Fogify: A Fog Computing Emulation Framework

Moysis Symeonides*, Zacharias Georgiou*, Demetris Trihinas†, George Pallis*, Marios D. Dikaiakos*

* Department of Computer Science
University of Cyprus
{ msymeo03, zgeorg03, gpallis, mdd }@cs.ucy.ac.cy

† Department of Computer Science
University of Nicosia
trihinas.d@unic.ac.cy

2020 IEEE/ACM Symposium on Edge Computing (SEC) | 978-1-7281-5943-0/20/$31.00 ©2020 IEEE | DOI: 10.1109/SEC50012.2020.00011

***Abstract*—Fog Computing is emerging as the dominating paradigm bridging the compute and connectivity gap between sensing devices and latency-sensitive services. However, experimenting and evaluating IoT services is a daunting task involving the manual configuration and deployment of a mixture of geo-distributed physical and virtual infrastructure with different resource and network requirements. This results in sub-optimal, costly and error-prone deployments due to numerous unexpected overheads not initially envisioned in the design phase and underwhelming testing conditions not resembling the end environment. In this paper, we introduce Fogify, an emulator easing the modeling, deployment and large-scale experimentation of fog and edge testbeds. Fogify provides a toolset to: (i) model complex fog topologies comprised of heterogeneous resources, network capabilities and QoS criteria; (ii) deploy the modelled configuration and services using popular containerized descriptions to a cloud or local environment; (iii) experiment, measure and evaluate the deployment by injecting faults and adapting the configuration at runtime to test different "what-if" scenarios that reveal the limitations of a service before introduced to the public. In the evaluation, proof-of-concept IoT services with real-world workloads are introduced to show the wide applicability and benefits of rapid prototyping via Fogify.**

***Index Terms*—Fog Computing, Internet of Things**

I. Introduction

IoT devices have the potential to change the way we monitor, understand, and interact with our physical world, bringing it closer to cyberspace [1]. Data collected by and retrieved from IoT devices are essential in building online, delay-sensitive services in various application domains, such as public transportation [2], industrial robotics [3], and content streaming [4]. However, IoT devices have inherent constraints, such as their limited processing power, small storage capacity and poor reliability, along with the restricted bandwidth and high latency of network links that connect them to centralised Cloud infrastructures. These raise several challenges for the development, deployment and operation of delay-sensitive, IoT-oriented applications [5]. Fog Computing represents an effort to mitigate some of these challenges pushing part of the computation and storage necessary for IoT applications closer to these devices and to the network's edge. Nevertheless, this push creates new challenges in application development.

Usually, IoT applications consist of fine-grained services that communicate and cooperate with each other [6]. These services utilize compute and network resources along the path that connects the Edge to the Fog and the Cloud. The heterogeneity of Fog devices, the non-uniform network bandwidth of edge and mobile networks and the high resource variability that arises because of physical faults, bandwidth saturation, network uncertainty, energy consumption and device mobility must be taken into account in service placement as they affect the performance and reliability of applications [7], [8]. Consequently, the design, deployment, and evaluation of IoT-driven applications becomes a complex and costly endeavor for researchers and engineers, since it requires the exploration of numerous conditions and parameters through repeatable and controllable experiments on a combination of physical and virtual testbeds that are hard to configure and scale [7].

Researchers seek solutions to model and analyze the behavior of infrastructure and services [9], [10], with development kits [7], and a handful of simulators or emulators that cover different infrastructure layers, from the network [11]–[13] and up to the Cloud [9], [14]. Most of these tools are built as extensions of existing Cloud simulators and/or focus on specific aspects of Fog modeling: resource heterogeneity, service scheduling etc. Therefore, following the evaluation of a Fog-based application with existing simulation or emulation tools, an application developer needs to re-invest substantial effort to develop and deploy the application on a real testbed. Moreover, existing tools would have left many unanswered questions regarding key pains of today's IoT services, ranging from fog node mobility, fault management and energy consumption to quality of service monitoring, cost of operation, and privacy-related restrictions in data movement [15], [16].

The focal point of our work is to introduce Fogify, an interactive Fog Computing emulation framework that enables the repeatable, measurable and controllable modelling, deployment and experimentation of IoT services under realistic environment assumptions, faults and uncertainties. To illustrate the wide applicability of the Fogify framework, we **extensively evaluate** various Fog deployments originating from ML-based object detection and intelligent transportation, with real-world workloads, QoS objectives and runtime uncertainties.

