This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JIOT.2021.3101449, IEEE Internet of Things.

EdgeFlow - Developing and Deploying Latency-Sensitive IoT Edge applications

Cosmin Avasalcai, *Student Member, IEEE,* Bahram Zarrin, and Schahram Dustdar *Fellow, IEEE,*

***Abstract*—Demanding latency-sensitive IoT applications have stringent requirements like low latency, better privacy and security. To meet such requirements, researchers proposed a new paradigm, i.e., edge computing. Edge computing consists of distributed computational resources and enables the execution of IoT applications closer to the edge of the network. However, the distributed nature of this paradigm makes the application deployment and development process more challenging since the developer must divide the application's functionality into multiple parts, assigning for each a set of requirements. As a result, the developer must (i) define the application's requirements and validate them at design time and (ii) find a deployment strategy on the target edge computing platform. In this paper, we propose EdgeFlow, a new IoT framework capable of assisting the developer in the application development process. Specifically, we introduce a methodology for latency-sensitive IoT applications development and deployment, consisting of three different stages, i.e., the development, validation, and deployment. To this end, we propose an extension of the Flow-Based Programming paradigm with new timing requirements and provide a resource allocation technique to assist with the deployment and validation of latency-sensitive IoT applications. Finally, we evaluate EdgeFlow by (i) presenting the application development methodology and (ii) performing a quantitative evaluation demonstrating our resource allocation technique's capabilities to find feasible and optimal deployment strategies. Experimental results illustrate the effectiveness of our methodology to assist the developer throughout the entire application development process.**

***Index Terms*—IoT Application development, Flow-Based Programming, Edge Computing, Resource Management.**

I. Introduction

Latency-sensitive Internet of Things (IoT) applications have stringent requirements, e.g., low latency, better privacy and security. Current cloud-centric solutions fail to satisfy these requirements since high volumes of data must be transferred to the cloud [1]. Hence, to successfully met the application's requirements, we must take advantage of the distributed computational nodes found in an IoT system. As a result, a latency-sensitive IoT application consists of multiple interconnected components; a component is capable of executing one part of the application's functionality. However, developing and deploying such an application model is not a trivial task since the developer must (i) define and validate the application's requirements at design time and (ii) find a deployment strategy such that it satisfies all application requirements.

To address the shortcoming of cloud computing, researchers have proposed edge computing [2]. Edge computing enables the utilization of available computation resources found at the edge of the network [3], [4] – a paradigm consisting of multiple geo-distributed resource-constrained devices capable of hosting deployed IoT applications. Edge computing assists cloud computing in satisfying the stringent requirements of latency-sensitive IoT applications, where components may be deployed on edge nodes. Some advantages of edge computing include low latency and data locality [5]. Nevertheless, deploying an application on an edge computing platform is challenging since heterogeneity and limited resource capabilities define an edge node. As a result, the successful deployment of latency-sensitive applications is dependent on new resource allocation techniques.

Edge computing brings many advantages for the deployment of latency-sensitive IoT applications. However, edge computing makes the application development process more challenging, since the developer must divide the application's functionality and define different requirements for each component [6]. Previously, in a cloud-centric system, a single component contains the entire application's functionality and it is deployed in a single location, i.e., in the cloud. In contrast, in an edge computing platform, the application model consists of multiple components that are distributed among different edge devices. An application model that is in line with the flow-based programming (FBP) paradigm [7] concepts; an application has a communication flow that connects different components to achieve certain functionality. Several FBP tools like noFlo [8], node-RED [9], and drawFBP [10] exist to aid the developer in creating new IoT application models and define their communication flow. However, it is still challenging to define and validate the application's timing and resource requirements during the development stage.

In this paper, we propose EdgeFlow, a new IoT framework for latency-sensitive IoT applications development and deployment. Our main contribution is a methodology for aiding the developer in the process of creating and deploying applications by (i) defining new applications' timing and resource requirements, (ii) validating all requirements, and (iii) finding a deployment strategy.

Development stage. We propose an *IoT application modeling paradigm* for developing latency-sensitive applications at design time. The purpose of this stage is to collect as much information as possible regarding the current application – information that improves the chances to successfully deploy an application on the target edge computing platform. As a result, we employ the FBP paradigm as an application model to define latency-sensitive IoT applications and extend this paradigm with new timing requirements allowing the developer to provide timing and resource requirements. We allow

C. Avasalcai and S. Dustdar are with the Distributed Systems Group, Vienna University of Technology, Vienna, 1040, Austria.
E-mail: {c.avasalcai, dustdar}@dsg.tuwien.ac.at
B. Zarrin is with Microsoft, Kongens Lyngby, Denmark.
E-mail: bahramzarrin@microsoft.com

for a higher granularity when defining the application's timing requirements. As a result, for a latency-sensitive IoT application, the developer can define an end-to-end (e2e) delay for many communication flows ranging from the communication link between two components to a flow containing the entire application (if possible). To evaluate our development stage, we create a prototypical framework based on the *drawFBP* tool. We describe the application development methodology by creating an IoT application.

Deployment and validation stages. The *deployment stage* offers support for deploying latency-sensitive applications on edge computing platforms. This stage provides validation for defined application constraints by determining eligible deployments (if any) of the designed application to the target edge computing platform. We cast our deployment technique within constraint programming (CP) paradigm [11], where we define the deployment constraints as a constraint satisfaction problem. Consequently, the deployment stage can generate feasible or optimal deployment strategies – it provides guarantees that if a deployment strategy exists, the technique can find it. A deployment strategy that (i) satisfies each component's resource requirements while not exceeding the device's available resources and (ii) meets the communication flow constraints, i.e., ensuring that the e2e delay of each communication flow does not exceed the determined one. Finally, we evaluate our deployment stage performance by assessing the execution time required to find an optimal deployment strategy.

The contributions of this paper are as follows:

- *EdgeFlow*. A methodology for latency-sensitive IoT applications development and deployment. Our proposed methodology aids the developer in defining and validating timing and resource requirements as well as finding optimal and feasible deployment strategies.
- *Development stage*. We propose an extension of the FBP programming paradigm with new concepts like timing and resource requirements. By introducing new timing requirements, we support the definition of multiple e2e delays for different communication flows for the developed application.
- *Deployment stage*. We introduce a novel resource allocation technique capable of finding optimal or feasible deployment strategies. Our main objective is to find a deployment strategy that satisfies all timing and resource requirements defined in the development stage.
- *Validation stage*. We provide validation for all application requirements introduced during the development stage using the proposed resource allocation technique from the deployment stage. As a result, the developer can refine the defined requirements, considering the target edge computing platform.

The remainder of the paper is structured as follows. In Section II we summarize the related work. Section III provides an overview of EdgeFlow and defines the application model, the edge computing platform, and the communication flows constraints, given as input files to the deployment stage. In Section IV we describe the implementation details of our proposed framework. Section V presents the application development methodology, while Section VI shows the results of our deployment stage evaluation. Finally, Section VII concludes the paper and provides an outlook on future work.

II. Related Work

The adoption of edge computing and the stringent application requirements have changed the application's deployment and development process. Recently, the consensus, in the research literature, depicts an application model as a collection of components to accommodate the distributed nature of edge computing [12], [13], [14], [15]. Typically, researchers consider as given the application model and its associated timing and resource requirements when proposing new application deployment techniques. However, developing an application model and defining all requirements is not a trivial task.

