Towards a Model-Based Serverless Platform for the Cloud-Edge-IoT Continuum

Nicolas Ferry, Rustem Dautov, Hui Song

Towards a Model-Based Serverless Platform for the Cloud-Edge-IoT Continuum

Nicolas Ferry
Université Côte d'Azur
I3S/INRIA Kairos
Sophia Antipolis, France
nicolas.ferry@univ-cotedazur.fr

Rustem Dautov
SINTEF Digital
Oslo, Norway
rustem.dautov@sintef.no

Hui Song
SINTEF Digital
Oslo, Norway
hui.song@sintef.no

***Abstract*—One of the most prominent implementations of the serverless programming model is Function-as-a-Service (FaaS). Using FaaS, application developers provide source code of serverless functions, typically describing only parts of a larger application, and define triggers for executing these functions on infrastructure components managed by the FaaS provider. There are still challenges that hinder the wider adoption of the FaaS model across the whole Cloud-Edge-IoT continuum. These include the high heterogeneity of the Edge and IoT infrastructure, vendor lock-in, the need to deploy and adapt serverless functions as well as their supporting services and software stacks into their cyber-physical execution environment. As a first step towards addressing these challenges, we introduce the SERVERLESS4IOT platform for the design, deployment, and maintenance of applications over the Cloud-Edge-IoT continuum. In particular, our platform enables the specification and deployment of serverless functions on Cloud and Edge resources, as well as the deployment of their supporting services and software stacks over the whole Cloud-Edge-IoT continuum.**

***Index Terms*—Internet of Things, Edge Computing, Cloud Computing, Model-Driven Engineering, Deployment**

I. INTRODUCTION

A recent forecast from the International Data Corporation (IDC) envisions over 50 billion Internet-of-Things (IoT) endpoints to be in use by 2025 [1]. These leaf devices, as well as edge gateways are now becoming part of a larger Cloud-Edge-IoT computing continuum, where processing and storage tasks are distributed across the whole network hierarchy, not only centralized in the Cloud [2]. The next generation of IoT systems will thus comprise software modules massively distributed on both Cloud resources and devices at the edge of the network (*e.g.*, gateways, routers, switches, base stations). Developing and managing software across such a complex and heterogeneous computing continuum is becoming a key challenge calling for a novel programming model.

Serverless computing has recently gained popularity as a suitable programming model for IoT systems. It allows IT specialists to focus on developing software, while shifting the responsibility of managing the underlying infrastructure to the Cloud provider. Serverless functions are simple operations expressing parts of the application logic, which are triggered on the occurrence of specific events. This event-driven nature, especially when implemented through Function-as-a-Service (FaaS) platforms, is a natural fit for IoT event and data processing [3], [4]. Since serverless functions are typically stateless and often need to be deployed together with supporting services (*e.g.*, message brokers and data stores), software systems are rarely built entirely from serverless components, but rather follow a hybrid approach where functions are combined with other types of software components.

However, there are still challenges hindering the wider adoption of FaaS solutions when developing systems for the whole Cloud-Edge-IoT continuum. First, in contrast to relatively homogeneous Cloud infrastructures, an Edge infrastructure consists of distributed and heterogeneous devices, which have different cyber-physical contexts in terms of hardware capacity, CPU architecture, network connection, user preferences, etc. As a result, serverless functions, and in particular the runtime stacks on top of which they are deployed, typically need to be tailored to this cyber-physical context. Second, in most cases, FaaS providers promote tight coupling between the serverless functions and their services. Given that FaaS solutions are typically heterogeneous and the provided features are often incompatible across providers, vendor lock-in is thereby promoted [4]. Third, the supporting services typically need to be deployed in the vicinity of the function, close to the data source (*e.g.*, on the same edge device), and also tailored to the serverless function cyber-physical context.

This paper presents SERVERLESS4IOT as a first step towards addressing the aforementioned challenges. SERVERLESS4IOT is a platform for designing, deploying and maintaining applications over the Cloud-Edge-IoT continuum, composed of two main components. First, a domain-specific modelling language provides a unified abstraction for specifying deployment of IoT applications, including serverless functions. This also covers specifying the software stack and supporting services to be deployed with each function. Following the serverless principles, the latter is needed only when none of the predefined templates fit the serverless function requirements. Second, an execution platform serves to enact deployments written in the SERVERLESS4IOT modelling language, which seamlessly integrates *(i)* a FaaS platform for deployment and operation of serverless functions on Cloud and Edge resources, and *(ii)* a generic solution for the deployment of software components on Cloud, Edge and IoT resources. Accordingly, the main contributions of the paper are:

– A **modelling language** to define deployment of hybrid IoT systems that may involve serverless functions.
– A **platform** to enable FaaS on Edge resources as well as the integration of serverless functions as part of a larger IoT system. This includes the deployment of supporting services at the edge as well as software components on IoT devices.
– A **template mechanism** to either fully specify and customize the software stack on top of which functions are deployed or simply reuse off-the-shelf stacks.
– **Demonstration of the proposed approach** through a running Smart Building use case.

The rest of this paper is organised as follows. Section II describes an example that motivates requirements for our proposed approach. Section III introduces the overall SERVERLESS4IOT approach and architecture before Sections IV and V detail the corresponding modelling language and its execution engine, respectively. Section VI discusses related works and Section VII concludes the paper and outlines future work.