Towards this, the main contributions of this paper are:

- A **comprehensive model specification** tailored to the unique characteristics of fog environments. The model expressivity enables developers to design, customize and configure complex fog deployments, including resource heterogeneity, network capabilities, operating regions, energy consumption and data movement restrictions.
- A **thorough experimentation pattern** in which "what-if" scenarios are described to fine-tune and reveal service

978-1-7281-5943-0/20/$31.00 ©2020 IEEE
DOI 10.1109/SEC50012.2020.00011

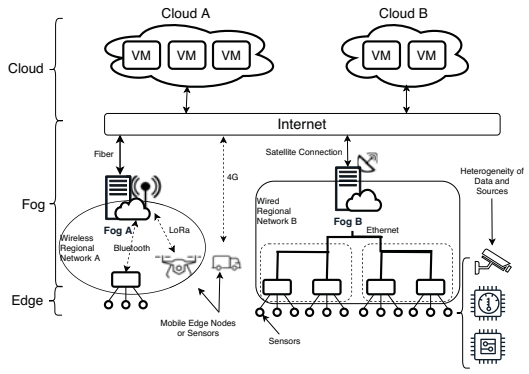

Fig. 1: High-Level Overview of an Exemplary Fog Topology

strengths and limitations before introduced in production. Specifically, developers can inject faults, entity and infrastructure downtime, network uncertainties, stress specific devices, adjust the workload, restrict data movement and adapt the entire configuration at runtime.

- An **open-source and scalable fog emulator** [17] with extensible and customizable interfaces that enables the deployment of modelled Fog environments on a local host or Cloud infrastructure. To ease the description of fog environments, Fogify adopts and extends the Docker Compose infrastructure-as-code specification to support the "fogified" model specification. When an emulated testbed is requested, the system allocates resources as isolated containerized processes, provisions network connectivity among entities, configures the infrastructure based on the modelled description accordingly, and deploys enablers for runtime assessment ("what-if" scenarios). With this, the deployment can emulate the direct behavior of an actual geo-distributed fog environment.
- A **monitoring and post experiment analysis system** that enables the measurement and evaluation of the deployment. Through that, developers inspect low-level monitoring metrics of emulated nodes, expose application-specific metrics and, finally, compose and analyze more complex models, such as energy consumption models, QoS models and monetary costs.

The rest of the paper is structured as follows: Section II presents technological aspects related to fog computing. Section III introduces the Fogify Framework, while Section IV illustrates our modeling specification and Section V describes the system implementation. A comprehensive experimentation is illustrated in Section VI. Finally, Section VII presents the related work and Section VIII concludes the paper.

II. Background & Challenges

A. Background

1) Fog Continuum: Figure 1 depicts a high-level overview of an exemplary fog topology. The lower level, denoted as the "Edge", comprises sensing devices that monitor the physical environment and which are embedded therein or within fog nodes, such as drones and vehicles. Typically, sensing devices generate raw data on a continuous basis, e.g., a sensor residing on a video surveillance device may generate 6Mbps of video content, which means the data payload easily reaches the magnitude of 500 GB in less than a week. The generated data can be modelled as an infinite timestamped data stream that the sensors transmit to more powerful Fog Nodes through various protocols, such as MQTT, Bluetooth, 4G, LoRa, etc. [15].

Fog nodes encompass compute and network components, such as gateways and access points. At the Fog layer, nodes are usually organized hierarchically. The lower sub-layers consist of lower-capacity physical devices, such as single-board computers (SBCs) or Industrial IoT Gateways [18] [19]. These devices are placed near the "edge" devices, usually in the same local network. Their role is to capture and possibly do low-cost processing on incoming data streams with the goal of easing the pressure on "last mile" networks, mediating between the Edge and "latency-far" clouds [20]. Processing data "locally" is critical for latency-sensitive services, such as road safety and autonomous driving, which require latency less than 50ms that cannot be achieved via back-and-forth communication with cloud data centers [21]. The capabilities of Fog nodes increase when moving from lower to higher levels in the Fog layer. Finally, at the top layer of the infrastructure exists the Cloud that has (theoretically) unlimited resources and stable connections between interconnected entities.

2) IoT services: usually adopt the microservices architecture paradigm, where the business logic of an application is decomposed into smaller modules (services), which run as independent processes and intercommunicate using lightweight communication mechanisms [22]. Microservice execution typically relies upon lightweight virtualization technologies, such as containerization, to cope with Fog-node Operating System heterogeneity and enhance IoT service portability [23]. Thus, each (micro-) service is realized as an independent container that runs on a physical or virtual node.

B. Design and Deployment Challenges of IoT Microservices

Application developers who wish to develop, deploy, and manage IoT services over a Fog infrastructure are faced with a number of challenges:

Evaluation of the proper devices can be costly: A wide range of physical devices can be deployed in the Fog layer including: (i) single board micro-controllers (e.g., Raspberry Pi 's, Odroids, DragonBoards) with limited capabilities (1-4 cores @ 1.5GHz with 1-4GB RAM); (ii) industrial routers and IoT Gateways (e.g., Industrial Routers, HPE GL20 IoT Gateway); or even (iii) cloudlets deployed on embedded devices with increased processing power (e.g., 2-8 cores @ 4GHz with 8-32 GB RAM). The prices of these devices vary widely, from $35 for a Raspberry Pi 3 B+ to $2000 for Cisco 800 Series Routers [21]. Therefore, from the outset, the selection of and experimentation with different devices of choice for an optimal application setup requires the exploration of numerous alternatives and a significant investment in time and money.