Only recently, researchers have proposed techniques to aid with the IoT application development process. Giang et al. [16] present a distributed dataflow programming model for fog computing that aids the developer during the application development process. In this case, the developer defines the application model as a directed graph, dividing the application's functionality between different application nodes. Wang et al. [17] propose a stream processing approach, i.e., Edge-Stream, for building new applications for edge computing systems. Edge-Stream represents data flows between the application's components as streams. Frasad [18] is another framework that helps with the IoT application development and makes use of a model-driven design approach to enhance the reusability, flexibility, and maintainability of sensor software. Rafique et al. [19] develop an IoT application development framework using model-driven development and attribute-driven design. The framework transforms the application's requirements into a solution architecture using the attribute-driven design and then uses model-driven development to generate models to transform the application's components into software artifacts. Other papers make use of FBP for the IoT applications development process. Szydlo et al. [20] introduce a heuristic data flow transformation technique to successfully distribute flows on the target network, while Belsa et al. [21] present a solution to interconnect services from different IoT platforms. Jain et al. [22] propose a mapping technique composed of two stages: (i) the IoT application is modeled into multiple different tasks annotated with target location information and (ii) each task is deployed on an edge node based on its location. The authors extend Node-RED to allow the development of the IoT application and deployment of defined components to their predefined location, i.e., cloud or edge. Compared to the related IoT development approaches, we focus on deployment and validation of IoT applications on different edge computing platforms without the need to introduce predefined locations for components – we enable the deployment of applications on large-scale platforms.

The deployment problem exists in many variants in the scientific literature [23], [24], [25], [26]. The two most common scenarios where researchers propose deployment techniques are (i) service placement and (ii) service offloading. The former migrates services that reside in the cloud closer to the edge of the network, i.e., on edge or fog nodes. In contrast, the latter moves services from resource-constrained devices, e.g., smartphones, to nearby edge nodes, in an attempt to preserve the energy of devices. Brogi et al. [27] propose a deployment technique having as objective the latency and bandwidth. As a result, the proposed solution provides Quality of Service (QoS) aware deployments of IoT applications on a target fog computing architecture. Scoca et al. [28] propose a latency, bandwidth, and resource-aware scheduling algorithm that finds a mapping of services to edge nodes. The approach uses a score-based technique that evaluates the target edge nodes and communication links and computes a scoring mapping for each service. The main objective of this technique is to guarantee optimal service quality. In [29], Redowan et al. introduce a latency-aware technique aiming to deploy the application's modules on fog computing such that it satisfies all objectives. The approach has two objectives, i.e., (i) to satisfy the application's latency requirements and (ii) to optimize the utilization of the node's available resources. Liu et al. [30] propose a task offloading technique that aims to minimize the system cost, i.e., energy and latency. This technique groups the users into clusters based on their priorities and decided if a cluster should run all its tasks locally or should be offloaded to an edge server. Grosu et al. [31] introduce an online heuristic algorithm based on Mixed Integer Linear Program to deploy multi-components applications on edge computing platforms. As we can see, all approaches strive to achieve at least one objective, i.e., latency. However, the deployment problem is implemented as a resource allocation optimization problem leveraging assumptions about the application model.

Note that, in the presented related work, some solutions consider as target deployment platforms either fog or edge computing paradigms. These two paradigms have the same underlying premise of migrating computational resources closer to the edge of the network [6]. Therefore, from the perspective of EdgeFlow, using one paradigm over the other poses no impact on the EdgeFlow functionality – in both cases, the available resources are shared between the participant devices. We differentiate ourselves from the aforementioned related work from two big perspectives: we (i) extend the FBP paradigm with new timing requirements and (ii) propose a new deployment technique. With the former, we allow the developer to define new timing requirements for each application component and communication link. Moreover, we introduce a new timing constraint for multiple communication flows, i.e., the developer can define individual delays for many communication flows of different sizes. The latter technique can find feasible or optimal deployment strategies, fulfilling the timing and resource requirements. Furthermore, since we are using CP, the technique can validate the application's timing and resource requirements considering the target edge computing platform.

III. EdgeFlow: Application Development and Deployment framework

New latency-sensitive IoT application models achieved through edge computing decompose the application's functionality into multiple distributed components. As a result, the developer must define new timing and resource requirements for each component and the overall application – besides the maximum e2e delay involved in the correct application functionality, the developer must define specific requirements for each component. As such, the developer must be able to define and validate all requirements during the application's development process; since these requirements play an active role in the application deployment.

In our framework, we provide a methodology to develop and deploy IoT applications on the target edge computing platform. For the former, we offer support for creating the application model and defining the timing and performance requirements. For the latter, we propose a deployment stage capable of finding a deployment strategy at design time. Depending on the type of the target edge computing platform, i.e., static or dynamic, the deployment stage provides a different utility. In the case of static architectures, e.g., like in a smart factory, the deployment stage can generate feasible or optimal deployment strategies. However, if the target platform is a dynamic edge computing architecture characterized by high uncertainty and node mobility, then the deployment stage can only validate the application requirements; a dynamic network may change while we search for the optimal deployment strategy at design time. As a result, for dynamic architectures, we can use a decentralized resource allocation technique capable of finding deployment strategies at runtime [32]. Figure 1 presents an overview of our EdgeFlow methodology consisting of three distinct stages, i.e., (i) the development stage, (ii) the deployment stage, and (iii) the validation stage.

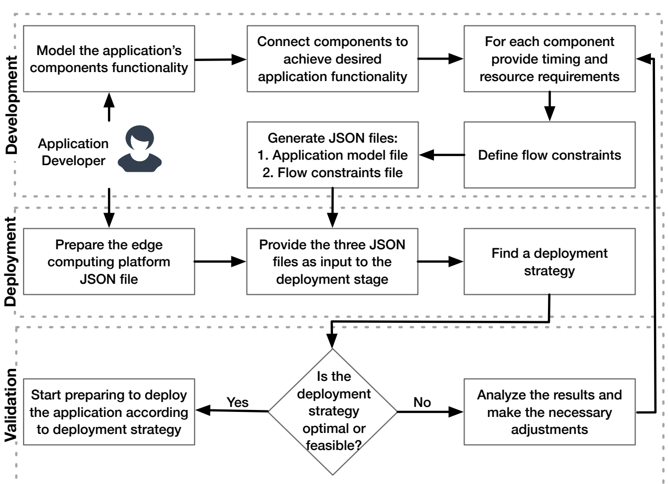


Fig. 1: EdgeFlow methodology overview.

A. Development stage

The *application development stage* is an extension of the FBP programming paradigm, introducing new timing and resource requirements. The application modeling paradigm offers the possibility to divide the application's functionality into different components and build the application's communication flow such that the application performs certain functionality.

The FBP paradigm views an application as a network of processes, i.e., components, interconnected via predefined communication links. Each component runs asynchronously and communicates via streams of data chunks, i.e., Information Packets (IPs) [33]. FBP is component-oriented, allowing the developer to develop different applications using the same network of components – a practice that improves the application development process and enhances the reusability of components. FBP is not a coding language. As a result, it is ideal to use predefined components from a library.

FBP extension. The FBP paradigm does not provide the possibility to define Quality of Service (QoS) requirements, i.e., timing requirements, data locality, affinity and anti-affinity constraints between components, privacy [34], [35], and security [36], during the application's modeling stage. In this paper, we target the development and deployment of latency-sensitive IoT applications – one of the fundamental concerns of these applications is latency. To this end, we propose an extension of the current FBP paradigm with new timing requirements[1].

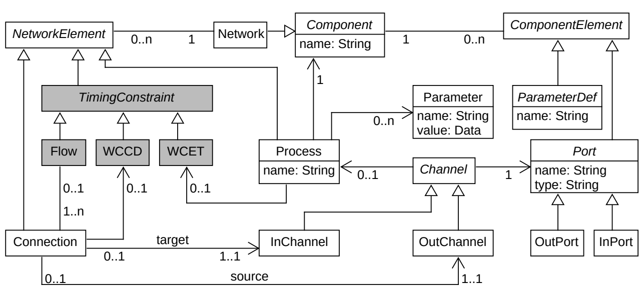


Fig. 2: The FBP paradigm extension metamodel.

