optimally by considering Quality-of-Service (QoS) constraints like deadlines on the execution time of applications. The service placement problem is formulated as an integration linear programming problem where the placement solutions are evaluated in terms of the cost of execution and QoS adherence. Consequently, the services are placed on Edge nodes (if possible). In case of any failure, the services are allocated to the Cloud.

Concerning the research papers mentioned above, the majority of the proposed methodologies often rely on mathematical approaches attempting to optimize different trade-offs (e.g., latency and resource utilization). In case of failures, in the majority of the approaches, applications are deployed in the Cloud. In contrast, our approach enables the dynamic generation of deployment topologies based on the complexity of users' goals, the performance requirements, and considering available resources on the hand. Notably, this enables a more efficient utilization of resources.

Very few works aim at enabling IoT systems to self-adapt to dynamic changes in their deployment topologies [7], [28]. Contreras et al. [19] proposed an architecture for supporting the availability of services in mobile and dynamic Edge environments. When an Edge node (e.g., a laptop) that provides a service moves, becomes unreachable, or is about to run out of battery, the service consumers elect another node that runs a replica of the service. The selection is made by performing heuristics on the hardware capabilities and battery levels of available nodes. Unlike our approach, the mentioned work does not consider resources at the Cloud, network latency, and performance requirements. Additionally, if two Edge nodes have equal battery levels, a service will be deployed to the node that has the highest computational capabilities, although the other node might be able to run it satisfactorily. Our approach enables more efficient utilization of the resources in the Edge-Cloud continuum by evaluating the complexity of the goals and generating deployment topologies accordingly.

Filiposka et al. [15] introduced a novel location-aware resource management technique for Edge Computing to support the automated migration of live services when users or devices move. The mentioned work and our approach are complementary to each other. In [5], the authors consider Edge infrastructures as resource-constrained environments where IoT systems are modeled as directed acyclic graphs and deployed in many nodes. The systems are deployed using a decentralized algorithm that also enables the dynamic re-deployment of their software components to meet Service Level Agreements (SLAs). Unlike our approach, the mentioned work does not support hybrid deployment models and is not goal-driven. Similarly, Sahni et al. [31] proposed a novel task allocation

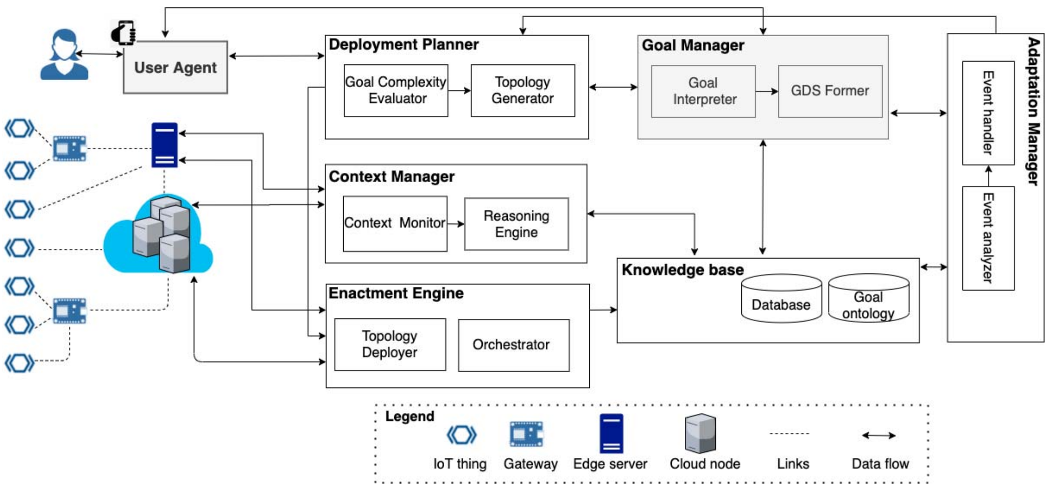

Fig. 1. The architecture of the approach

technique based on an improved genetic algorithm that aims at optimizing the total energy consumption of all tasks deployed at the Edge. Unlike our approach, the proposed technique does not support IoT applications to adapt to the changes in their deployment topologies.

To summarize, the novelty of our approach lies in enabling the dynamic deployment of GDS considering the complexity of the goals they aim to achieve and the performance requirements. Also, our approach enables GDS to self-adapt in response to dynamic changes in the status of their constituents and also their deployment topologies.

III. The Approach

Following an in-depth analysis of the related studies presented in Section II, in this section, we present the architecture and processes of the goal-driven approach for deploying GDS. The approach exploits the notion of Monitor-Analyze-Plan-Execute plus Knowledge (MAPE-K) loop adopted from the field of Self-adaptive Systems [22].

A. Architecture

Figure 1 illustrates the architecture of the approach. It comprises seven main components as described in the following.

User Agent. It is an application that runs on one (or more) of the available smart devices (e.g., smartphone). It enables users to express their goals and interact with the system. Exploring this component is out of the scope of this paper as we focus mainly on addressing deployment issues. Some studies (e.g., [23], [25]) explore the design of user agents. However, more efforts are needed to enable users to interact with GDS effectively.

Goal Manager. This component comprises two sub-components responsible for forming GDS that achieve users' goals (if possible). The *Goal Interpreter* is responsible for analyzing the goals in the context of their spatial boundaries (e.g., room, building). The *GDS Former* is responsible for dynamically forming GDS (if possible). More specifically, based on the functionalities and the things available within a goal's spatial boundaries, the GDS former is responsible for identifying: (i) the set of functionalities needed to achieve the goal; (ii) the things that either can perform the functionalities or their input is needed to enact them; (iii) the order in which the functionalities should be enacted. For instance, to adjust the light level in a room, the functionalities could be the following:

1) Get the current light level: this functionality can be performed by a light sensor.
2) Specify the suitable light level: this functionality can be performed, for instance, by a camera and an application that recognizes the ongoing activity (e.g., presentation) and specifies the suitable light level accordingly.
3) Set the specified light level: this functionality can be performed by actuators, such as connected curtains and/or lamps.