II. MOTIVATING EXAMPLE: SMART BUILDING

The serverless model is a natural fit for hierarchical systems, wherein heterogeneous and potentially resource-constrained computing nodes are spread across the Cloud-Edge-IoT continuum, while application requirements tend to dynamically change. A representative example of such cyber-physical systems is the smart building described in this section, which will act as a motivating example. This example is a part of a larger smart building case study successfully implemented in the frame of the H2020 ENACT project [2].

Fig. 1 depicts the overall deployment model of the case study using the graphical syntax of the SERVERLESS4IOT modelling language. The smart building is composed of two systems (separated by the large dotted line in Figure 1). The first system, IoT Smart Space, includes a gateway RPI-2 that connects to *(i)* a wireless sensor/actuator network via the Z-Wave protocol, *(ii)* smart devices (*i.e.*, Arduino-1 to control roller shutters or Smart TV) using WiFi, and *(iii)* other gateways with sensors distributed in the building for monitoring sound and human presence. The second system, App Server, hosts applications that receive data from the sensors and send commands to the actuators available in the other system. App Server interacts with IoT Smart Space and Building Control using a Cloud-based instance of the SMOOL IoT middleware [5], which includes a semantic broker for connecting heterogeneous devices or data sources.

There are two applications deployed on App Server. The UserComfort application accesses sensor data to make decisions for user comfort and sends commands to control the actuators, *e.g.*, roller shutters. In particular, it controls luminosity level trying to maximize the exploitation of daylight, regulates the in-door temperature, and enables basic management of multimedia devices. The second application, CommunicationCenter, maintains an acceptable ambient acoustic level, including when emitting and receiving phone or conference calls. A set of software components are deployed in IoT Smart Space, enabling both applications to achieve their goals by interacting with sensors and actuators. While most of these are regular software components, three of them are implemented as serverless functions (*cf.* components with yellow borders in Fig. 1): *(i)* the Presence function interacts with a Bluetooth dongle to regularly scan for present Bluetooth devices, *(ii)* the Arduino_flow function marshals the MQTT input and output messages into messages that can be understood by the Arduino, and *(iii)* the IoTFlow function offers similar behaviour for the smart devices. These functions are implemented either in Node.js or using the Node-RED IoT platform[1]. Taken together, this example motivates the following requirements:

– $\boldsymbol{R_1}$: **Cloud-Edge-IoT automated deployment**. SERVERLESS4IOT should support deployment of software, including serverless functions, on Cloud, Edge and IoT resources.
– $\boldsymbol{R_2}$: **Serverless functions integration**. SERVERLESS4IOT should support specification and deployment of IoT systems that seamlessly integrate regular software components and serverless functions.
– $\boldsymbol{R_3}$: **Edge-based distribution**. SERVERLESS4IOT should support deployment of supporting services in the vicinity of serverless functions. Also, serverless functions residing in the Edge should be resilient to loss of connectivity with the Cloud.
– $\boldsymbol{R_4}$: **Customizable software stack**. SERVERLESS4IOT should support management of a serverless function's software stack at two levels. First, following the serverless principle, it should enable deployment of serverless functions without the need for manually specifying its software stack, but rather by reusing existing templates. Second, when required, it should support specification of such software stacks.
– $\boldsymbol{R_5}$: **Provider independence**. SERVERLESS4IOT should support provider-agnostic deployment specification, while serverless functions should not be tightly coupled with provider-specific supporting services.

$\boldsymbol{R_1}$ and $\boldsymbol{R_2}$ are motivated by the need of deploying both regular software components and functions on Cloud (*e.g.*, SMOOL), Edge (*e.g.*, the Presence function), and IoT (*e.g.*, Arduino boards) resources as part of the overall smart building system. $\boldsymbol{R_3}$ stems from the need for deploying supporting services such as brokers and data storage close to functions in IoT Smart Space. Communication within this system should remain available even if connection with the Cloud and with the deployment solution is lost. Several serverless functions are deployed on top of the same software stack (*e.g.*, Node-RED for easy programming), while the others might use a different one (*e.g.*, J2EE stack), thereby justifying $\boldsymbol{R_4}$. Finally, it should be possible to replicate the system demonstrated in this use case to any other building, thus underpinning $\boldsymbol{R_5}$.

III. THE SERVERLESS4IOT APPROACH

A. Overall approach

Fig. 2 depicts a high-level view of the SERVERLESS4IOT platform, wherein the Orchestrator is the core component. It allocates serverless functions and software components on Cloud, Edge or IoT devices and triggers their deployment. It