Configuration is time-consuming and complicated: Following the hardware selection, service operators are expected to manually setup, connect, configure, and test every physical

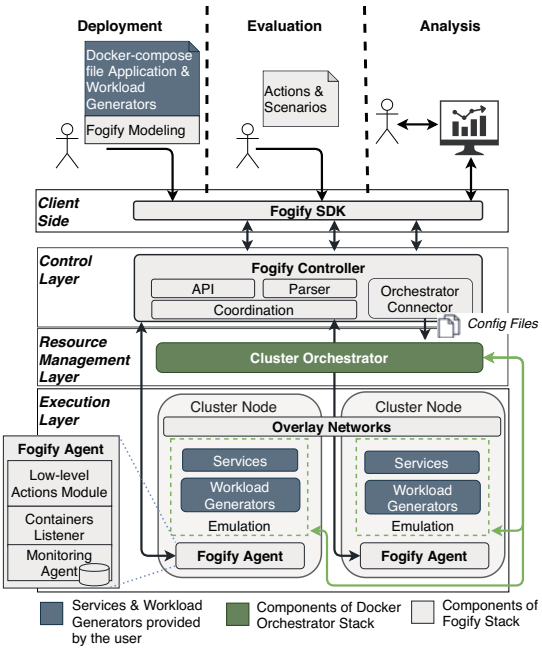

Fig. 2: A High-Level Overview of the Fogify Emulator

and virtual device of the infrastructure, although infrastructure may be spread throughout a wide geographic area or entail mobile components. The configuration and installation process itself needs to ensure that software dependencies related to virtualization and application software are respected, networks are properly instantiated, and the functionality of the system is thoroughly tested. Moreover, often, legal concerns regarding the protection of data impose the implementation of additional requirements for network device-specific configurations that ensure the geographic or administrative separation of established networks. To the best of our knowledge, there is no established and standardized way to automate this process, which is cumbersome and shifts resources from service development to infrastructure configuration.

Network heterogeneity's impact: The network path from the Edge to the Cloud entails a variety of network technologies, from optical fiber, satellite or Internet links that interconnect Fog and Cloud elements, to wireless or wired local network links that connect edge nodes [24]. Connectivity technologies at the edge vary a lot in terms of bandwidth, with wireless protocols like Bluetooth, Zigbee and LoRa offering 2 Mbps, 250 Kbps and 50 Kbps max bandwidth, respectively [25], while more stable wireless links like WiFi offering up to 95 Mbps (Section VI). The network latency between Fog nodes varies from under 3ms inside data-centers [26] to less than 15ms for one-hop wireless connections [24], to over 100ms for links between Edge and Cloud. Inside the data-center, virtual and physical machines can present intra-rack and inter-rack bandwidth of up to 10 Gbps with low network delays (1.26ms and 2.43ms respectively) [26]. Heterogeneity in networking technologies, bandwidth and latency must be taken into account in service design and placement [8]. However, the interplay between these factors and their impact on Cloud-Fog-Edge service performance has not been fully explored [19], [27] and, to a large extent, can only be investigated on top of a real deployment, which incurs significant effort and cost.

Availability concerns: Cloud operators strive to achieve very high availability with a combination of resource over-provisioning, intelligent elastic scaling, and sophisticated management tools [28]. Nevertheless, even in a stable datacenter, network device failures can cause up to 29% of significant and user-impacting incidents [29]. As we move out of the highly regulated datacenter environment and towards the Edge, operational conditions become more challenging: over-provisioning at the Fog layer is limited, proper maintenance and continuous monitoring are harder to achieve, cheaper devices have worse reliability profiles, frequent network disconnections become the norm, and devices operate in conditions that are not easily controlled, with over-heated or defective edge devices resulting in inefficient execution of IoT services [19], [20]. Finally, the adoption of virtualization increases the multi-tenancy, but, at the same time, may lead to unexpected processing interference to the co-located services [30]. In fact, failures, network uncertainties and processing interference are extremely difficult to be reproduced because they are unexpected and wide-spread in a non controllable environment.

Infrastructure monitoring requires external tools or software: Finally, aspects like QoS, energy consumption and running monetary cost related to a Fog infrastructure are crucial in the development and design phase. For instance, the power consumption of a micro-controller (e.g., a Raspberry Pi) doubles when the compute load is at max capacity, compared to when the device is in idle state [1]. Another crucial factor of energy consumption is data transfer, since it consumes 10-12% of the energy in data centers and it can reach up to 50% in Fog infrastructures [26] [31]. To capture the desired metrics the developer should deploy monitoring systems and/or even physical sensors, which may further increase configuration complexity, deployment pain or even monetary cost.