To achieve the goal, the above functionalities should be executed in the following order 1) then 2) and then 3).

As presented in Section II, several approaches have been proposed to enable the dynamic formation of goal-driven IoT systems by exploiting different techniques from different domains. Thus, the exploration of this component is out of the scope of this paper. In our prototype, we leveraged an extended version of the ECo-IoT approach [2], which is well-aligned with our vision and uses open source technologies to realize GDS.

Deployment Planner. This component comprises two sub-components responsible for planning the deployment of GDS. The *Goal Complexity Evaluator* is responsible for evaluating the complexity of users' goals using the following metrics [14], [17]:

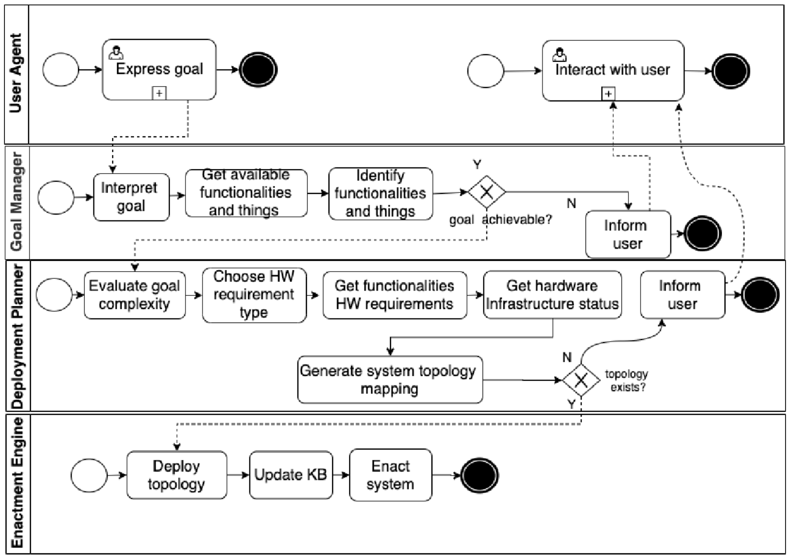

Fig. 2. The process of deploying GDS

1) The number of the leaf sub-goals of a goal in the ontology: Leaf goals are atomic goals that cannot be decomposed. On this metric, a goal is considered *primitive* if the number of its leaf sub-goals is less than or equal to seven, and *complex* if the number is greater than seven.
2) The number of things within the spatial boundaries of the goal: The knowledge base has information about the locations of the things. On this metric, a goal is considered *primitive* if the number of the things within its spatial boundaries (e.g., room, building) is less than or equal to fifty, and *complex* if the number is greater than fifty.
3) The depth of the decomposition tree of the goal in the ontology: The ontology represents the decomposition trees of goals via the relation "has sub-goal" (see below). On this metric, a goal is considered *primitive* if the depth of its decomposition tree is less than or equal to three, and *complex* if the depth is greater than three.

A goal is considered *primitive*, if it is primitive on all the above metrics, otherwise it is considered *complex*. In this paper, we consider these two options only, we plan to extend this classification criteria to cover the option(s) between them.

The *Topology Generator* is responsible for generating deployment topologies for the GDS. A topology specifies the deployment nodes that run the GDS functionalities, the communication links among the nodes, and the links between the nodes and GDS things.

Context Manager. This component comprises two sub-components responsible for maintaining and reasoning about the context of GDS. In this paper, the *Context Monitor* is responsible for monitoring hardware infrastructures and updating the Knowledge base when changes occur. More specifically, it monitors the following:

1) The availability status of Cloud and Edge nodes, their QoS, and hardware utilization rates.
2) The availability status of the things, their locations, and battery levels (when applicable).
3) The status and latency of the communication links.

We do not investigate how the monitoring services are implemented, a study that addresses relevant aspects is presented in [6]. The QoS of the deployment nodes is evaluated by comparing the time it takes the nodes to execute the functionalities at runtime with the time specified in the knowledge base for the different categories of software functionalities (see below).

The *Reasoning Engine* is responsible for analyzing the changes in the context and triggering events that reflect them. The reasoning engine can trigger two categories of events:

1) Category one (C1): This category comprises the following types of events: a deployment node is not available; the QoS of a node has degraded; a link is not available; a link experiences a high latency.
2) Category two (C2): This category comprises the following types of events: a thing has moved out of the spatial boundaries of a GDS that comprises it; a thing is disconnected or turned off.

Enactment Engine. This component comprises two sub-components responsible for deploying GDS and enacting them. The *Topology Deployer* is responsible for deploying GDS software functionalities within hardware infrastructures as specified in the topologies generated by the deployment planner. The *Orchestrator* is responsible for enacting the functionalities of GDS in the order specified by the goal manager.

Adaptation Manager. This is an event-driven component that comprises two sub-components responsible for adapting

TABLE I
THE CLASSIFICATION OF THE SOFTWARE FUNCTIONALITIES BASED ON THEIR HARDWARE REQUIREMENTS

Category	CPUs	RAM (GB)	HDD (GB)
Small	1	1	10
Medium	2	3	30
Large	>=4	>=4	>=40

GDS in response to the events triggered by the context manager. The *Event Monitor* is responsible for monitoring the knowledge base and identifying unprocessed events. The *Event Handler* is responsible for analyzing how the detected events affect the GDS whose enactment has not completed and for triggering the appropriate adaptation processes.