[1]https://github.com/node-red/node-red

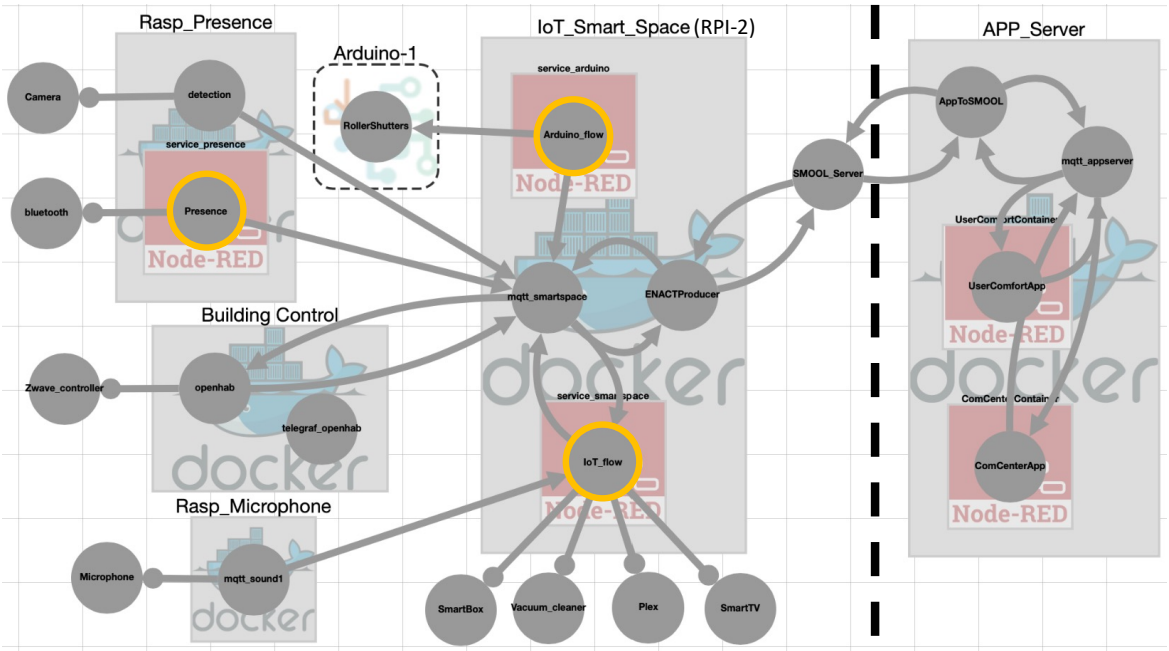

Figure 1. Deployment model of the Smart Building motivating example.

is also the main entry point for end users, *i.e.*, developers, who can interact via a GUI or an IDE. At runtime, this component maintains a list of available devices and in particular of the SERVERLESS4IOT Gateways.

SERVERLESS4IOT gateways are Cloud and Edge devices that host serverless functions. They are typically connected to a set of devices that together form a local subsystem (depicted as dashed circles in Fig. 2). They are equipped with a deployment engine (initially deployed on each gateway by the Orchestrator) to enact the deployment of serverless functions and other software components (including functions' runtime software stack and supporting services) in the sub-system. Placing the deployment engine on the gateway is not common in FaaS platforms and the rationale behind this choice is twofold: *(i)* this way, the deployment engine gets access to devices which might not be accessible from the Cloud (*e.g.*, devices accessible only from within a local area network (LAN) for security reasons or devices indirectly connected to the Internet via the gateway), and *(ii)* it improves resilience of the system as it is still possible to enact or adapt a deployment locally even if the connection to the Orchestrator is lost. This design decision contributes to addressing $\boldsymbol{R_1}$ and $\boldsymbol{R_3}$.

The communication between different subsystems is ensured by a global broker (or a cluster of brokers), while the communication within a sub-system is made possible by a local broker executing on SERVERLESS4IOT gateways. Similarly, the rationale for this choice is twofold: *(i)* to improve the overall resilience of the system (*i.e.*, communication within a subsystem is maintained even if the global broker is not reachable), and *(ii)* to prevent messages exchanged locally within a subsystem from going all the way up to the global broker. This design decision also contributes to addressing $\boldsymbol{R_3}$.

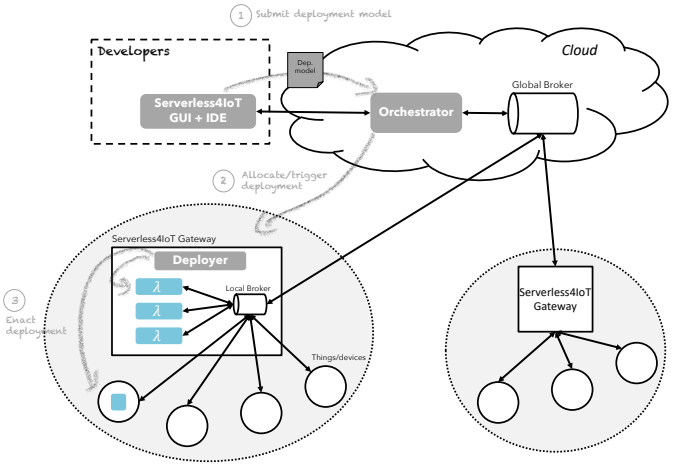

Figure 2. High-level architecture of the platform.

At a high level, the overall deployment process can be summarized as follows. Once deployable artefacts (*e.g.*, functions, microservices, services) are implemented, developers will first use the SERVERLESS4IOT GUI and IDE to specify a corresponding deployment model. In its simplest form, the deployment model includes a serverless function associated with a template of its runtime software stack (this template can be off-the-shelf, which contributes to addressing $\boldsymbol{R_4}$). This deployment model is then pushed to the Orchestrator, which distributes the software artefacts to the different gateways which in turn enact the deployment locally.

B. The Serverless4IoT Gateway

As depicted in Fig. 3, SERVERLESS4IOT gateways are equipped with the following technical services.