In our opinion, three essential timing requirements define a latency-sensitive application, i.e., worst-case execution time (WCET), e2e delay for different flows, and worst-case communication delay (WCCD). Each application's component has associated a WCET representing the time required to produce a result. Similarly, we define, for each communication link, a maximum WCCD serving as the time that an IP needs to reach its destination. Finally, we provide the means to define an e2e delay for multiple communication flows. Notice that the first two timing constraints are part of the e2e delay computation since a communication flow consists of one or more components and communications links alike. In [37], authors formalized the syntax and semantics of Flow-Based languages, and they proposed a metamodel for FBP. Figure 2 presents our extended metamodel based on their formalism.

Application model. An application model is defined as an FBP network which consists of a set of components $C=\{c_1, c_2, ...\}$ that collaborates to perform a certain goal. An application may have one or more source components (i.e., the component that provides the required data) as well as at least one sink (i.e., a component that acts according to the data received). In Figure 3, we present an example of an application model having one source and two sinks, where the communication flow starts with the *source component*, i.e., c_0, and finishes with two *sink components*, i.e., c_4 and c_6.

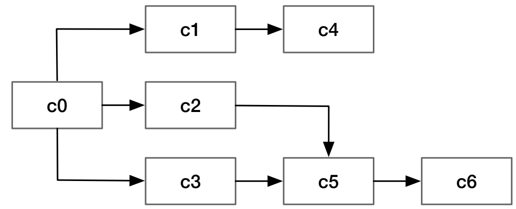


Fig. 3: Latency-sensitive IoT application model.

A component c_i performs a certain functionality and represents a containerized microservice or serverless function. Each component is characterized by a set of timing and resource requirements, $C_{req}=\{r_1, r_2, ...\}$ as well as a set of input and output ports, $C_{in}=\{in_1, in_2, ...\}$ and $C_{out}=\{out_1, out_2, ...\}$. During the application development process, the developer defines these requirements according to the application's goals. A resource requirement represents the generic memory (i.e., RAM), computational power (i.e., CPU), and storage (i.e., HDD) requirements, while the WCET of a component is an example of a timing requirement. To fit better the application's needs, in future work, we intend to extend the components' resource requirements with specific requirements, e.g., hardware requirements like GPUs for high computational components or specific data that must be present on the host node.

B. Edge Computing platform

An edge computing platform consists of multiple distributed edge nodes, having the following characteristics: (i) heterogeneity, (ii) limited computational resources, and (iii) mobility. Let $E_N=\{E_1, E_2, ...\}$ be a set of edge nodes found in the target architecture. Each node is characterized by a set of available resources, $E_{res}=\{r_1, r_2, ...\}$, like RAM, CPU, and HDD, and a list of communication links $Link_{com}=\{link_1, link_2, ...\}$ – each $link_i$ having associated a bandwidth.

Based on the platform's characteristics and the administrative entity control level, we identify two types of edge computing platforms, i.e., a dynamic platform and a static platform. The dynamic platform consists of different mobile and static edge nodes owned by distinct administrative entities. As a consequence, it introduces a high uncertainty level into the system, making the deployment of an application at design time more challenging; an example of such a platform is the typical smart city scenario. In contrast, the static platform has a low uncertainty where the developer knows the nodes' characteristics at design time, e.g., a smart factory scenario.

C. Deployment and validation stages

The *deployment and validation stages* uses a resource allocation technique aiming to help the developer to validate the defined application's requirements considering the target edge platform's available resources. Consequently, we develop our resource allocation technique using Constraint Programming (CP), which produces both feasible and optimal deployment strategies. Notice that CP fits rather well with our deployment

[1] We identify all other QoS requirements as an interesting path for future work. Furthermore, researchers can contribute to the IoT application modeling paradigm by providing extensions for new requirements.

stage since our primary focus is to validate the application's requirements – CP provides guarantees that if a deployment strategy is possible, then it satisfies all requirements. To deploy an application, the deployment stage requires information regarding the application model and the target edge computing platform. The developer provides all required information as three input files, i.e., *application model file*, *edge computing platform file*, and *flow constraints file*; the developer can generate the files using the *IoT application modeling stage* or can create them manually.

Application model file. The *development stage* provides the developer with the ability to generate the application model file –a process that stores all application resource requirements in a JSON file. In the end, the JSON file contains information about the application's components, such as input and output ports, the period and data size associated with the ports, resource requirements (RAM, CPU, HDD), and the WCET.

Edge computing platform file. Since the edge computing platform is not modeled with the FBP paradigm, we assume that the developer obtains this file from the administrative entity that owns the platform, i.e., considering the static platform scenario. For the dynamic platform, we assume that the file represents an estimation of the possible current topology determined from the history data stored in the cloud.

Flow constraints file. This file contains the application's constraints – the deployment strategy uses them as objectives. We offer the developer the possibility to add for each flow found in the application model an e2e delay constraint. The e2e delay considers both the WCET of each component found on the path as well as the communication latency used when components exchange IPs. For example, consider the application shown in Figure 3, the developer can create multiple constraint flows between its components. There are three big flows consisting of the following components: (i) $c_0 - c_2 - c_5 - c_6$, (ii) $c_0 - c_3 - c_5 - c_6$, and (iii) $c_0 - c_1 - c_4$ respectively. However, the developer can add a constraint even for a smaller flow consisting of a minimum of two components, e.g., $c_0 - c_1$. In this paper, we assume that the developer provides at least the number of flows required to involve all communication links and components found in the application model. If any component remains outside of a defined flow constraint, then our deployment stage will consider it as a single component with no dependencies.

Grammar 1: Flow Constraint Language

```
<Delay> :: <Number> ms | <Number> ns
<BooleanOp> :: < | > | >= | <= | =
<FlowSource> :: <InPortID> | <DataPacketID>
<FlowSink> :: <OutPortID> | <ComponentFlow>
<ComponentFlow> :: <InPortID> <ComponentID> <OutPortID>
<FlowPath> :: <FlowPath> -> <ComponentFlow> |
    <ComponentFlow>
<Flow> :: <FlowSource> -> <FlowPath> -> <FlowSink> |
    <FlowSource> -> <FlowSink>
<FlowConstraint> :: <FlowID> : <Flow> <BooleanOp> <Delay>
<FlowConstraints> :: <FlowConstraints> ; <FlowConstraint> |
    <FlowConstraint>
<FlowConstraintsDef> :: flow constraints <AppID>
    <FlowConstraints> end
```

The proposed IoT framework utilizes a mini-language to specify the flow constraints, see Grammar 1. The developer can add the *flow constraints* using this language to specify the flow's path and the maximum e2e delay. In Equation 1, we present an example of a flow containing two components c_1 and c_2. In the flow declaration, the IN and OUT ports represent the name of the input and output ports used by each component. As we can observe, the colon separates the flow's path declaration from its id, while $\leq$ shows the relation between the path and the e2e delay and $\rightarrow$ represents the direction of the communication. Furthermore, the last component does not need an output port, this highlighting that the path is ending.

$$\mathsf{path}_1 : \mathsf{IN}\, c_1\, \mathsf{OUT} \rightarrow \mathsf{IN}\, c_2 \leq \mathsf{e2eDelay} \tag{1}$$

IV. Application development and deployment stages

In this section, we present the two stages that represent the core of the EdgeFlow framework, i.e., the *development stage* and the *deployment stage*. For the former, to prove our concept, we develop a prototype to help the developer in creating new application models and defining their timing and performance requirements. For the latter, we propose a deployment technique capable of providing deployment strategies such that it satisfies all application's requirements and constraints.

A. Development stage prototype

To prove the benefits of creating a new latency-sensitive application using our IoT framework, we develop an *application development prototype* based on *drawFBP*. DrawFBP uses FBP at its core and allows developers to create diagrams using blocks, i.e., components [10]. An advantage of drawFBP is that developers can generate different components that other developers can reuse – the developer can create them using Java, C#, or JSON. As a result, the developer can use existing components from the drawFBP library during the application development process. In this case, the development process resumes at creating a communication flow between selected components such that it fulfills the application's goals. However, defining the application's communication flow and choosing the components is not enough; the developer must define specific requirements for both the communication flows and for each component.