Knowledge Base. This component is the container of the GDS context. It comprises two sub-components, a *Database* (DB) and a *Goal Ontology*. The DB has information about the following:

1) Software functionalities: For each functionality, the DB stores information about: the software dependencies and the hardware requirements needed to enact it. For instance, a functionality should be deployed to a node that runs a Java Virtual Machine.

   The *hardware requirements* are classified into *minimal* and *optimal*. The former specifies the critical hardware resources (i.e., CPU, RAM, and storage) needed to enact the functionality. In contrast, the latter specifies the least hardware resources that achieve maximum performance (i.e., adding more resources will not improve the execution time of the functionality). Each functionality is automatically classified into one of the categories presented in Table I based on the average of their minimal and optimal hardware requirements. We assume that these requirements are specified at design time.
2) Hardware infrastructures: The DB stores information about the available Edge and Cloud nodes, available things, and the communication links. For both Edge and Cloud nodes, the DB stores information about the software systems and applications that they run (e.g., operating systems and the installed frameworks), the set of things connected to each node (See Figure 1), and the approximate time needed for a node to execute small, medium, and big software functionalities. Additionally, for an Edge node, the DB stores information about its type (e.g., a server or a gateway), hardware resources (i.e., CPU cores, RAM, and storage), and the utilization rates of those resources. The hardware capabilities of the Cloud nodes are assumed to be unbounded as customers can buy more processing power, RAM, and storage [8].

   For each thing, the DB stores information about its type (e.g., light sensor), availability status, location, and battery level (when applicable). Finally, for each communication link, the DB stores information about its latency and download and upload bandwidth.
3) Events: The DB stores the events triggered by the context manager. For each event, it stores information about when it was triggered, its type (e.g., node became unavailable), category (i.e., C1 or C2), and if it was processed by the adaptation manager. Additionally, based on the event type, different metadata are considered. For instance, for the types under the category C1, the related deployment node or communication link are specified.
4) Deployed functionalities: The database stores information about where the software functionalities of GDS are deployed within a hardware infrastructure.
5) Locations: The DB stores information about the known spatial boundaries (e.g., a room, building).
6) GDS: The DB stores information about the formed GDS including the goals they aim to achieve, the goals' spatial boundaries, the identifiers of the things and the functionalities that constitute the GDS, and the identifiers of the used deployment nodes and communication links.

The Goal Ontology contains semantic knowledge about users' goals. It has one class called "Goal" that models the goals' *types* (e.g., adjust light level), and one object property that models the relation "has sub-goal". Moreover, it includes an attribute that specifies the *performance requirements*, which determine the maximum acceptable time for enacting a GDS. More specifically, the specified time is the sum of the time needed to execute the GDS functionalities and the latency of all involved communication links. We chose to represent this component as an ontology because of ontologies' expressiveness in representing shared understanding of knowledge among people, and the availability of several tools that support their usage [29], [35].

B. Process

1) The Process of Deploying GDS: As illustrated in Figure 2, the process[1] starts when a user expresses her/his goal via the user agent. This means specifying at least the goal type (e.g., adjust light level) and spatial boundaries (e.g., a meeting room). At this phase, this is achieved by asking the user to select the goal type and spatial boundaries from predefined lists of goals and locations (we plan to work on a mixed-initiative approach as future work). Then, the goal manager interprets the goal, retrieves from the knowledge base the functionalities and the things available within the spatial boundaries, and forms a GDS that can achieve the goal (if possible).

If the goal is achievable, the goal manager forwards the set of functionalities and related things that constitute the GDS to the deployment planner, which evaluates the complexity of the goal using the metrics presented in Section III-A. If the goal is *primitive*, the minimal hardware requirements of the specified functionalities are considered when generating a deployment topology for the GDS. Instead, the optimal hardware requirements are considered, if the goal is *complex*. After

[1]The processes in Figure 2 and Figure 3 are modeled using the standard Business Process Model and Notation (BPMN) http://www.bpmn.org

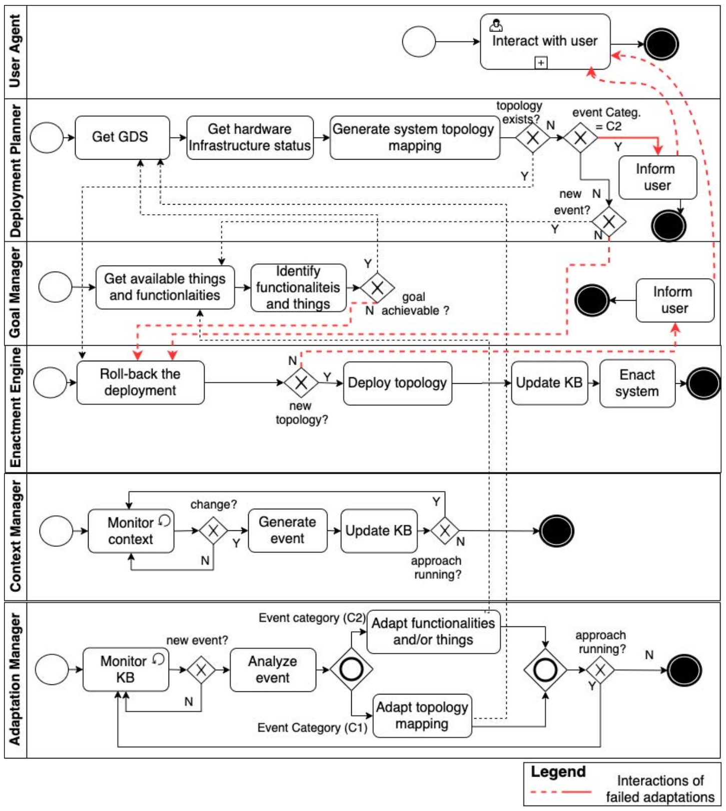

Fig. 3. The deployment adaptation process

that, the deployment planner gets the status of the hardware infrastructure from the knowledge base including the available nodes, the software installed on them, their current hardware utilization rates, and the latency of the communication links among them.