III. The Fogify Framework

A. Fog Emulation Framework Objectives

Developers of Fog-based applications, who seek to explore application performance and configuration under various operational conditions, can overcome the challenges mentioned above by using infrastructure and application emulation before undertaking the deployment and testing on real testbeds. Emulation is a process that realizes the behavior of a platform, device or infrastructure on a host device or host infrastructure, at a fraction of the real deployment cost. A Fog computing emulator has to provide an execution environment that captures realistically the conditions and behavior of a Fog application deployment, focusing on the following features:

Resource Heterogeneity: the ability to emulate Fog nodes with heterogeneous resources and capabilities.

Network Link Heterogeneity: the ability to control the link quality, such as latency, bandwidth, error rate, etc., and even reproduce node-to-node and node-to-network connections.

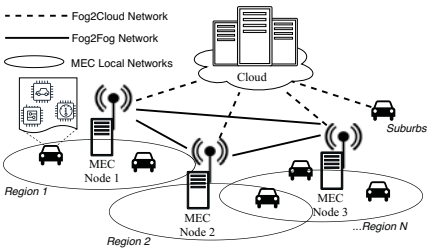

Fig. 3: Exemplary Overview of Smart Transportation Scenario

Controllable Faults and Alterations: the ability to change a running topology by injecting faults, alter network quality, and inject (varying) workload and compute resources.

Any-scale Experimentation: the emulation execution environment should be scalable from topologies with a limited number of nodes, capable to run on a single laptop or PC, to hundreds or thousands nodes, running on a whole cluster.

Monitoring Capabilities: an emulation framework must collect, manage, and process metrics from emulated Fog Nodes, network connections, and application-level information.

Rapid Application Deployment: the functional prototype of an application should demand no modifications to its business logic in order run on a fog emulation framework.

Our aim is to provide a framework that goes beyond the current state-of-the-art in Fog emulators so that users can emulate key aspects of Fog environments with minimum effort. Specifically, Fogify enables service operators to solely focus on the evaluation and testing of their services, with simplified deployment and management of scalable IoT microservices on top of the emulated environment. Emulating an execution environment involves configuring the underlying services, application deployment, monitoring, and providing the ability for run-time changes. Fogify takes care of all the above (resource and network link heterogeneity; controllable faults and alterations) and removes the burden of having to deal with these challenges from the users.

B. Fogify Overview

Figure 2 depicts a high-level, abstract overview of the Fogify architecture. The deployment starts, at the SDK level, with the description of an IoT microservices application, workload, and Fog topology. The services of an application and the workload generator are containerized services provided by the user and described in a docker-compose file. Docker-compose is a specification for "infrastructure-as-code," which helps define services and their execution parameters, prior to deployment. Fogify extends the docker-compose specification so that it encapsulates a wide variety of Fog infrastructure properties such as computing resources, network capabilities, QoS constraints, and placement policies. Users still develop their application using familiar docker constructs with the added functionality of Fogify not affecting portability. This means that a Fogify enhanced description will run in any docker runtime environment without any alterations, however users will lose the functionality offered by Fogify.

When an application is ready for deployment, the **Fogify Controller** acts as the coordinator between the SDK and the execution environment. Specifically, the Controller performs the validation of the submitted description to detect potential problems such as insufficient underlying resources. If there are no violations, the Controller provisions the Fog nodes and overlay mesh networks inter-connecting emulated devices, instantiates the (micro-) services, and disseminates (any) network restrictions to Fogify Agents at the execution layer. To achieve this, the Controller communicates via the **Orchestrator Connector** with the Resource Management Layer. The connector translates the model specification to underlying orchestration primitives for the Cluster Orchestrator. The **Cluster Orchestrator** guarantees the streamline execution of the containerized services over the Fog environment. In a nutshell, the Cluster Orchestrator manages the local Docker Engine process of each Fog node and controls the containers' execution. In the current Fogify prototype, we utilize Docker Swarm as the Orchestrator. However, our model specification is generic and capable of encapsulating other orchestrators (e.g., Kubernetes), by implementing new connectors. **Fogify Agents** are lightweight processes deployed on every cluster node. Every agent consists of three modules, each with a specific role: (i) a *server*, receives requests from the Fogify Controller and proceeds with the execution at the host node; (ii) a *listener*, listens for updates via the container socket; when an emulated device connects to a network this task applies the proper network QoS, and (iii) *monitoring*, captures performance and user-defined metrics in a non-intrusive manner.

In the runtime phase, Fogify enables developers to apply **Actions** to their IoT microservices, such as ad-hoc faults and topology changes. Faults and changes include network connectivity alterations, interference injection, device down-time, scaling actions, etc. Furthermore, developers can introduce "what-if" **Scenarios**, which comprise sequences of scheduled Actions that influence deployed devices and networks. When Actions and Scenarios are submitted, the Fogify Controller coordinates their execution with the Cluster Orchestrator and the respective Fogify Agents. Developers are encouraged to store Actions and Scenarios for future execution rather than just using the SDK for "one-off" Action submission.