We extend drawFBP with new options, like *set component requirements*, *set flow constraints*, *application model: generate JSON file*, and *flow constraints: generate JSON file*, offering developers the possibility of adding timing requirements. Using the *set component requirements* option, the developer can describe for each component the following characteristics, i.e., WCET, period, message size, and resource requirements (RAM, CPU, HDD). Furthermore, the developer can define different e2e delay constraints for custom communication flows using the *set flow constraints* option. Finally, we collect all information into two input files, i.e., *the application model* and *the communication flow constraints*, using the two *generate JSON file* options; files that the *deployment stage* uses as input.

B. Deployment stage technique

Our deployment technique helps the developer to decide if the application can be deployed on the target edge computing platform. Depending on the success of the deployment, the developer gets more clarity in defining the component's resource requirements and the application's constraints. Two cases lead to deployment failure, i.e., (i) the application has very stringent requirements and (ii) the target platform lacks the required available resources. Under these conditions, if the *deployment stage* does not find a deployment strategy, the developer can investigate one of the two cases and make the required adjustments accordingly. Therefore, the developer can use the *deployment stage* to understand if the target edge computing platform can host the application. As a result, developers can create better application models suitable for deployment on a large variety of platforms.

As mentioned in the previous section, we implement the resource allocation technique using CP. Depending on the strategy found, CP can return one of the four different status values, i.e., (i) *optimal*, (ii) *feasible*, (iii) *unknown*, and (iv) *infeasible*. The deployment technique found a deployment strategy that meets the requirements if the returned status is (i) or (ii). In contrast, if the returned status is (iv), then the technique cannot find a deployment strategy that meets all requirements. An interesting state, i.e., (iii), may appear when the developer decides to limit the execution time of the deployment stage. Under these conditions, the technique is unable to decide if the current deployment strategy satisfies all application's requirements – as a result, it returns *unknown*. To find a deployment strategy, we model the problem using *decision variables*, *constraints*, and *global objective* and solve it using a CP solver.

The procedure starts once the deployment technique receives the required input files from the developer. Using the information received as input, we can create a set of *decisions variables* used in the CP model. A decision variable represents a variable for which the CP solver tries to assign a value chosen from a predefined domain to satisfy the application's requirements. In our case, we identify four different decision variables, i.e., *component variables*, *latency variables*, *WCET variables*, and *resource variables*. From all these decision variables, only the *component variables* yields an allocation, while all the other variables are support variables for validating the chosen deployment strategy.

Component variables. The *component variables* define for each component a domain containing a list of edge nodes where the current component can be mapped. For example, component c_1 can be mapped only on nodes E_1, E_2, and E_3; hence, a valid domain for the decision variable of c_1 is $D=\{E_1, E_2, E_3\}$. Under these conditions, the solver can only choose a node from D to allocate c_1.

Latency variables. These variables are in charge of saving the communication latency between two components considering their mapping. Let us consider that two components c_1 and c_2 communicate with each other – c_1 is mapped on E_1 and c_2 is mapped on E_2. We can use the IP size and the bandwidth of the communication link used for communication to find the communication latency between the two components. To build the variable's domain, we use the dependencies between components described in the *communication flow constraints* input file and all possible locations from their respective domains devised in the *component variables*. As a result, to compute the communication latency between two dependent components, c_i and c_j, we take all possible distinct edge node combinations from their associated domains.

WCET variables. The *WCET variables* has the same purpose as the *latency variables*, i.e., to store the WCET given to each component. Since the WCET of a component is strictly dependent on the host's internal status, obtaining the exact WCET of a component is challenging; the edge computing platform consists of multiple heterogeneous devices, requiring a complete analysis of the WCET of a component on every edge node. We consider such analysis as out of scope for the current paper. Therefore, to lower the challenge in finding a suitable WCET, we assume the developer can provide a lower and an upper bound for the WCET of each component.

Resource variables. These variables keep track of the edge node's available resources. Every node starts with a predefined set of available resources; resources that decrease with the resource requirements of new mapped components. An approach that ensures the correct distribution of components on nodes without exceeding the node's available resources.

Once we add all decision variables to the CP model, we can continue with the introduction of our *constraints*. Each constraint represents an important part of our model, guiding the CP solver towards a feasible deployment strategy that considers the application's constraints. For this purpose, we define two different constraints, i.e., *components constraints* and *flows constraints*.

Components constraints. The *components constraints* ensure that the distribution of components on edge nodes does not exceed the node's available resources. To achieve such purpose, the *components constraints* make use of the following decision variables, i.e., *component variables* and *resource variables*. Equation 2, Equation 3, and Equation 4 guarantee that a deployment strategy does not exceed nodes' available resource, where n_c represents the total number of components mapped on the current node.

$$\mathsf{usedCPU} = \sum_{i=1}^{n_c} c_{CPU} \leq \mathsf{availalbe}_{CPU} \tag{2}$$

$$\mathsf{usedRAM} = \sum_{i=1}^{n_c} c_{RAM} \leq \mathsf{available}_{RAM} \tag{3}$$

$$\mathsf{usedHDD} = \sum_{i=1}^{n_c} c_{HDD} \leq \mathsf{available}_{HDD} \tag{4}$$

Flows constraints. By validating the *components constraints*, we can successfully deploy the application on the target edge computing platform. However, we only consider the application's resource requirements as a deployment objective. Therefore, we introduce a new set of constraints, i.e., *flows constraints*, to consider the flow constraints introduced in the *flow constraints file*. We build these constraints based on the *components variables*, *WCET variables*, and *latency variables*. By combining the three decision variables, we manage to further enforce constraints on the deployment strategy. In conclusion, we can observe that these constraints consider both WCET and communication latency. Equation 5 guarantees that the flow's e2e delay does not exceed the maximum e2e delay associated with it; the e2e delay of a flow is the sum of all participants components' WCET and their communication latency. In Equation 5, $\mathsf{l_f}$ represents the total number of links found in a flow *f*, $\mathsf{c_f}$ is the total number of components part of a flow f, and $\mathsf{maxE2Edelay_f}$ represents the maximum e2e delay allowed for flow *f*.

$$\mathsf{e2eDelay} = \sum_{\mathsf{link}}^{\mathsf{l_f}} \mathsf{link_{latency}} + \sum_{\mathsf{c}}^{\mathsf{c_f}} \mathsf{c_{wcet}} \leq \mathsf{maxE2Edelay_f} \quad (5)$$

Global objective. The purpose of this objective is to minimize the e2e delay of each flow. In doing so, we obtain a solution that offers an optimal deployment strategy if there is enough time to search for it. Equation 6 shows the *global objective*, where $\mathsf{n_f}$ represents the total number of flow constraints defined in the *communication flow constraints file* and $\mathsf{flowE2E_i}$ is the current e2e delay of flow *i*.

$$Min(\sum_{\mathsf{i=1}}^{\mathsf{n_f}} \mathsf{flowE2E_i}) \quad (6)$$

V. Application Development Methodology

EdgeFlow provides a framework that aids the developer in creating emergent latency-sensitive IoT applications and deploy them in an edge computing platform. In this section, we evaluate the applicability of our IoT framework by presenting the *application development experience*. We describe this as a step-by-step process using one latency-sensitive IoT application. First, we describe the development of the IoT application model and generate the input files that the *deployment stage* requires to validate the requirements and find a deployment strategy to map the application on an edge computing platform. The application development prototype and the deployment stage technique are available in our online appendix [2] and our git repository [3].