Next, the deployment planner tries to generate a deployment topology that maps the GDS functionalities to the available deployment nodes taking into consideration the performance requirements that the GDS should meet as specified in the goal ontology. For this purpose, it runs a refined version of the algorithm presented in [8]. More specifically, for each functionality, it finds all the compatible deployment nodes that meet the hardware and software requirements of the functionality. A compatible node should also be connected (directly or indirectly) to the set of things related to the functionality.

After that, the deployment planner applies heuristics, based on the available hardware resources (i.e., CPU, RAM, and disk storage) of the candidate nodes and ranks them accordingly. Then, it sorts the list of functionalities in ascending order based on the number of the compatible nodes that can run them. To form the topology, it starts mapping the functionalities that have the least number of candidate nodes as follows. If the requested goal is complex, a functionality is mapped to the compatible node with the highest rank. Instead, if the goal is primitive, the functionality is mapped to the node with the lowest rank.

Then, the deployment planner estimates the total time required to enact the GDS using the selected nodes by summing the time needed for the nodes to run the GDS functionalities and the latency of the involved communication links. The sum is performed considering the order of the functionalities specified by the goal manager (see Section III-A). If the performance requirements specified in the ontology cannot be met and the goal is complex, the user is notified about the expected delay in realizing her/his goal. If the goal is primitive, the deployment planner tries to map the functionalities to nodes that have higher ranks in the list of candidate nodes. Whenever a functionality is re-mapped, the sum is recalculated. If the performance requirements cannot be met, the user is notified about the expected delay. Note that, whenever a functionality is mapped to a node, the deployment planner anticipates the remaining available resources on the node.

If a topology exists, the enactment engine deploys the functionalities of the GDS in the deployment nodes. Also, it updates the knowledge base by deducting the hardware requirements of the functionalities from the consumable hardware resources (i.e., RAM and disk storage) of the deployment nodes. Similarly, it updates the available bandwidth of the communication links specified in the topology. Moreover, it updates the deployed functionalities repository to reflect where each functionality is deployed, and it adds the GDS to the

GDS repository. Finally, it enacts the GDS by executing each functionality in the order specified by the goal manager.

2) The Process of Adapting GDS: As illustrated in Figure 3, the context manager continuously monitors the context, generates events when changes are detected, and stores those events in the knowledge base. The adaptation manager continuously monitors the events in the knowledge base and analyzes them. This means identifying the set of GDS whose enactment has not completed yet and are affected by the changes in the context that are reflected by the events.

After identifying the set of affected GDS, the adaptation manager triggers the proper adaptation process based on the category of the detected event. If the event is of category C1 (e.g., deployment node became unavailable), the adaptation manager tries to generate a new deployment topology for each of the affected GDS, as explained in the deployment process previously. For each GDS, if a new topology is generated, the enactment engine rolls back the deployment of the unused links and/or nodes of the old topology, deploys the new ones, and enacts the GDS.

If the deployment planner cannot generate a deployment topology for a GDS, or if the event is of category C2, the goal manager is requested to form a new GDS that can maintain the achievement of the goal (if possible). If a new GDS is formed, the deployment planner generates a new deployment topology for it (if possible). Then, the enactment engine rolls back the deployment of the unused links and/or nodes of the previous topology, deploys the new one, and enacts the GDS. If the goal manager cannot form a new GDS or the deployment planner cannot generate a deployment topology for the newly formed GDS, the enactment engine rolls back the deployment of the old topology and the user is informed that her/his goal is no longer achievable.

IV. Prototype Implementation

In this section, we present some details about the prototype we implemented and exploited to run simulations to validate the feasibility of the approach. The prototype was implemented in Java.

1) Goal Manager: We extended the ECo-IoT approach [2] to simulate the dynamic formation of concurrent GDS. The goal of a GDS is randomly selected from a list that contains the goals in the knowledge base. The number of functionalities of the GDS is also randomly specified and ranges between two and nine. The generated functionalities were sequentially linked. Additionally, for each functionality, the number of related things is randomly specified and ranges between one and three.

2) Deployment Planner: To implement the *goal complexity evaluator*, we integrated the OWL-API (version 5.1.3)[2] to query the goal ontology. To implement the *topology generator*, we refined and extended parts of the FogTorchΠ simulator to generate deployment topologies for GDS. FogTorchΠ was originally proposed to support IoT designers in making decisions about where to deploy the functionalities of IoT systems. It finds eligible deployment topologies and provides analysis of the resource consumption rate for each topology [8]. Note that, unlike our approach, it considers a static set of things, it is not goal-driven, and does not support the self-adaptation of GDS in response to dynamic changes in their environments.

3) Context Manager: In this first prototype implementation, the context manager was implemented as a couple of continuously running threads. The *context monitor* exposes APIs to receive changes about the context including those at the infrastructure level, and the *reasoning engine* analyzes those changes and generates events that reflect them as explained in Section III-A. None of the existing Java-based simulators supports simulating the events we consider at the level of a hardware infrastructure. Therefore, we implemented a simulator that given a hardware infrastructure and a set of deployment topologies, it randomly generates events of the categories C1 and C2.

4) Enactment Engine: In this first prototype implementation, the *topology deployer* and the *orchestrator* were implemented as Java classes. They simulated the deployment and enactment of GDS.

5) Adaptation Manager: Both the *event analyzer* and the *event handler* were implemented as continuously running Java threads.