A key feature of Fogify is that it goes beyond Fog emulation by also supporting the runtime **monitoring** and assessment of the deployment and scheduled scenarios. Fogify captures performance metrics in a non-intrusive manner, directly from Docker containers and users are free to define and submit app-level metric updates through the Fogify Agent listening interface. All monitored metrics are stored at the Agent's local storage. The local storage minimizes data retrieval time by applying a compound indexing scheme for efficient execution of time-range queries. Users can then extract metrics to generate useful insights about QoS, cost, and predictive analytics. This is achieved through the Fogify SDK, which retrieves local metrics to an in-memory data structure providing exploratory analysis methods that produce plots and summary statistics. Finally, since the SDK can retrieve metrics and inject actions

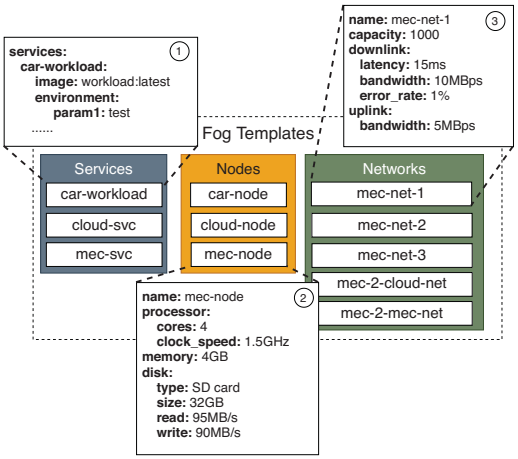

Fig. 4: Templates of Smart Transportation Scenario

to Fogify in a programmable manner, users can integrate new functionalities related to their business logic.

IV. Model Specification

The Fogify model is composed of: (i) *Fog Templates*, enabling the description of IoT services, Fog resources and networks; (ii) the *Fog Topology*, denoting how services run on the Fog testbed; and (iii) the *Runtime Evaluation*, offering run-time adaptation for the testbed. To provide interoperability between docker-compose and Fogify, we extend the docker-compose specification (v3.0) by introducing a new section named, `x-fogify`, supporting container deployments to seamlessly work in both Fogify and the docker engine.

To better understand the Fogify model specification, let us consider an example IoT application inspired from the transportation domain. Figure 3 depicts vehicles moving across city regions with each vehicle equipped with sensors monitoring critical components, e.g., tire pressure, brakes, road condition, and a GPS sensor reporting vehicle location and speed. The generated data are transmitted to nearby Multi-Access Edge Computing (MEC) nodes, which aggregate, analyse and notify local drivers for potential incidents and traffic congestion. After analysis at the MEC level, insights are propagated to a high-end cloud server for global-scale analytics and training of ML models for predicting vehicles' maintenance.

A. Fog Templates

To model the aforementioned scenario, Fogify provides the `templates` primitive. Figure 4 depicts an example, where the user denotes IoT application services and defines desired Fog node and network capabilities. In our scenario, the application is composed of 3 services. Services are self-contained binaries, made up of libraries, tools, and dependencies for the codebase. Hence, a `Service` template (Fig. 4 ①) is used to represent the application services (*mec-svc* and *cloud-svc*) and the workload generator emulating the set of aforementioned sensors (*car-workload*). Next, the `Nodes` template is used to define Fog resources. In our scenario we have 3 different types of nodes (*car*, *cloud*, and *mec*). The `Node` template allows

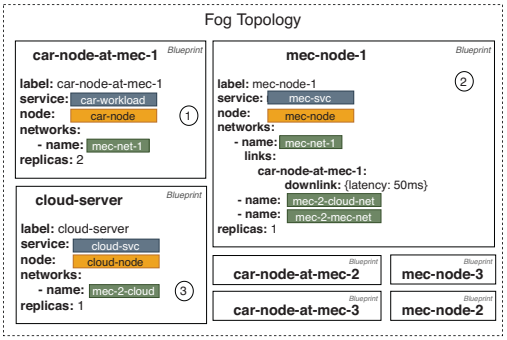

Fig. 5: Examples of Fog Topology Model

users to describe the resource characteristics of a physical or virtual host, including properties for the `Processor`, `Memory`, and `Storage`. For example, the *mec-node* (Fig. 4 ②) has 4 cores at 1.5GHz, 4GB RAM and 32GB disk.