As a running exemplar, we model a public safety IoT application deployed in a smart city scenario. The application aims to prevent any possible attacks by analyzing all the images and videos from an area. The application consists of multiple components capable of analyzing both the environment as well as people. For example, the application sends an emergency signal to the police department if a suspicious package is found in the monitored area. Since our focus is to show the extensions and improvements we bring with our proposed IoT framework, we assume that the public safety application's components are available in the drawFBP library. In this setting, the developer must connect the components and add the timing and resource requirements.

[2]https://dsg.tuwien.ac.at/team/cavasalcai/projects/EdgeFlow
[3]https://github.com/cavasalcai/EdgeFlow

Considering the safety implications, the smart city application must adhere to some timing requirements such as low e2e delay. To prevent a possible disaster scenario, the application must be able to provide alerts without delay. Thus, the application must execute at the edge of the network. As a consequence, a prerequisite for the developer is to validate the timing and performance requirements on the target edge computing platform before deploying the application. As we will show, our IoT framework is capable of performing such validation. In our case, the application consists of five components, each enacting a specific functionality.

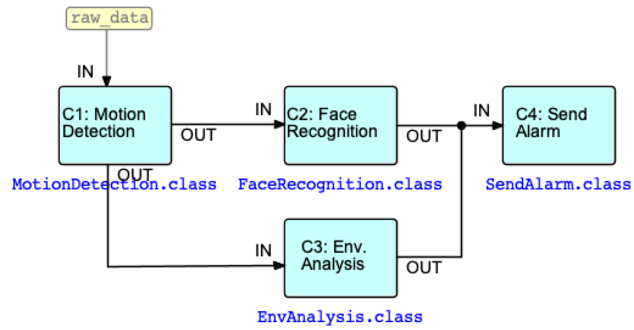


Fig. 4: Public Safety Application DrawFBP model.

Development stage. Using the *application development prototype*, the developer can create all components required for the application, add the functionality, and connect them via ports to create the application's functionality (see Figure 4). Currently, the developer has created the application model without defining the timing and resource requirements.

Once the model is complete, the developer can specify the timing and resource requirements using our FBP extension options presented in Section IV-A. To assign the component's requirements, the developer can use the option *set component requirements* available in the component menu; to access this menu, right-click on the target component. The process of setting the component's requirements goes through each requirement and asks the developer to provide a value or a range (in the case of WCET). To create new flow constraints, the developer can select the *set flow constraints* option from *file menu* and define a new flow constraint using the template from Equation 1. We have added support options, i.e., *display flow constraints* and *delete flow constraints*, that help the developer to display and delete all existing flow constraints.

Finally, there is one more step to perform before the developer can move to the*deployment stage*, i.e., to generate the *application model file* and the *communication flow constraints*. The developer can generate these files using the two options, i.e., *Application Model: generate JSON file* and *Flow Constraints: generate JSON file*, from the *file menu*. As explained in Section III-C, the developer can obtain the *edge computing platform file* from other sources.

Deployment stage. The contents of the three input files are presented in Table I, Table II, and Figure 5. Table I shows the target edge computing platform, where we can see the node's available resources, connections with other nodes, and the bandwidth of each communication link. For simplicity, we choose for each available resource, i.e., RAM, CPU, HDD, a value between 15 and 30 *units*. The deployment stage can operate with different units, e.g., MB or GB, as long as there is consistency between the available and required resources.

Nodes	Available Resources			Connections	
	RAM	CPU	HDD	destination	bandwidth
n_0	20	22	15	n_1	10
				n_2	15
n_1	17	30	18	n_0	10
				n_2	15
n_2	15	25	16	n_0	15
				n_1	15

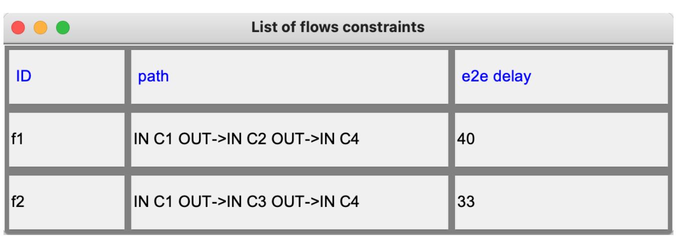
TABLE I: Edge Computing platform characteristics.

The *application model file* contains the timing and resource requirements of all components; requirements that we present in Table II. For our public safety application, we choose for each component the following: all resource requirements have a value between 1 to 15 units, we randomly select a data size value between 30 and 115 units, and add a custom range for the WCET considering each component's functionality.

Components	Resource Requirements			WCET	Data size
	RAM	CPU	HDD		
c_1	6	7	5	[5, 9]	60
c_2	13	15	7	[20, 35]	90
c_3	10	13	10	[15, 25]	75
c_4	3	2	3	[2, 4]	30

TABLE II: Application resource and timing requirements.

Figure 5 shows the declaration of the flow's path in the communication flow constraints file where a list of communication ports, the source, and the destination component describe each link found in the path. For our application, we assign two different flows, i.e., f_1 with the path $c_1 - c_2 - c_4$ and f_2 having the path $c_1 - c_3 - c_4$. For f_1 we chose an e2e delay equal to *40 ms* and for f_2 the maximum e2e delay is *33 ms*. To choose the maximum e2e delay, we consider the sum of the lower bound of the WCET of all components found on the communication flow. Furthermore, to this, we add a value of 10; this value reflects the impact of the communication latency between components.


List of flows constraints

ID	path	e2e delay
f1	IN C1 OUT->IN C2 OUT->IN C4	40
f2	IN C1 OUT->IN C3 OUT->IN C4	33

Fig. 5: Flow constraints for public safety application.

With the three files ready, the developer can start the process of finding a satisfiable deployment strategy. Considering the target edge computing platform, the deployment technique tries to find an optimal or feasible deployment strategy. Depending on what status the CP solver returns, the developer must decide if he/she should change the application's requirements or try to find a more suitable edge computing platform with more resources. In Figure 6, we can see the output of the deployment stage. We highlight the host node of each component by placing the node's id on the top left corner. For example, component c_1 is mapped on E_0 and component c_2 is mapped on E_1. In this case, the deployment stage finds the optimal deployment strategy in *10 ms*.

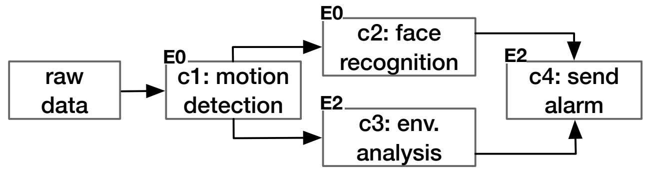


Fig. 6: Deployment strategy for public safety application.

The *deployment stage* returns a detail report showing the communication latency between components and their WCET concerning each communication flow constraint. In Table III we present the flows' e2e delay and the communication latency for the deployment strategy presented in Figure 6. We can observe that for the optimal solution, the actual e2e delay of flow f_1 is *34* ms, while for f_2 is *32* ms. Also, we can see that for f_1 the communication latency between components c_1 and c_2 is equal to *6* ms.

Flows	Components		Communication Latency		e2e delay
	ID	WCET	destination	latency	
f_1	c_1	5	c_2	0	33
	c_2	20	c_4	6	
	c_4	2	-	-	
f_2	c_1	5	c_3	4	26
	c_3	15	c_4	0	
	c_4	2	-	-	

TABLE III: Flows e2e delay and communication latency.

Validation stage. As we can observe from the results of the *deployment stage*, for the current running example application, there is no need to redefine the timing requirements – the deployment stage has found an optimal solution that fulfills all application's requirements. However, if the *deployment stage* cannot find a solution, then the developer can change the requirements and employ the deployment stage again.