– **Proxies to interact with IoT devices** are software components that facilitate interaction between functions and IoT devices. A proxy typically reads inputs from one or several devices, transforms these inputs if necessary, and publishes the transformed data to a local broker. At present, proxies need to be manually selected from a predefined list to match a specific device that they represent. In the future, thanks to this architecture, we aim at developing a mechanism to dynamically generate proxies.
– **Local broker** is responsible for communication within a local subsystem, as explained above. Current implementation of SERVERLESS4IOT leverages a Mosquitto MQTT broker.[2]
– **Bridge to global broker** links local and global brokers, also restructuring transferred data if necessary (*e.g.*, transforming data in NGSI format when using the global Orion context broker[3]). It is also responsible for registering the functions to their input data stream. This means: *(i)* subscribing to the serverless function's trigger events from the global broker, and *(ii)* forwarding these events to the local broker on a dedicated channel. Finally, the bridge also registers each gateway in the SERVERLESS4IOT platform. Upon start, a registration request is emitted to the platform, possibly including meta-information about the gateway (*e.g.*, its physical location).
– **Deployer** is an instance of GeneSIS [6] – a tool to support the orchestration and deployment of IoT software systems across the Cloud-Edge-IoT continuum. GeneSIS includes: *(i)* a domain-specific modelling language to specify the deployment of IoT systems on Cloud, Edge and IoT infrastructures, including devices with no or limited access to the Internet, and *(ii)* a *models@run.time* deployment engine responsible for orchestration, deployment, and adaptation.

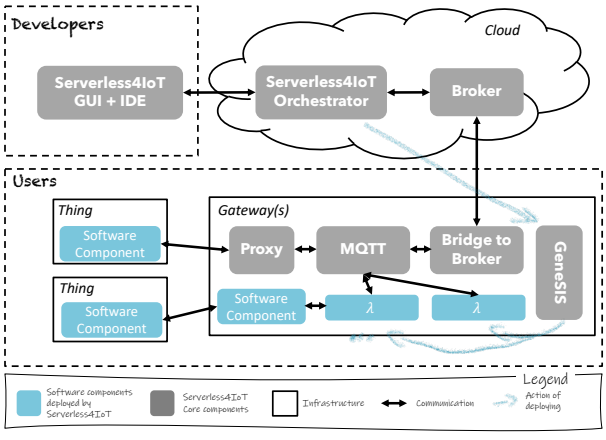

Figure 3. SERVERLESS4IOT detailed architecture.

[2]https://mosquitto.org/
[3]https://fiware-orion.readthedocs.io/

IV. THE SERVERLESS4IOT MODELLING LANGUAGE

To deploy an application, implementations of each software component, *i.e.*, deployable artefacts [7], need to be allocated to target host infrastructures. Where and how these deployable artefacts are allocated is specified in a deployment model defined using the SERVERLESS4IOT modelling language. The language is built on top and extends the GeneSIS modelling language [6] used for specifying deployment models of software systems over Cloud, Edge and IoT infrastructures in a provider-agnostic way (addressing $\boldsymbol{R_5}$). It does not currently support deployment of serverless functions, and the SERVERLESS4IOT modelling language addresses this issue with two extensions: *(i)* added the missing concepts for specifying deployment of serverless functions (addressing R_2), and *(ii)* a template mechanism, which transforms deployment models into re-usable templates, instantiated and refined at deployment time in order to specify the runtime software stack for a serverless function (addressing $\boldsymbol{R_4}$). Below, we will first brief the reader on the main concepts of the GeneSIS modelling language before going into details of these two extensions.

A. Baseline: GeneSIS modelling language

Main concepts in GeneSIS and the relations between them are depicted by the yellow classes of the meta-model in Fig. 4. The complete description of the meta-model can be found in [6]. A GeneSIS deployment model is composed of GeneSISElements, each associated with a set of properties in the form of key-value pairs. A GeneSIS deployment model can be seen as an assembly of components, and thus the two main types of GeneSISElements are Components and Links.

A Component represents a reusable type of node that will compose a DeploymentModel. A Component can be a SoftwareComponent representing a piece of software to be deployed on a host. A SoftwareComponent can be an InternalComponent meaning that it is managed by GeneSIS, or an ExternalComponent meaning that it is either managed by an external provider or hosted on a black box device. A SoftwareComponent can be associated with Resources (*e.g.*, scripts, configuration files) for managing its deployment life-cycle (*i.e.*, download, configure, install, start, and stop).

An InfrastructureComponent provides hosting facilities (*i.e.*, an execution environment) to SoftwareComponents. The property needDeployer indicates that a local connection is required to deploy a SoftwareComponent on an InfrastructureComponent via a PhysicalPort (*e.g.*, an Arduino that can only be accessed locally via a serial port). This property is typically used for devices with no direct access to the Internet, only reachable indirectly via other devices configured by GeneSIS.

The two types of Links are Hostings and Communications. A Hosting depicts that an InternalComponent will execute on a specific host, which can be any component, meaning that it is possible to describe the whole software stack required to run an InternalComponent. A Communication represents a binding between two SoftwareComponents. Finally, the property isDeployer specifies that an InternalComponent