6) Knowledge base: : The *goal ontology* was represented using the Ontology Web Language OWL [26]. Other techniques (e.g., Resource Description Framework (RDF)) could also be used to realize the ontology. We chose OWL due to its expressiveness and the high performance of the open-source OWL-API. The DB component was realized as a PostgreSQL relational database. The set of functionalities in the DB were automatically generated. The minimal and optimal hardware requirements of those functionalities were randomly specified as follows. The minimal CPU cores are either 1 or 2, the minimal required available RAM ranges between (0.2 — 0.9 GB), and the minimal required available storage ranges between (0.2 — 1.5 GB). The optimal CPU cores range between (2 — 4), the optimal required available RAM ranges between (1 — 3 GB), and the optimal required available storage ranges between (1.5 — 4 GB). The software requirements of the nodes that can run the generated functionalities were randomly specified from a set that contains 10 software systems and applications (e.g., Java, Linux, Microsoft Office).

V. Validation

The dynamicity of the IoT environment and the involvement of the human in the loop require the approach to be responsive when deploying GDS. To validate the feasibility of our approach, we simulated the deployment of GDS using the implemented prototype. We simulated a smart home and smart building environments, as explained below. The simulations were conducted on a dual-core CPU running at 2.7 GHz, with 16 GB memory.

A. Simulating the Deployment of GDS in a Smart Home

Figure 4 illustrates the hardware infrastructure of the smart home where we simulated the deployment of GDS. The infras-

[2]http://owlapi.sourceforge.net/

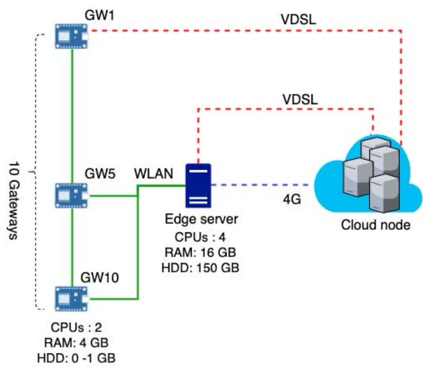

Fig. 4. Smart home infrastructure

tructure comprises 10 gateways that are connected to 40 things, an Edge server, and a Cloud node. For the gateways hardware specifications, we considered the specifications of Raspberry Pi[3] and extended the storage capabilities of some of them to reach 1 GB. Eight Gateways are connected to the Edge server, while two gateways are connected to the Cloud node directly, to simulate devices, such as Alexa[4] and Google Nest hub[5]. The QoS profiles for the communication links in the figure are presented in Table II. Seven software systems and applications that run on each node were randomly specified from the same set that was used to specify the software requirements of the functionalities. Moreover, the time required to run small, medium, and big functionalities were specified for each node based on its hardware capabilities.

TABLE II
THE QOS PROFILES OF COMMUNICATION LINKS [9]

Profile	Latency	Download	Upload
4G	53 ms	22.67 Mbps	16.97 Mbps
VDSL	60 ms	60 Mbps	6 Mbps
WLAN	15 ms	32 Mbps	32 Mbps
Satellite 14M	40 ms	10.5 Mbps	4.5 Mbps
Fiber	5 ms	1000 Mbps	1000 Mbps

TABLE III
GENERATING DEPLOYMENT TOPOLOGIES FOR GDS

Number of concurrent GDS	Total number of functionalities	Total number of things	Average time/GDS (ms)
5	30	41	63
10	48	75	72
15	81	115	83
20	100	156	92

Figure 5 illustrates the average time for generating deployment topologies for GDS by the deployment planner. Table III provides the *total* number of functionalities and things that constitute the simulated GDS. Note that, a thing can be related

[3]https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/

[4]https://aws.amazon.com/iot/solutions/connected-home/iot-and-alexa/

[5]https://store.google.com/se/product/google$_nest_hub$

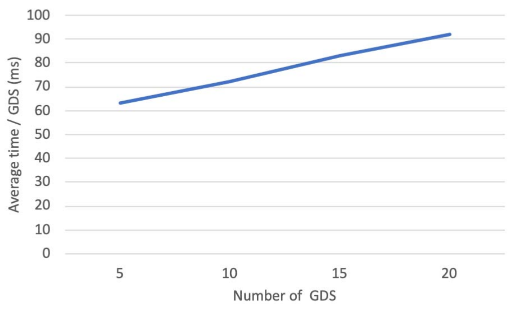

Fig. 5. Generating deployment topologies for GDS in the simulated smart home

to more than one functionality and be among the constituents of multiple GDS. We also simulated the adaptation of the GDS in response to dynamically triggered events. Figure 6 and Figure 7 illustrate the average time for adapting GDS in response to events of the categories C1 and C2, respectively. Table IV and Table V provide information about the number of performed adaptations for each category. As can be noted, the simulations show that deployment topologies were generated and adapted in the scale of milliseconds.

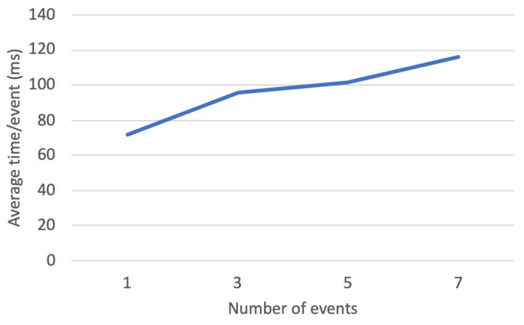

Fig. 6. Adapting GDS in response to event of the category C1 in the simulated smart home