VI. Evaluation

In this section, we perform a quantitative evaluation to assess the deployment stage's capabilities in terms of the time required to provide optimal and feasible deployment strategies for different scenarios. We are interested in finding how certain markers like (i) the application size, (ii) the edge computing platform size, and (iii) the number of flow constraints impact the tool's performance. Considering our evaluation objective, we propose three distinct scenarios, each having a different application model and flow constraints input files. Furthermore, in every scenario, we deploy the application on multiple edge computing platforms; each target platform has a different number of available edge nodes.

We proceed by generating the three input files for each scenario. We can obtain these files using the *application modeling stage*, as proven in Section V. However, considering our evaluation objective, it is not feasible nor required to develop the applications and add timing and resource requirements manually. As a result, we randomly generate all input files using different procedures.

Application model file: generation. All considered applications have one source component and one sink component – a decision that does not alter the evaluation objective and results since an application with multiple sources and sinks only implies a higher initial number of flows. We choose a different number of components for each scenario, starting from 10 for the first scenario up to 30 components for the last one; in our case, we increase the application size by 10. We first model the component's resource requirements as a tuple, i.e., (RAM, CPU, HDD), choosing for each resource a random value between *[5, 15]* units. Next, we choose the WCET range *[l, u]* for a component by selecting a value for *l* and *u*, from *[4, 10]* ms and *[10, 12]* ms respectively. Finally, the period has a value between *[10, 30]* ms, the IP's data size is between *[30, 120]* bytes, and we define for each component a total of two input and output ports.

Edge platform file: generation. We create multiple edge computing platforms, having a size between 10 and 500 nodes. In each scenario, we gradually increase the size by 10, generate the edge platform file, and employ the deployment stage to find an application deployment strategy. Similar to the components' resource requirements, we model the available resources of an edge node as a tuple and choose for each resources a value between *[15, 30]* units. Finally, we choose for each communication link an available bandwidth between *[30, 90]* bytes/ms.

Flow constraints file: generation. We randomly generate multiple flow constraints for every application model. The procedure takes as input the total number of flow constraints defined in a file and the maximum e2e delay. We set the maximum e2e delay to a high value, i.e., *500* ms, for all flows. Choosing a smaller e2e delay does not impact the deployment stage's performance; however, it may influence its ability to find a deployment strategy if we set the e2e delay to a very stringent value. Moreover, in Section V, we have demonstrated the capability to generate deployment strategies under demanding e2e delay requirements.

After we choose the e2e delay value of a flow, we must provide the associated communication path. In our case, we define three flow constraints files for every scenario, i.e., a file containing (i) one flow constraint, (ii) three flow constraints, and (iii) a total number of five flows. Remember that the developer must define flow constraints such that it involves all communication links and components at least once. As a result, in our procedure, the first flow will always traverse the application from the source component to the sink component – involving all other components in between.

To create a communication path between the participating components, the procedure creates a pair of two components, i.e., (*src*, *dest*), starting from the source component and selects the next destination components. Next, we create a new pair using as *src* the *dest* component from the previous pair and choosing as the new *dest* a new component. The procedure continues until the destination becomes the sink component. For example, let us consider that we want to build flow *f1* from Section V. In this case, we have four components involved in *f1*, i.e., C=$\{c_0, c_1, c_2, c_4\}$. To build the flow constraint, the procedure starts from c_0 and chooses the destination c_1 forming the first pair (c_0, c_1). Next pair is formed by making c_1 as the source and choosing c_2 as the new destination, resulting in the new pair (c_1, c_2). Finally, the procedure stops with the pair (c_2, c_4), since c_4 is the sink component. For all other flows, we randomly select the number of participating components and restart the procedure.

To evaluate the performance of our deployment stage, we perform 50 deployments for each scenario. In this case, once we find an optimal deployment strategy for the current edge computing platform, we increase the platform size and attempt to find a new deployment strategy for our IoT application. In Figure 7, we present the execution time required by the deployment stage to find an optimal deployment strategy for all three scenarios. The *x-axis* represents the total number of nodes found in the target platform, while *y-axis* represents the execution time in seconds.

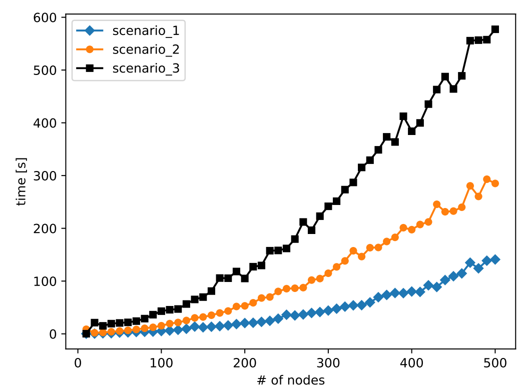


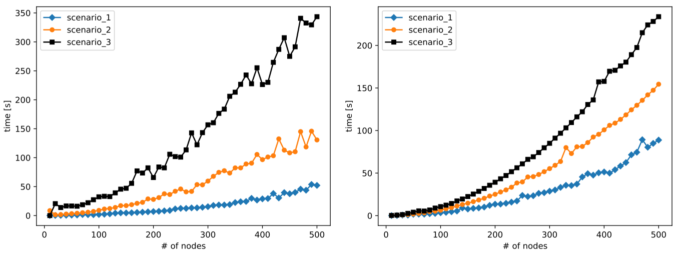
Fig. 7: Execution time of the deployment stage for different scenarios over different edge computing platform sizes, considering only one flow constraint.

In Figure 7, we show the total execution time required by the deployment stage to yield an optimal deployment strategy. However, the deployment technique consists of two different parts, i.e., (i) building the CP model and (ii) solving the model using a CP solver. As a result, we are interested in finding how much execution time each part requires (see Figure 8). In Figure 8b, we present the execution time required to generate the CP model, while Figure 8a) presents the time required by the CP solver to find an optimal deployment strategy.

In all experiments presented above, we have kept the number of flow constraints equal to 1. However, we are interested in observing the impact of multiple flow constraints on the execution time of both the CP solver and model as well. As a result, we perform the same set of experiments, increasing the number of flow constraints as well – we perform 50 deployments using a total of 3 respectively 5 flow constraints. In Figure 9, we show the execution time required to solve a model for each scenario, while in Figure 10 we show the time required to build the CP model.

A. Discussion

We have demonstrated that with the proposed deployment technique, we can successfully find an optimal deployment



(a) CP solver. (b) CP model.

Fig. 8: Execution time of (a) finding a deployment strategy and (b) building the CP model over different edge computing platform sizes, considering one flow constraint.

strategy. Contrary to how we chose the flows' maximum e2e delay in Section V, we have decided to choose a less stringent maximum e2e delay since this does not impact our evaluation results; we use the same e2e delay for all scenarios. Results of Figure 7 shows that (i) the number of nodes found in the target platform and (ii) the application's size impacts the execution time required to find an optimal deployment strategy. Breaking down the execution time, see Figure 8a and Figure 8b, we can observe that the execution time required to build the CP model or to solve it is dependent on the number of nodes and components. Note that in scenario 1, when we deploy the IoT application on an edge computing platform consisting of 500 nodes, building the CP model requires more time than finding an optimal deployment strategy. In contrast, in scenario 3, the CP solver is more demanding than building the CP model, requiring more time to find an optimal solution – a trend that continues with an increase in both application and edge computing platform size.