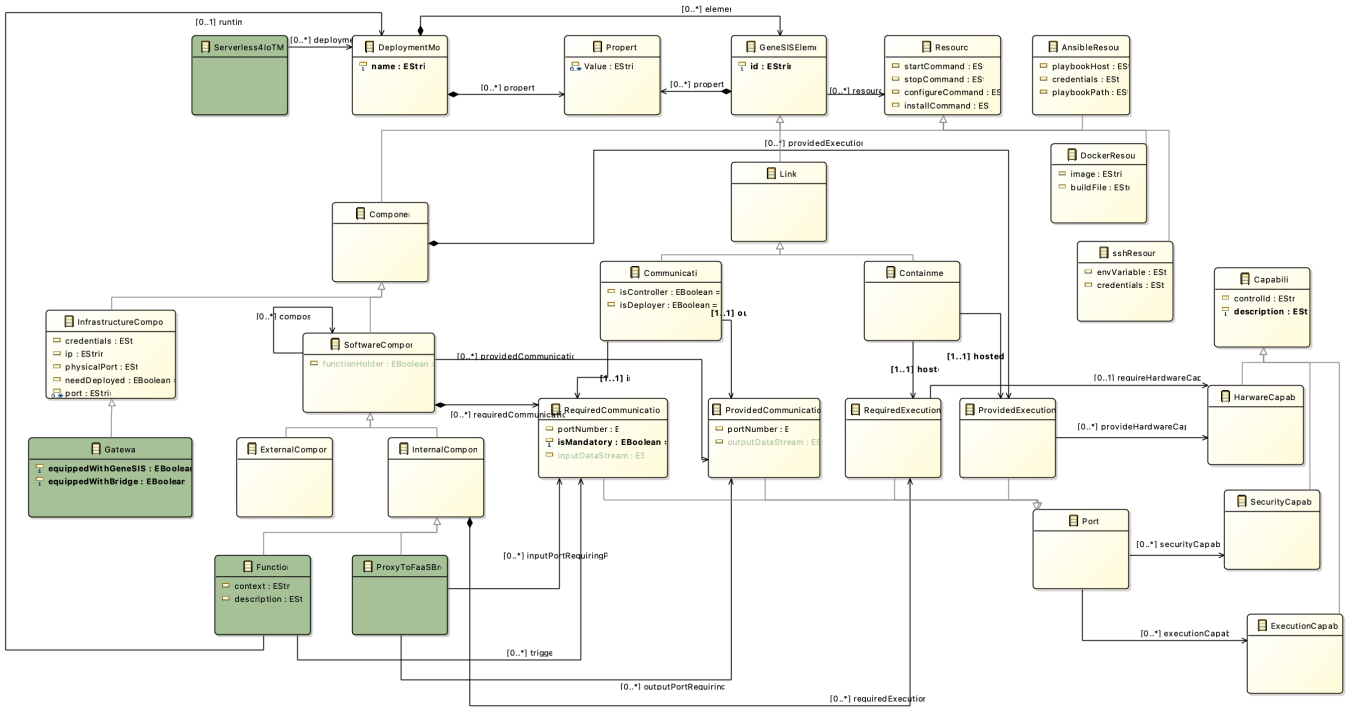

Figure 4. The SERVERLESS4IOT modelling language meta-model.

(one of the endpoints of a Communication) hosted on an InfrastructureComponent with the needDeployer property should be deployed from the host of the other SoftwareComponent (the other endpoint of a Communication).

B. Extension 1: Modelling deployment of serverless functions

A Serverless4IoTModel contains one or several DeploymentModels, which may include Functions. A Function is a specialization of an InternalComponent. As any other Component, a Function has a unique ID and may be described using the description property. The property context represent the context in which the Function should be deployed (*e.g.*, in a specific location). Similarly, the minimum requirements (*e.g.*, GPU hardware requirements) for a Function to execute can be defined using its RequiredExecutionPorts. A Function can refer to a DeploymentModel specifying a runtime software stack, as well as services needed for its execution and how to deploy them (*cf.* Section IV-C for more details).

Serverless functions typically consume inputs and publish outputs via a broker, and are invoked when certain events or messages are received. Each RequiredCommunicationPort of a Function may contain inputDatastream and outputDatastream properties which respectively represent input and output data for a Function. As opposed to other InternalComponents, this property is important for a Function, as it can be used to automatically trigger the subscription of the function to the broker. The trigger property is used to maintain a list of inputDatastreams that may trigger the Function.

ProxyToFaaSBridge is another specialization of InternalComponent representing the Proxy objects. A Gateway is a specialization of InfrastructureComponent. It represents an Edge device, typically a gateway, equipped with the SERVERLESS4IOT facilities for managing Functions (*i.e.*, the gateways as defined in Fig. 3). When true, the property equippedWithGeneSIS indicates that an instance of GeneSIS is up and running on the device. Similarly, the property equippedWithBridge indicates the presence of local brokers and mechanisms to interact with the global broker.

C. Extension 2: Deployment templates

The objective of this extension is to enable the use of a deployment model as a deployment template where some of the software components are not concretely instantiated but rather act as placeholders to be refined with concrete instances of software components at deployment time. Following $\boldsymbol{R_4}$, this allows developers to deploy serverless functions only by specifying the source code location together with a reference to a deployment template (available either off the shelf or defined manually) that describes the function runtime software stack and its supporting services (see Listing 1 for the deployment of the Presence function from our motivating example).

Listing 1. Deployment of the Presence function with the runtime stack.

```
{
    id: "presence",
    triggers: ["/presence/presenceIndicators"],
    runtime: "deployment_presence.json",
    src: "./presence-function"
}
```

More concretely, a deployment template is a DeploymentModel that includes InternalComponents with the placeholder property set to true. At present, there can only be placeholders for InternalComponents, because the SERVERLESS4IOT execution engine only supports the refinement of this type of components. In the future, the placeholder property might be attached to any Component thus providing more flexibility. In particular, enabling the refinement of InfrastructureComponents will be useful for creating deployment templates applicable across different infrastructures and handling dynamic discovery. The deployment of an InternalComponent replacing a placeholder can be specified either in the template (as a Resource attached to the placeholder component) or in a Resource attached to the component.