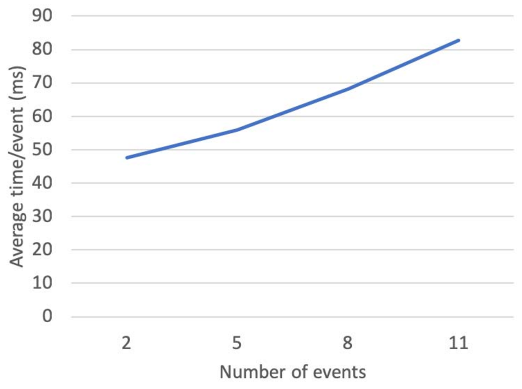

Fig. 7. Adapting GDS in response to event of the category C2 in the simulated smart home

TABLE IV
SIMULATION RESULTS FOR ADAPTING GDS IN RESPONSE TO EVENTS OF CATEGORY C1 IN THE SMART HOME

Number of concurrent GDS	Number of events	Number of performed adaptations	Average time/event (ms)
5	1	2	72
10	3	5	96
15	5	9	102
20	7	11	116

TABLE V
SIMULATION RESULTS FOR ADAPTING GDS IN RESPONSE TO EVENTS OF CATEGORY C2 IN THE SMART HOME

Number of concurrent GDS	Number of events	Number of performed adaptations	Average time/event (ms)
5	2	4	48
10	5	9	56
15	8	14	68
20	11	23	83

B. Simulating the Deployment of GDS in a Smart Building

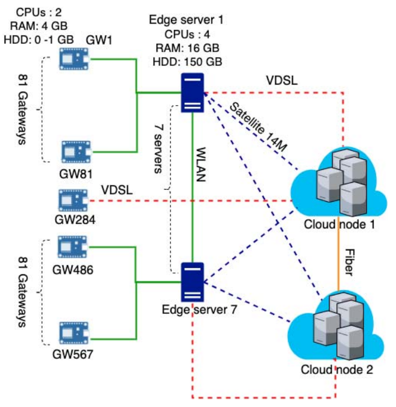

Fig. 8. Smart Building infrastructure

Figure 8 illustrates the hardware infrastructure of the smart building where we simulated the deployment of GDS. The building has 7 floors, each of them has an Edge server. Each Edge server is connected to 81 gateways and each gateway is connected to 4 things. Thus, the total number of included things is 2268. Additionally, each Edge server is connected to two Cloud nodes. The capabilities of the nodes in the infrastructure and the settings of experiments are similar to those presented in Section V-A. The QoS profiles for the communication links in the figure are shown in Table II.

Figure 9 illustrates the average time for evaluating the complexity of the randomly selected goals and *generating deployment topologies* of the dynamically formed GDS to achieve them. Table VI provides the total number of functionalities and things that constitute the simulated GDS.

We also simulated the adaptation of the GDS in response to dynamically triggered events. Figure 10 and Figure 11 illustrate the average time for adapting GDS in response to events of the categories C1 and C2, respectively. Table VII and Table VIII provide more information about the number of performed adaptations in each simulation. As can be noted, the

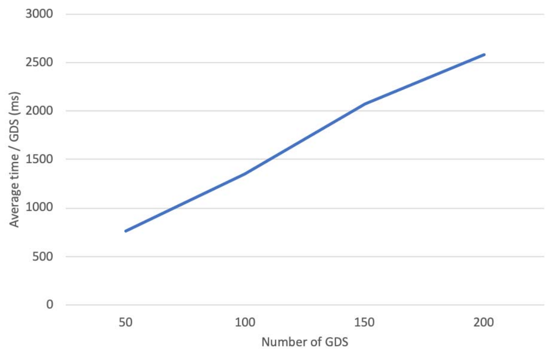

Fig. 9. Generating deployment topologies for GDS in the simulated smart building

TABLE VI
GENERATING DEPLOYMENT TOPOLOGIES FOR GDS IN THE SMART BUILDING

Number of concurrent GDS	Total number of functionalities	Total number of things	Average time / GDS (ms)
50	227	339	767
100	483	691	1350
150	669	1138	2077
200	928	1600	2587

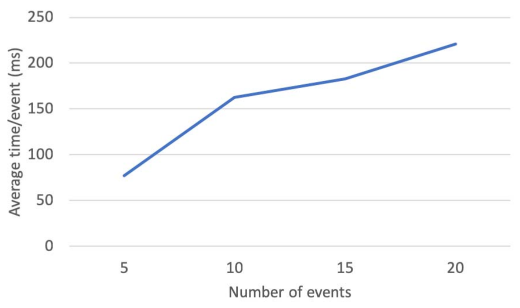

Fig. 10. Adapting GDS in response to event of the category C1 in the simulated smart building

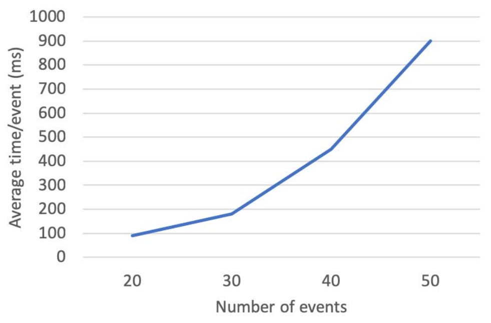

Fig. 11. Adapting GDS in response to event of the category C2 in the simulated smart building

TABLE VII
SIMULATION RESULTS FOR ADAPTING GDS IN RESPONSE TO EVENTS OF CATEGORY C1 IN THE SMART BUILDING

Number of Concurrent GDS	Number of events	Number of performed adaptations	Average time/event (ms)
50	5	8	77
100	10	15	162
150	15	30	182
200	20	63	220

TABLE VIII
SIMULATION RESULTS FOR ADAPTING GDS IN RESPONSE TO EVENTS OF CATEGORY C2 IN THE SMART BUILDING

Number of Concurrent GDS	Number of events	Number of performed adaptations	Average time/event (ms)
50	20	13	90
100	30	22	180
150	40	32	450
200	50	74	900

simulations show that the approach can generate deployment topologies for 200 GDS, which comprise in total 928 functionalities and 1600 things, in an average time of 2.5 seconds. Additionally, the simulations show that the approach enables the adaptation of GDS in response to events of category C1 in an average time of less than 250 milliseconds, when 20 events are triggered concurrently. Moreover, the expirements reveal that the approach enables the adaptation of GDS in response to events of category C2 in an average time of 900 milliseconds, when 50 events are triggered concurrently.