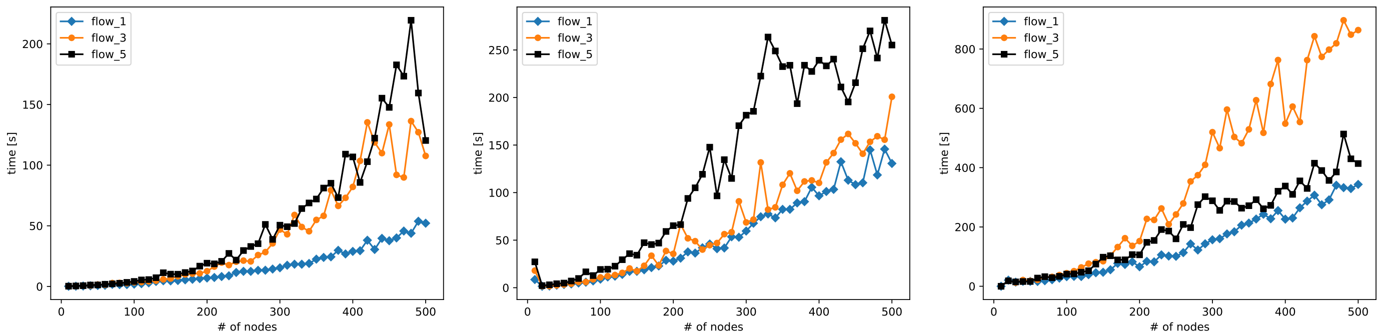
In Figures 7 and 8, we have demonstrate the scalability of our resource management technique considering the application and platform size. However, other factors impact the technique's execution time, i.e., the number of flow constraints. From Figure 10, we can conclude that the number of flow constraints marginally increases the time required to generate the CP model – by adding more flows in the CP model, we must create more variables for the added *flows constraints*. We can observe that the execution time for building the CP model gradually increases with an increase in the (i) number of nodes, (ii) number of components, and (iii) number of flow constraints – an expected behavior, since the number of model variables and constraints increases in the CP model. Comparing to the execution time seen in Figure 8b, we can conclude that the flow constraints do not have a big impact on the execution time required to build the CP model. In contrast, the number of flows has a great impact on the CP solver's execution time (see Figure 9) – the problem to be solved has become more challenging. Compared to the results seen in Figure 8a, the number of flows severely impact the execution time required by the CP solver. On the one hand, we can see that the execution time fluctuates between different deployments when the edge platform size grows – a trend that is the result of all the default optimizations the CP solver has. On the other hand, in some scenarios (see Figure 9c), the CP solver requires up to *x2.5* more time to find a deployment strategy – the addition of different flow constraints makes the problem harder to solve. As a result, the CP solver's execution time depends more on how complex the problem is. Note that the complexity of a problem depends on the node's available resources, the application size, the component's resource requirements, and the defined flow constraints. For example, in Figure 9c, we can observe that the solver manages to find a deployment strategy faster when there are 5 flows than when we have 3 flows constraints. A possible reason for this is that the overall problem complexity is higher with the addition of the three flow constraints. Therefore, we can conclude that not only the number of flows impact the execution time, but also the construction of each flow, i.e., the communication path, number of components, and the component's dependencies. However, since we built the scenarios randomly and the CP solver has its own optimizations, we cannot say with certainty why this behavior appears or the fluctuations in execution time.

Finally, one advantage of using CP for our deployment technique is the ability to allow the developer to limit the CP solver's execution time. For example, we can find a feasible deployment strategy for scenario 3, having one flow constraint, and 500 nodes (see Figure 8a), in *120* ms. By applying a time limitation we can lower the total execution time required to find a deployment strategy – lowering the time required to validate all requirements. However, it is important to mention that if the time limit is set too low, then the solver may not be able to decide if a solution exists. Hence, the solver's output will be 'UNKNOWN' – the solver does not have enough knowledge to determine if the solution is infeasible or feasible. As a consequence, the developer should pick a reasonable time for complex problems where finding the optimal deployment strategy requires too much time.

We acknowledge the high computational demands of our deployment stage when finding optimal deployment strategies for scenarios where the problem becomes too complex. We can see in Figure 7 that the deployment stage requires around 600 seconds to find the optimal deployment strategy for scenario 3. However, we argue that the execution time is not an issue since the deployment stage takes place at design time when the application is not operational.

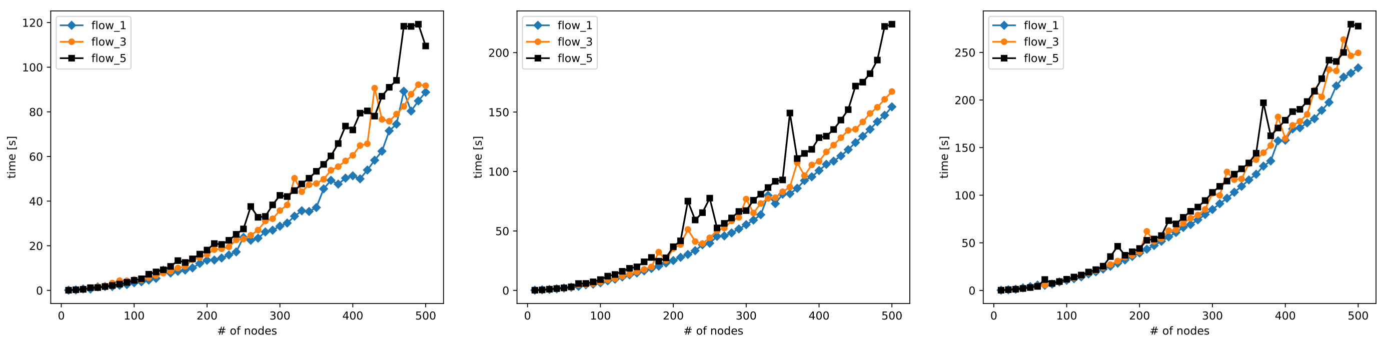
B. Challenges and Limitations

We identify two types of latency-sensitive applications that would benefit from edge computing, i.e., the hard real-time IoT applications and soft real-time IoT applications. Both applications are similar since their correct functionality relies on having a low e2e delay and meeting their deadlines. However, there is an important distinction between the two, i.e., in the case of a soft real-time IoT application, violating the deadline of a component impacts the quality of the application, compared to hard real-time applications where missing a deadline can have catastrophic events. Hence, in this paper, we focus on the development of soft real-time IoT applications offering the possibility to validate only the e2e delay set for each flow, i.e., it does not violate its maximum allowed e2e delay deadline for a certain flow. We do not provide time



(a) Scenario 1. (b) Scenario 2. (c) Scenario 3.

Fig. 9: Impact of the number of flow constraints on solver execution time considering all three scenarios.



(a) Scenario 1. (b) Scenario 2. (c) Scenario 3.

Fig. 10: Impact of the number of flow constraints on model execution time considering all three scenarios.

analysis strategies for validating the component's WCET on the host node. In conclusion, for the current paper, we can only deploy soft real-time IoT applications.

There are two main challenges for the developer during the application modeling stage, i.e., assigning the WCET and the resource requirements for each component. The former plays an important role in the overall e2e delay while the latter is critical for the deployment technique; without knowing the resource requirements of a component, the deployment technique cannot find a deployment strategy.

Finding the WCET is not a trivial task. The WCET of a component is directly dependent on the host node, i.e., the developer must know the internal status of the node (i.e., the current load and the available resources) and the location of the component. An approach to determine the component's WCET is to compute it at deployment time. We can integrate the WCET analysis into the *deployment stage* similar to how we do for the latency communication computation. An approach that automates the process of finding the WCET and simplifies the tasks of the application developer. In this paper, we assume that the developer provides the WCET using an external tool; the implementation of an automatic approach is our target for future work. Similar to the WCET computation, finding the component's resource requirements is a challenging task. One option to find and estimate these resources (i.e., RAM, CPU, HDD) is to benchmark the application on multiple edge computing platforms and take the maximum usage as an estimate.

Finally, besides finding an allocation of components to nodes, we must map the input and output virtual ports as well. There are two approaches that we can follow to achieve port mapping, i.e., manual and automatic. The former requires that the engineer performs manually the mapping of virtual ports to the host node's real ports following the deployment strategy suggestion; a scenario that is possible only if the target edge computing platform is known and has a relatively small size. In comparison, in the latter approach, the resource allocation technique is in charge of mapping the ports and the components without requiring the help of an engineer.