The `Network` template is used to define inter-connecting mesh networks and links among Fog nodes. Fog nodes can be directly connected via an emulated physical link and/or belong to the same network administrative domain. The `Network` template provides the `Capacity`, which restricts the number of connected devices on a network, and a set of default `Network QoS` features. In regards to QoS, one can specify `Downlink` and `Uplink` connectivity properties. We note that although our focus is not on specific network protocols (e.g., bluetooth, LoRa, etc), Fogify provides pre-compiled templates for various protocols with the most critical QoS constraints defined, such as `Latency`, `Bandwidth`, `Packet Drops`, `Error Rate`, and `Reordering`. Moreover, Fogify enables users to denote the latency distribution of a network link. Specifically, Fogify provides an extensible collection of latency distributions, including *uniform*, *gaussian*, *pareto* and *paretonormal*. Furthermore, users are able to upload network trace files via the SDK, so that Fogify adopts a custom latency distribution. Since the Fogify model is not bound to specific technologies but only to QoS constraints, it can easily describe even connectivity for technologies that have yet to-be tested. Figure 4 ③ depicts the definition of the *mec-net-1* network, which has maximum capacity 1000 nodes, and default network QoS are: for downlink bandwidth 10Mbps, network delay of 15ms and 1% error rate, while the uplink bandwidth is 5MBps.

B. Fog Topology

Templates only describe the provisioning of services and resources, lacking information on how a Fog topology is realised and interconnected, and where services are placed. So Fogify provides the `Topology` primitive, enabling users to specify a set of `Blueprints`, which is a combination of a `Node`, `Service`, set of `Networks`, `replicas` and a `label`. For instance, the `blueprint` of Fig. 5 ① materializes two vehicles (*replicas: 2*) that are connected to the *mec-net-1* network and run the *car-workload* service. Similarly, the user describes all components from the application scenario illustrated in Figure 3. The combination of services, nodes and templates allows users to create more complex topologies.

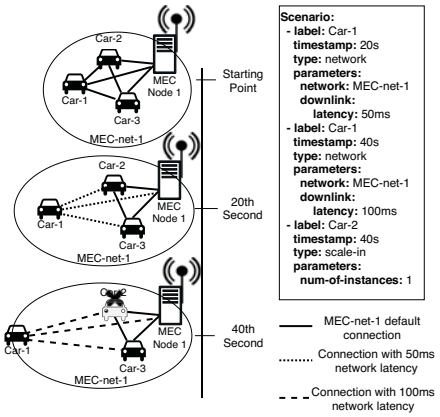

Fig. 6: Moving Node & Fault Scenario

Furthermore, `Blueprints` support the overriding of (i) network-level QoS properties; and (ii) the properties of specific network links. Altering network-level QoS will affect the traffic between all Fog Nodes of the network, whereas link-level overrides will affect only the network traffic of the specific link. For instance, in Fig. 5 ②, the traffic from *mec-node-1* to the *car-node-at-mec-1* in the Network *mec-net-1* will experience 50ms network latency while the network QoS for the rest of nodes will be the default network QoS of *mec-net-1*.

C. Runtime Evaluation Model

The aforementioned model abstractions enable the rapid prototyping of emulated Fog deployments. However, Fog deployments are usually neither statically provisioned nor stable. This instability highly impacts the execution of IoT services, especially in uncontrollable domains. Fogify addresses this challenge by providing runtime Actions & Scenarios.

An `Action` is a process that changes properties of a running Fog Topology. To apply an action, Fogify uses the Fog node `label`, the `type` of Action and any action parameters. There are three Action types, namely: `scaling_action`, `network_action`, and `stress_action`. A `scaling_action` is either horizontal or vertical. Horizontal scaling can be applied to mimic a virtual scaling action, such as the spawning of a containerized service or the failure of a Fog node. Vertical actions emulate transient failures or fluctuations of the processing capabilities of a Fog node. Furthermore, a `network_action` refers to changes in network QoS. Specifically, a network action can be seen as an ad-hoc network fault, physical network interference, connectivity uncertainty, or even a malicious attack. Finally, a `stress_action` provides the ability to simulate workload interference on a running Fog node. This is useful to evaluate the behaviour of a service during workload variation and/or interference from other services.

To enable the repeatable execution of a series of Actions, Fogify provides the Scenario primitive. A `Scenario` is a sequence of time scheduled actions that Fogify will execute to emulate more complex user-driven experiments. Figure 6 depicts a scenario of a moving node (*car-1*), moving away from a base station, with a two step connectivity degradation. After 20s from the experiment start, the moving node becomes distant from the others, thus the experienced network delay becomes 50ms. The node continues to move away with the same speed, so at the next 20s the experienced network delay becomes 100ms. At this point, let us assume that *car-2* has an accident that destroys its compute unit so the emulated node should be removed from the testbed, thus the scenario performs a `SCALE_IN` action. Scenarios are written in YAML format, with users executing them at anytime during the evaluation. The latter enables experiment reproducibility, aiding the rapid evaluation on different fog topologies.