VI. DISCUSSION

Compared to the approaches discussed in Section II, our approach enables the dynamic deployment of GDS in the Edge-Cloud continuum considering the complexity of users' goals and the performance requirements. Additionally, it supports the automated adaptation of those systems in response to dynamic changes in their deployment topologies and the status of their constituents. However, our approach does not guarantee that the generated topologies are the most optimal ones with respect to resource consumption rates and performance. GDS are realized within well-defined spatial boundaries (e.g., building, room). Consequently, we do not expect the number of the things available in those boundaries to be massive. Under the assumptions that the leveraged components of the FogTorchΠ simulator and the other components in our approach meet their requirements, our approach generates correct topologies and scales well at the level of smart homes and (big) smart buildings, as shown in Section V.

In this paper, we assumed that the software components of the proposed approach are deployed offline. At the smart city scale, several instances of the approach would be needed to be deployed. Those instances should be able to automatically scale up to meet the performance requirements or down to optimize the usage and the cost of using the available resources. Multiple requirements can influence the decisions of where to deploy GDS software functionalities within hardware infrastructures. For instance, some resources at the Edge (e.g., smartphones) have limited energy resources. Moreover, some users might be unwilling to process their private data in the Cloud or to pay for the cost of using it. Furthermore, goals might have different priorities. Thus, the GDS that aim to achieve the goals with the highest priorities should be realized before those having less priorities. We plan to address these aspects in our future work.

Finally, our approach requires developers to construct the goal ontology, specify the software requirements, and minimal and optimal hardware requirements of available functionalities. Moreover, they also need to model resources within available hardware infrastructures. In our future work, we plan to evolve the approach and enable it to automatically detect and monitor the available resources in the Edge-Cloud continuum.

VII. CONCLUSION AND FUTURE WORK

In this paper, we proposed a goal-driven approach that enables the automated deployment of GDS in the Edge-Cloud continuum and the adaptation of those systems in response to dynamic changes in their deployment topologies and the status of their constituents. To validate the feasibility of our approach, we simulated the generation and adaptation of deployment topologies of GDS in a smart home and a smart building. The results of the simulations reveal that the approach scaled well in both cases.

In our future work, we plan to perform a more extensive evaluation of the approach. Also, we will investigate how to enable the proposed approach to scale to support the realization of GDS in a large scale IoT environment (e.g., smart city). The approach can also be extended in other directions as described in the following. To generate more reliable deployment topologies for GDS, the approach can be extended to consider the energy consumption of battery powered Edge nodes. Other QoS attributes and constraints that are also relevant include privacy, security, and cost. Also, the approach can be evolved to automatically evaluate the trade-offs among the QoS attributes and generate deployment topologies accordingly. Moreover, it can be extended to enable the deployment of GDS formed to achieve goals with different priorities.

ACKNOWLEDGMENT

This work is partially financed by the Knowledge Foundation through the Internet of Things and People research profile (Malmö University, Sweden) and by the Research Cluster "Smart Communities and Technologies (Smart CT)" at TU Vienna.

REFERENCES

[1] Fahed Alkhabbas, Romina Spalazzese, and Paul Davidsson. Architecting Emergent Configurations in the Internet of Things. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 221–224. IEEE, 2017.

[2] Fahed Alkhabbas, Romina Spalazzese, and Paul Davidsson. ECo-IoT: An architectural Approach for Realizing Emergent Configurations in the Internet of Things. In *European Conference on Software Architecture*, pages 86–102. Springer, 2018.

[3] Majid Ashouri, Paul Davidsson, and Romina Spalazzese. Cloud, Edge, or Both? Towards Decision Support for Designing IoT Applications. In *2018 Fifth International Conference on Internet of Things: Systems, Management and Security*, pages 155–162. IEEE, 2018.

[4] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Computer Networks*, 54(15):2787–2805, 2010.

[5] Cosmin Avasalcai and Schahram Dustdar. Latency-Aware Distributed Resource Provisioning for Deploying IoT Applications at the Edge of the Network. In *Future of Information and Communication Conference*, pages 377–391. Springer, 2019.

[6] Antonio Brogi, Stefano Forti, and Marco Gaglianese. Measuring the Fog, Gently. In *International Conference on Service-Oriented Computing*, pages 523–538. Springer, 2019.

[7] Antonio Brogi, Stefano Forti, Carlos Guerrero, and Isaac Lera. How to Place Your Apps in the Fog-State of the Art and Open Challenges. *arXiv preprint arXiv:1901.05717*, 2019.

[8] Antonio Brogi, Stefano Forti, and Ahmad Ibrahim. How to Best Deploy your Fog Applications, Probably. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 105–114. IEEE, 2017.

[9] Antonio Brogi, Stefano Forti, and Ahmad Ibrahim. Deploying fog applications: How much does it cost, by the way? In *CLOSER*, pages 68–77, 2018.

[10] Federico Ciccozzi and Romina Spalazzese. MDE4IoT: Supporting the Internet of Things with Model-driven Engineering. In *International Symposium on Intelligent and Distributed Computing*, pages 67–76. Springer, 2016.

[11] Simone Cirani, Gianluigi Ferrari, Nicola Iotti, and Marco Picone. The IoT Hub: A Fog Node for Seamless Management of Heterogeneous Connected Smart Objects. In *2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking-Workshops (SECON Workshops)*, pages 1–6. IEEE, 2015.