V. FUNCTION DEPLOYMENT AND ORCHESTRATION

As discussed above, deploying a serverless function is not exactly the same as deploying classical software components. It is important to *(i)* keep track of the serverless functions under operation, and *(ii)* prepare the platform so that a newly deployed function can start interacting with the rest of the system (typically via a broker), including leaf nodes with no direct Internet connection. Below, we discuss how serverless functions are deployed using SERVERLESS4IOT and how software can be deployed on IoT nodes (*i.e.*, leaf nodes) for future interaction with the functions.

A. Serverless function deployment

In FaaS platforms, to deploy a function, developers typically first need to manually create an image of a container with the function code and its runtime software stack. The platform is then responsible for deploying this container. By contrast, in SERVERLESS4IOT, this process can be automated. A deployment template may include necessary instructions for packaging and deploying a function and its runtime software stack (*e.g.*, building and deploying a container). Accordingly, to deploy a function, SERVERLESS4IOT proceeds as follows:

1) First, the SERVERLESS4IOT orchestrator packages the function as defined in the deployment model (*i.e.*, in Resource attached to the Component placeholder). For instance, for Node.js functions, an archive with the source code and required Node.js modules is created, while for Node-RED functions it will include the flow and its Node.js dependencies.
2) The function is then registered in the platform and remains marked as under deployment until eventually deployed.
3) Following the architecture presented in Section 3, when a function is deployed, it registers with both its local and the global brokers. Interaction between the global broker and the function is ensured by the BridgeToBroker component. Next, the SERVERLESS4IOT Orchestrator instructs BridgeToBroker on the target SERVERLESS4IOT gateway to subscribe to the serverless function's input streams from the global broker. At this point, BridgeToBroker already starts forwarding the incoming data from the global broker to the local broker using function names as topics (*e.g.*, `/fc1/in/water`). As a result, a function can be as simple as Listing 2.

Listing 2. Simple function template in Node.js.

```
const mqtt = require("mqtt");
// Name of the function
const fc_name = "/fc1";

var clientMQTT =
    mqtt.connect("mqtt://host.docker.internal");

clientMQTT.on("connect", function () {
    console.log("Function connected");
    clientMQTT.subscribe(fc_name + "/in");
});

clientMQTT.on("message", (topic, message) => {
    let mesg = message.toString();
    if (topic === fc_name + "/in" && mesg !== "") {
        //Main function code from here
    }
});
```

4) The next step deploys both the function and its runtime software stack. To do so, the SERVERLESS4IOT orchestrator sends the deployment model to the GeneSIS instance running on the SERVERLESS4IOT gateway (and also deploys it, if absent) and triggers its deployment.
5) Finally, the function is started, registers itself with the local broker, and is marked in the platform as running.

B. Deployment on leaf nodes with no Internet connection

Some leaf nodes (*e.g.*, Leaf node 1 in Fig. 5) are not directly connected to the Ìnternet and can only be reached via some other devices, typically gateways. For instance, refer to the Arduino connected to the Rasp_Presence board via serial port in our motivating example. Gateways are thereby act as entry points for operating a deployment on leaf nodes. However, they are not necessarily *(i)* SERVERLESS4IOT gateways, meaning they do not host the SERVERLESS4IOT deployment engine, or *(ii)* directly accessible from outside of the LAN), which prevents their use by the Orchestrator to deploy or update software on target leaf nodes [8].

To address this issue, we exploit the concept of a *deployment agent*. While SERVERLESS4IOT gateways cannot access leaf nodes connected to another gateway, they are part of a local subsystem and thus have access to all other gateways in the same LAN. We use the SERVERLESS4IOT deployer to generate and deploy an agent on other devices, to which it can partially delegate its deployment activities, thus making all nodes in the local subsystem accessible for deployment. As a result, the overall process of deploying or updating software on Leaf node 1 depicted in Fig. 5 consists of the following steps. Once the SERVERLESS4IOT deployer is up and running on the SERVERLESS4IOT gateway (step ① in Fig. 5), the latter generates and deploys a deployment agent on the regular gateway (step ②), which in turn is responsible for deploying a new software component on Leaf node 1 (step ③).

The target device for hosting the deployment agent is either defined in the deployment model, tagging a communication link between two Components as requiring a deployment agent (*i.e.*, needDeployer property) or automatically deduced if the host is linked only to one other InfrastructureComponent via a communication tagged as physical.

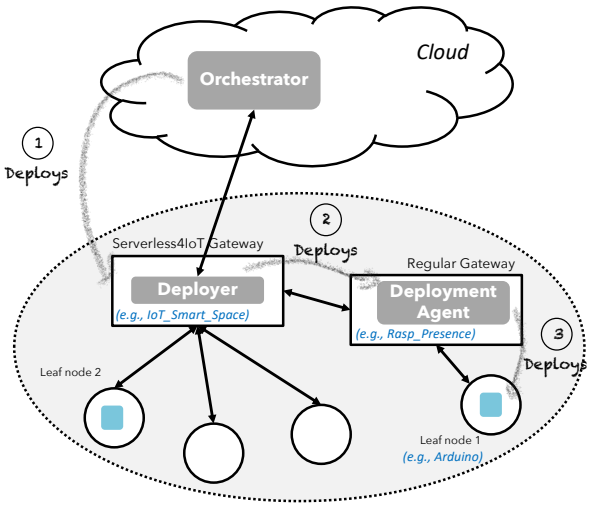

Figure 5. Deployment on leaf nodes.