V. Fogify Implementation Details

A. The Fogify SDK

The Fogify Controller exposes a REST API for interacting with the emulated testbed. We provide a python SDK with programming primitives for interacting with the Fogify Controller. Table I summarizes available SDK and API calls. The Fogify SDK provides the ability to submit IoT service descriptions adopting the Fogify model specification, manipulate service execution by applying actions and submitting "what-if" scenarios, extract real-time monitoring data and assess running deployments. Furthermore, the Fogify SDK has a set of built-in analytics functions applicable on monitoring data, as well as, plotting functionality for easing metric examination. With the interactive changes of the topology and the inspection of changes' effects, Fogify SDK improves the programmability of the platform and gives developers the opportunity to implement their own components with more complex behavior, such as external modules or connections with their tools.

B. Computational Resources Management

Container virtualization is used as the execution environment for emulated Fog nodes, due to their low computing overhead and fast instance spawning time. Docker provides a standardized way for building, sharing and executing containerized applications. Specifically, docker images are files that package all the service's code, dependencies, artifacts, etc. Docker containers are the running instances of them. Thus, we consider that the containers are the building blocks of a microservice application and, in our framework, represent the emulated execution environment for a service.

Realistic Fog node emulation requires the isolation among operating services and constraining computing resources. On the one hand, container-based virtualization offers the desired level of isolation by executing a containerized service as an isolated process in a user-space on the host OS. The Docker engine abstracts the related reachable resources from every running container such as process IDs, network interfaces, hostnames, etc. On the other hand, the Docker engine restricts the processing capabilities of running containers by utilizing Linux Control Groups (cgroups). A cgroup is a Linux kernel mechanism managing OS resource allocation for specific processes (e.g., CPU, RAM). Even if tools for resource constraining exist, the mapping of the desired emulated restrictions to

SDK Function (API Path)	Method	Description
Control functions		
deploy (/topology/)	POST	Deploys a specification file
undeploy (/topology/)	DELETE	Remove the fog infrastructure
Actions & Scenario functions		
stress (/action/stress/)	POST	Inject stress workload on fog instances
h_scaling_up (/action/scaling/horizontal/)	POST	Starts N fog instances
h_scaling_down (/action/scaling/horizontal/)	POST	Stops N fog instances
v_scaling (/action/scaling/vertical/)	POST	Alters the processing capabilities of a running node
update_network (/action/network/)	POST	Updates network QoS
delay_distribution (/delay-distribution/)	POST	Creates a new delay distribution which will be utilized in the experiments
scenario_execution	-	Executes a scenario (currently supported only by FogifySDK)
Monitor functions		
get_metrics (/metrics/)	GET	Retrieve metrics
clean_metrics (/metrics/)	DELETE	Removes all stored data

TABLE I: Fogify's Python SDK & API reference

low-level host characteristics is not always an easy task. For some resources, like memory, the mapping is straight-forward. For instance, an Edge device with 2GB RAM will be mapped to a 2GB constrained container. However, for resources like CPU, the mapping is not so obvious. Let us assume that we would like to emulate a Fog node equipped with 2 cores @ 1.5GHz on a host equipped with a more powerful processor, e.g. 10 cores @ 3.6GHz. The question that arises is: *How to restrict container computing capabilities to emulate the computing power of a Fog node?* To address this, we introduce a generic CPU power metric, the cumulative clock rate (CCR), which is the number of clock cycles a processor executes per second multiplied by the number of its cores:

$$CCR = Cores \times Cycles \tag{1}$$

With the host and emulated node CCR, we compute the CPU rate between them which illustrates the portion of the host processor power that will be occupied by the emulated node:

$$CPU_{rate} = \frac{emu\_node_{CCR}}{host_{CCR}} \tag{2}$$

Fogify translates the CPU_{rate} to the container cgroup CPU quota. Specifically, cgroups restrict the container CPU allocation by specifying how long the container can run over a fixed interval less or equal to the host clock speed. In a nutshell, the Linux Process Scheduler measures how long the container has run in the current interval, and when its total runtime reaches the quota, the container is throttled. When a new interval starts, the scheduler renews the container runtime. Thus, we consider the container compute interval equal to the CPU_{rate}.

Fogify enables the configuration of an *over-subscription* percentage for compute resources. Thus, when a user opts for a 20% over-subscription, if sufficient resources exist, the emulated nodes occupy their CPU_{rate} , otherwise, they share at most 20% of the CPU time. In this way, Fogify emulates large topologies even on a single host, permitting via a user-defined parameter a tradeoff between compute performance

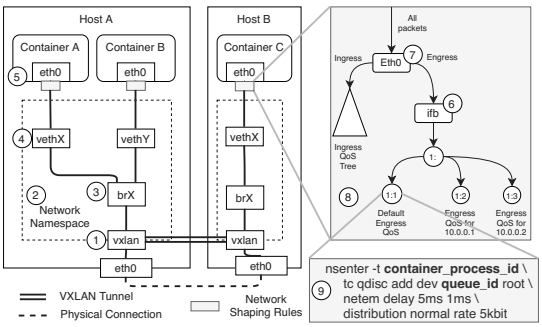

Fig. 7: Low-level network shaping of Fogify

and experiment scale. In conclusion, Fogify creates all cgroups quotas, builds the configuration and submits it to the Cluster Orchestrator. At this point, we note that the emulated deployment will not be feasible only if the underlying resources do not meet the minimum provisioning requirements, bounded by the ability of Fogify to run Docker on each fog node.