[12] Martina De Sanctis, Romina Spalazzese, and Catia Trubiani. QoS-Based Formation of Software Architectures in the Internet of Things. In *European Conference on Software Architecture*, pages 178–194. Springer, 2019.

[13] Schahram Dustdar, Cosmin Avasalcai, and Ilir Murturi. Edge and Fog Computing: Vision and Research Challenges. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 96–9609. IEEE, 2019.

[14] Patrícia Espada, Miguel Goulão, and João Araújo. A Framework to Evaluate Complexity and Completeness of KAOS Goal Models. In *International Conference on Advanced Information Systems Engineering*, pages 562–577. Springer, 2013.

[15] Sonja Filiposka, Anastas Mishev, and Katja Gilly. Mobile-aware Dynamic Resource Management for Edge Computing. *Transactions on Emerging Telecommunications Technologies*, page e3626.

[16] Daniel Giusto, Antonio Iera, Giacomo Morabito, and Luigi Atzori. *The Internet of Things: 20th Tyrrhenian Workshop on Digital Communications*. Springer Science & Business Media, 2010.

[17] Catarina Gralha, João Araújo, and Miguel Goulão. Metrics for Measuring Complexity and Completeness for Social Goal Models. *Information Systems*, 53:346–362, 2015.

[18] Lin Gu, Deze Zeng, Song Guo, Ahmed Barnawi, and Yong Xiang. Cost Efficient Resource Management in Fog Computing Supported Medical Cyber-Physical System. *IEEE Transactions on Emerging Topics in Computing*, 5(1):108–119, 2015.

[19] Gabriel Guerrero-Contreras, Jose Luis Garrido, Sara Balderas-Diaz, and Carlos Rodríguez-Domínguez. A Context-aware Architecture Supporting Service Availability in Mobile Cloud Computing. *IEEE Transactions on Services Computing*, 10(6):956–968, 2016.

[20] Mohammed A Hassan, Mengbai Xiao, Qi Wei, and Songqing Chen. Help your Mobile Applications with Fog Computing. In *2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking-Workshops (SECON Workshops)*, pages 1–6. IEEE, 2015.

[21] Fatemeh Jalali, Kerry Hinton, Robert Ayre, Tansu Alpcan, and Rodney S Tucker. Fog Computing May Help to Save Energy in Cloud Computing. *IEEE Journal on Selected Areas in Communications*, 34(5):1728–1739, 2016.

[22] Jeffrey O Kephart and David M Chess. The Vision of Autonomic Computing. *Computer*, (1):41–50, 2003.

[23] In-Young Ko, Han-Gyu Ko, Angel Jimenez Molina, and Jung-Hyun Kwon. SoIoT: Toward a User-Centric IoT-based Service Framework. *ACM Transactions on Internet Technology (TOIT)*, 16(2):8, 2016.

[24] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. Fog Computing: A Taxonomy, Survey and Future Directions. In *Internet of everything*, pages 103–130. Springer, 2018.

[25] Simon Mayer, Ruben Verborgh, Matthias Kovatsch, and Friedemann Mattern. Smart Configuration of Smart Environments. *IEEE Transactions on Automation Science and Engineering*, 13(3):1247–1255, 2016.

[26] Deborah L McGuinness, Frank Van Harmelen, et al. OWL Web Ontology Language Overview. *W3C recommendation*, 10(10):2004, 2004.

[27] Ilir Murturi, Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. Edge-to-Edge Resource Discovery using Metadata Replication. In *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–6. IEEE, 2019.

[28] Phu Nguyen, Nicolas Ferry, Gencer Erdogan, Hui Song, Stéphane Lavirotte, Jean-Yves Tigli, and Arnor Solberg. Advances in Deployment and Orchestration Approaches for IoT-a Systematic Review. In *2019 IEEE International Congress on Internet of Things (ICIOT)*, pages 53–60. IEEE, 2019.

[29] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context Aware Computing for the Internet of Things: A Survey. *IEEE communications surveys & tutorials*, 16(1):414–454, 2013.

[30] M Reza Rahimi, Jian Ren, Chi Harold Liu, Athanasios V Vasilakos, and Nalini Venkatasubramanian. Mobile Cloud Computing: A Survey, State of Art and Future Directions. *Mobile Networks and Applications*, 19(2):133–143, 2014.

[31] Yuvraj Sahni, Jiannong Cao, Shigeng Zhang, and Lei Yang. Edge Mesh: A New Paradigm to Enable Distributed Intelligence in Internet of Things. *IEEE access*, 5:16441–16458, 2017.

[32] Heng Shi, Nan Chen, and Ralph Deters. Combining Mobile and Fog Computing: Using COAP to link Mobile Device Clouds with Fog Computing. In *2015 IEEE International Conference on Data Science and Data Intensive Systems*, pages 564–571. IEEE, 2015.

[33] Olena Skarlat, Matteo Nardelli, Stefan Schulte, and Schahram Dustdar. Towards QoS-Aware Fog Service Placement. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 89–96. IEEE, 2017.

[34] Christos Tsigkanos, Ilir Murturi, and Schahram Dustdar. Dependable Resource Coordination on the Edge at Runtime. *Proceedings of the IEEE*, 2019.

[35] Xiaohang Wang, Daqing Zhang, Tao Gu, Hung Keng Pung, et al. Ontology Based Context Modeling and Reasoning using OWL. In *Percom workshops*, volume 18, page 22. Citeseer, 2004.

[36] Deze Zeng, Lin Gu, Song Guo, Zixue Cheng, and Shui Yu. Joint Optimization of Task Scheduling and Image Placement in Fog Computing Supported Software-defined Embedded System. *IEEE Transactions on Computers*, 65(12):3702–3712, 2016.