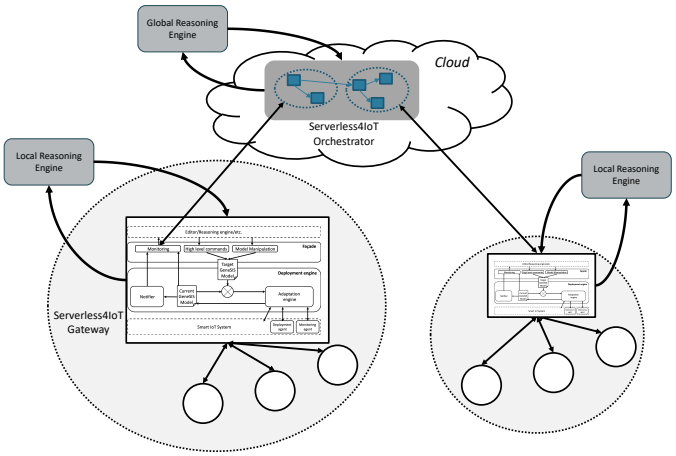

Figure 6. The SERVERLESS4IOT mega-model at runtime approach.

C. Orchestration deployment and system runtime management

As explained before, a SERVERLESS4IOT model can be composed of several deployment models, all following the same meta-model, thereby making the composition and links between the elements across the models models easy. A single deployment model is typically used to specify the deployment of one local subsystem. The overall SERVERLESS4IOT model is thus kept in the SERVERLESS4IOT orchestrator, while the deployment models it is composed of are distributed to the target local subsystems. More precisely, they are distributed to the deployer in each local subsystem. This distribution, in addition to the aforementioned benefits, allows developers to implement (self-)adaptation mechanisms with reasoning at the local and global (see Fig. 6) levels. To enable this, we adopt a *megamodel at runtime* approach [9]. A megamodel is defined "*as a model that contains models and relations between these models or between elements of those models*" [10]. The megamodel at runtime approach proposes to adopt the *models@run.time* [11] strategy at the scale of a megamodel. Below, we describe our megamodel at runtime solution.

At the local subsystem level, the deployer (*i.e.*, a GeneSIS deployment engine) implements the models@run.time architectural pattern [6]. Models@run.time [11] is an architectural pattern for dynamic adaptive systems that leverage models as executable artefacts that can be applied to support the execution of the system. Models@run.time provide abstract representations of the underlying running system, which facilitates reasoning, analysis, simulation, and adaptation. A change in the running system is automatically reflected in the model of the current system. Similarly, a change to this model is enacted on the running system on demand.

When a target model is fed to the deployment engine, it is compared with the deployment model representing the running system. The adaptation engine enacts an adaptation (*i.e.*, the deployment) by translating the difference between the current and the target states into one or several deployment steps. After the deployment, the engine synchronizes the current deployment model with the actual deployment result. The reasoning engine can use the monitoring interface exposed by the system to subscribe to all changes made on the current model (*i.e.*, when the running system is adapted).

Similarly, at the global level, the SERVERLESS4IOT Orchestrator also implements the same models@run.time pattern. However, because it cannot directly monitor the different running subsystems it is composed of, it subscribes to the monitoring interface of each of the deployers to synchronize its current model of the global system.

VI. RELATED WORK

Big cloud providers offer FaaS platforms (*e.g.*, AWS Greengrass[4]) and support for deploying and managing Edge and IoT infrastructures (*e.g.*, Azure IoT Hub[5]). However, in contrast to SERVERLESS4IOT, they typically: *(i)* promote vendor lock-in, and *(ii)* do not support the deployment of an ad-hoc software stack at the Edge level as well as supporting services required by a serverless function.

FogFlow [3], [12] is an open-source FaaS platform supporting the deployment and orchestration of functions, so called *fog functions*, on Cloud and Edge infrastructures. FogFlow adopts a data-centric programming model rather than following the more classical topic-based approach. While in many aspects our approach inspires from FogFlow, the latter focuses only on the deployment of functions and thereby does not integrate with other software deployment solution. Similarly, FogFlow does not offer specific support neither for the deployment of the software stack nor for the supporting services required by a serverless function. These have to be included in the container image that embeds the serverless function. Furthermore, FogFlow does not offer support for the integration of IoT devices, *i.e.*, for deploying software on devices with no direct Internet access, and the integration of

[4]https://aws.amazon.com/greengrass/
[5]https://azure.microsoft.com/services/iot-hub

sensors and actuators must be done using an IoT Agent, which is not part of FogFlow but of the FIWARE ecosystem.

Contrary to FogFlow, the RADON H2020 project[6] and the SODALITE@RT[7] solution [13] support deployment of hybrid applications combining microservices, serverless functions and data pipelines. Both use and extend OASIS TOSCA [14], a standard for specifying deployment and orchestration of cloud-based applications. Kappa [15] is a FaaS platform specifically designed for the IoT. It is a framework and REST API built on top of the Calvin [16] distributed IoT framework, which provides a solution to deploy and operate simple software components called actors on Cloud and Edge resources and to orchestrate them using CalvinScript. In contrast to our approach, these approaches do not offer specific support for the IoT space and in particular for the deployment of software components on IoT devices and devices with no direct access to Internet.