VII. Conclusion

In this paper, we present EdgeFlow, a new IoT framework aiming to assist the developer throughout the entire application development and deployment process. For this framework, we propose a methodology for latency-sensitive IoT applications consisting of three important stages, i.e., the *development stage*, *validation stage*, and *deployment stage*. For the *development stage*, we propose an extension of the FBP paradigm with timing and resource requirements. These requirements are crucial to the successful deployment of the application on the target edge computing platform. Further, we enable the introduction of multiple communication flow constraints, ensuring that the e2e delay of a certain communication flow does not exceed a certain e2e delay. For the *deployment and validation stages*, we introduce a new resource allocation technique capable of finding feasible or optimal deployment strategies. In conclusion, our methodology allows for a more detailed application description and assures that if the resource allocation technique finds a deployment strategy, then the strategy satisfies all application requirements.

For future work, we intend to extend our current work with techniques to analyze and compute the component's WCET; hence, we eliminate the need of introducing a WCET range, offering a more efficient deployment strategy. By having the possibility to compute the WCET of a component, our IoT framework can assist in the development and deployment of hard real-time IoT applications as well. Furthermore, there is one more important set of applications that are relevant in the edge computing context, i.e., edge intelligence applications – applications that have components that require machine learning supporting hardware (e.g., GPU and TPU) and specific data stored locally. As a result, we plan to extend EdgeFlow to (i) allow the developer to add the specific requirements to each component during the application development process and (ii) consider them during the deployment stage. Finally, we aim to provide further extensions to the FBP paradigm, i.e., add the possibility to (i) define QoS requirements and (ii) add privacy and security requirements for each component.

Acknowledgment

Research leading to these results has received funding from the EU's H2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785 FORA–Fog Computing for Robotics and Industrial Automation.

References

[1] S. Wang, X. Zhang, Y. Zhang, L. Wang, J. Yang, and W. Wang, "A survey on mobile edge networks: Convergence of computing, caching and communications," *IEEE Access*, vol. 5, pp. 6757–6779, 2017.

[2] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. June, pp. 30–39, Jan 2017.

[3] E. Ahmed, A. Ahmed, I. Yaqoob, J. Shuja, A. Gani, M. Imran, and M. Shoaib, "Bringing computation closer toward the user network: Is edge computing the solution?" *IEEE Communications Magazine*, vol. 55, no. 11, pp. 138–144, 2017.

[4] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[5] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, May 2016.

[6] C. Avasalcai, I. Murturi, and S. Dustdar, *Edge and Fog: A Survey, Use Cases, and Future Challenges*. John Wiley & Sons, Ltd, 2020, ch. 2, pp. 43–65.

[7] J. P. Morrison, *Flow-Based Programming: A new approach to application development*. CreateSpace, 2010.

[8] "Noflo," https://noflojs.org/, accessed: 2020-10-28.

[9] "Node-red," https://nodered.org/, accessed: 2020-10-28.

[10] "drawfbp," https://github.com/jpaulm/drawfbp, accessed: 2020-10-28.

[11] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier, 2006.

[12] V. D. Maio and I. Brandic, "First hop mobile offloading of dag computations," in *18th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing*, May 2018, pp. 83–92.

[13] M. Barzegaran, V. Karagiannis, C. Avasalcai, P. Pop, S. Schulte, and S. Dustdar, "Towards extensibility-aware scheduling of industrial applications on fog nodes," in *2020 IEEE International Conference on Edge Computing (EDGE)*, 2020.

[14] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296, 2017.

[15] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "Lavea: Latency-aware video analytics on edge computing platform," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ser. SEC '17. New York, NY, USA: Association for Computing Machinery, 2017.

[16] N. K. Giang, M. Blackstock, R. Lea, and V. C. M. Leung, "Developing iot applications in the fog: A distributed dataflow approach," in *2015 5th International Conference on the Internet of Things (IOT)*, 2015, pp. 155–162.

[17] X. Wang, Z. Zhou, P. Han, T. Meng, G. Sun, and J. Zhai, "Edge-stream: a stream processing approach for distributed applications on a hierarchical edge-computing system," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, 2020.

[18] X. T. Nguyen, H. T. Tran, H. Baraki, and K. Geihs, "Frasad: A framework for model-driven iot application development," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, 2015, pp. 387–392.

[19] W. Rafique, X. Zhao, S. Yu, I. Yaqoob, M. Imran, and W. Dou, "An application development framework for internet-of-things service orchestration," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4543–4556, 2020.

[20] T. Szydlo, R. Brzoza-Woch, J. Sendorek, M. Windak, and C. Gniady, "Flow-based programming for iot leveraging fog computing," in *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 2017, pp. 74–79.

[21] A. Belsa, D. Sarabia-Jacome, C. E. Palau, and M. Esteve, "Flow-based programming interoperability solution for iot platform applications," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018, pp. 304–309.

[22] R. Jain and S. Tata, "Cloud to edge: Distributed deployment of process-aware iot applications," in *IEEE International Conference on Edge Computing*, June 2017, pp. 182–189.

[23] K. Toczé and S. Nadjm-Tehrani, "A taxonomy for management and optimization of multiple resources in edge computing," *CoRR*, vol. abs/1801.05610, 2018.

[24] O. Skarlat, M. Nardelli, S. Schulte, M. Borkowski, and P. Leitner, "Optimized iot service placement in the fog," *Service Oriented Computing and Applications*, vol. 11, no. 4, pp. 427–443, Dec 2017.

[25] N. Daneshfar, N. Pappas, V. Polishchuk, and V. Angelakis, "Service allocation in a mobile fog infrastructure under availability and qos constraints," in *IEEE Global Communications Conference*, 2018.

[26] C. Liu, M. Bennis, M. Debbah, and H. V. Poor, "Dynamic task offloading and resource allocation for ultra-reliable low-latency edge computing," *IEEE Transactions on Communications*, pp. 4132–4150, 2019.

[27] A. Brogi and S. Forti, "Qos-aware deployment of iot applications through the fog," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1185–1192, 2017.

[28] V. Scoca, A. Aral, I. Brandic, R. De Nicola, and R. B. Uriarte, "Scheduling latency-sensitive applications in edge computing." in *Closer*, 2018, pp. 158–168.

[29] R. Mahmud, K. Ramamohanarao, and R. Buyya, "Latency-aware application module management for fog computing environments," *ACM Trans. Internet Technol.*, vol. 19, no. 1, Nov. 2018.

[30] X. Liu, J. Yu, J. Wang, and Y. Gao, "Resource allocation with edge computing in iot networks via machine learning," *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 3415–3426, 2020.

[31] T. Bahreini and D. Grosu, "Efficient placement of multi-component applications in edge computing systems," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ser. SEC '17. New York, NY, USA: Association for Computing Machinery, 2017.

[32] C. Avasalcai, B. Zarrin, P. Pop, and S. Dustdar, "Efficient hosting of robust iot applications on edge computing platform," in *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*, 2020, pp. 1–10.

[33] "Flow-based programming," https://jpaulm.github.io/fbp/, accessed: 2020-10-28.

[34] C. Tsigkanos, C. Avasalcai, and S. Dustdar, "Architectural considerations for privacy on the edge," *IEEE Internet Computing*, vol. 23, no. 4, pp. 76–83, 2019.

[35] F. Rao and E. Bertino, "Privacy techniques for edge computing systems," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1632–1654, 2019.

[36] Y. Xiao, Y. Jia, C. Liu, X. Cheng, J. Yu, and W. Lv, "Edge computing security: State of the art and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1608–1631, 2019.

[37] B. Zarrin, H. Baumeister, and H. Sarjoughian, "An integrated framework to develop domain-specific languages: Extended case study," in *International Conference on Model-Driven Engineering and Software Development*. Springer, 2018, pp. 159–184.