VII. Conclusion and Future Work

This paper presented our initial solution for the design, deployment, as well as maintenance of hybrid applications involving serverless functions and other software components over the Cloud-Edge-IoT continuum. The proposed solution is built using Model-Driven Engineering techniques and methods. The domain-specific modelling language facilitates the specification of reusable and platform-independent deployment of hybrid applications, while the underlying platform enables actual deployment over the whole Cloud-Edge-IoT continuum, including devices with limited access to the Internet. SERVERLESS4IOT is work in progress, and we identified some challenges we intend to address in future work.

– **Cold start**: Currently, SERVERLESS4IOT functions are automatically deployed, started, and terminated by the platform, functions are stopped on explicit user requests. In the future, we will investigate a configurable solution, allowing developers to define their own cold start strategies.

– **High availability and security**: We plan to extend the platform to support aspects related to high availability. As a first step, we will investigate the use of solutions complementary to Docker such as Kubernetes or KubeEdge. Security will also be considered in future version of the platform.

– **Deployment on IoT devices/Things**: We will investigate the feasibility of deploying serverless functions on IoT devices (*e.g.*, Arduino, Espruino). This raises a set of research questions. For example, how to ensure isolation between functions on such devices? Virtualization is the preferred way to facilitate isolation in the Cloud or Edge contexts. However, in many cases, IoT devices are not capable of running neither containers nor virtual machines. Another question is how to manage access to resources that need to be shared between serverless functions? In the first place, these resources include hardware sensors and actuators, which thus will need to be concurrently accessed by the serverless functions.

[6]https://radon-h2020.eu

[7]https://www.sodalite.eu

– **Smart allocation of functions and scalability**: Currently, serverless functions and software components are manually allocated on various Cloud, Edge, and IoT resources. Also, our solution does not offer any auto-scaling mechanism. In future work, we will extend the SERVERLESS4IOT Orchestrator with mechanisms for dynamic allocation of serverless functions and software components as well as support for auto-scaling.

Acknowledgement

This work is partially funded by the European Commission's H2020 Programme under grant 101020416 (ERATOSTHENES), and the Norwegian Research Council's IPN programme no. 309700 (FLEET)

References

[1] H. Ujhazy and B. Rojas, "Architecting the Data Landscape in IoT," International Data Corporation, Tech. Rep., 2021.

[2] N. Ferry, A. Solberg, H. Song, S. Lavirotte, J.-Y. Tigli, T. Winter, V. Muntés-Mulero, A. Metzger, E. R. Velasco, and A. C. Aguirre, "Enact: Development, operation, and quality assurance of trustworthy smart iot systems," in *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Springer, 2018, pp. 112–127.

[3] B. Cheng, J. Fuerst, G. Solmaz, and T. Sanada, "Fog function: Serverless fog computing for data intensive IoT services," in *2019 IEEE International Conference on Services Computing*. IEEE, 2019, pp. 28–35.

[4] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of function-as-a-service software development in industrial practice," *Journal of Systems and Software*, vol. 149, pp. 340–359, 2019.

[5] A. Noguero, A. Rego, and S. Schuster, "Towards a Smart Applications Development Framework," *Social Media and Publicity*, vol. 27, 2014.

[6] N. Ferry, P. H. Nguyen, H. Song, E. Rios, E. Iturbe, S. Martinez, A. Rego *et al.*, "Continuous deployment of trustworthy smart iot systems." *The Journal of Object Technology*, 2020.

[7] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, and F. Leymann, "A systematic review of cloud modeling languages," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–38, 2018.

[8] H. Song, R. Dautov, N. Ferry, A. Solberg, and F. Fleurey, "Model-based fleet deployment of edge computing applications," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2020, pp. 132–142.

[9] T. Vogel, A. Seibel, and H. Giese, "The role of models and megamodels at runtime," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2010, pp. 224–238.

[10] J. Bézivin, F. Jouault, and P. Valduriez, "On the Need for Megamodels," in *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, Canada, Oct. 2004.

[11] G. Blair, N. Bencomo, and R. France, "Models@run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[12] B. Cheng, G. Solmaz, F. Cirillo, E. Kovacs, K. Terasawa, and A. Kitazawa, "FogFlow: Easy programming of IoT services over cloud and edges for smart cities," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 696–707, 2017.

[13] I. Kumara, P. Mundt, K. Tokmakov, D. Radolović, A. Maslennikov, R. S. González, J. F. Fabeiro, G. Quattrocchi, K. Meth, E. Di Nitto *et al.*, "Sodalite@rt: orchestrating applications on cloud-edge infrastructures," *Journal of Grid Computing*, vol. 19, no. 3, pp. 1–23, 2021.

[14] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "TOSCA: portable automated deployment and management of cloud applications," in *Advanced Web Services*. Springer, 2014, pp. 527–549.

[15] P. Persson and O. Angelsmark, "Kappa: serverless ioi deployment," in *Proceedings of the 2nd International Workshop on Serverless Computing*, 2017, pp. 16–21.

[16] ——, "Calvin–merging cloud and IoT," *Procedia Computer Science*, vol. 52, pp. 210–217, 2015.