C. Network Shaping

To establish connectivity between containers on different hosts and emulate isolated networks, Fogify creates an overlay mesh network for each network description (Fig. 7). Every overlay network is realized as a Virtual eXtensible LAN (VXLAN) ① which is extended in software across the underlying network, connecting separate hosts while using an http routing mesh. To achieve the connectivity between the emulated Fog nodes, Fogify creates a new network namespace ② on each host. A network namespace is comprised of an isolated virtual network stack and a virtual bridge ③. The virtual bridge acts as an interconnecting interface between the VXLAN tunnel endpoints of different hosts. When Fogify connects a container to a network, it creates a virtual ethernet adapter (veth) connected to the local virtual bridge ④. Inside each container, there is an emulated Ethernet endpoint (eth) ⑤, one for each network, and this endpoint is mapped to the external veth. It is on these Ethernet endpoints that Fogify places and applies the different user-defined QoS network rules by utilizing various traffic flow tools such as the Linux nsenter tool [32], the token bucket filter [33] and tc-tool [34]. To achieve this, Fogify enables through the Linux kernel of the container, an intermediate functional block device, or simply ibf interface ⑥, that handles and separates all traffic filtering for both uplink and downlink network connections ⑦.

When the uplink and downlink traffic are separated, Fogify enables the QoS enforcement, redirecting packets to the respected network queues of the container. Fogify utilises *classful queuing disciplines* (qdisc) to filter and redirect packets to a particular network QoS queue. The classful qdisc allows the definition of a tree-based structure ⑧, where the root is the network interface (physical or virtual) and nodes are classes. When traffic reaches a network interface, it traverses the tree nodes by following the packet filtering. Filters redirect the packet flow to mapped nodes, found on the leaf level of the tree. Every leaf outputs the traffic to a network queue

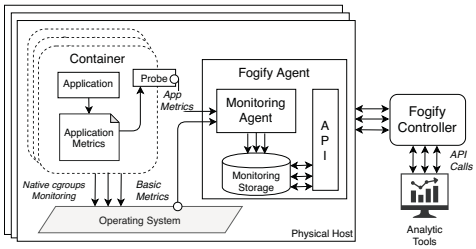

Fig. 8: Monitoring System Overview

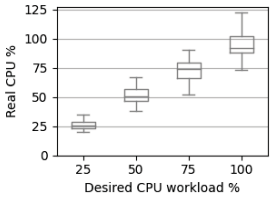

Fig. 9: Compute Realism

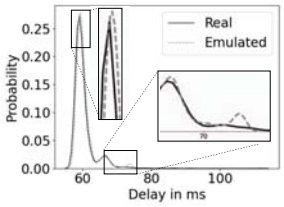

Fig. 10: Latency Distribution

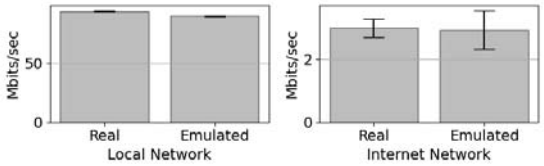

Fig. 11: Network Link Bandwidth Realism

with the desired QoS characteristics. To be manageable, each queue has an identifier assigned by Fogify. With this identifier, the Fogify Agent employs a network rule on a specific queue to emulate network QoS (i.e., latency, bandwidth, drop rate). Fig. 7 ⑨ depicts an example of a network rule that Fogify applies to a network queue. This rule will affect the network queue with `queue_id` that is placed in the network namespace of the container with process identifier `container_process_id`. The latter restricts bandwidth up to 5 kbps, assigns a per packet latency of 5ms with 1ms deviation, with values following the normal distribution.

The emulation of network characteristics becomes even more complex if the emulated nodes are placed on different hosts. For example, in Figure 7, containers A and C represent Fog Nodes A and C, respectively, and are connected to the same network. In this case, the Fogify Agent, located within FN-A on Host-A, is not provided with the essential information regarding FN-C. In an attempt to overcome this situation, Fogify Agents inspect the modeling definition of a network link in cases when a new container is spawn. If any link is likely to influence an instance that is placed on a different host, the Agent sends a notification to the Fogify Controller. The Fogify Controller disseminates the network update to the appropriate agents. In the example, the Host-B Agent is about to send a notification to the Controller and, subsequently, the Controller will send an update to the Agent of Host-A. For Fog topologies comprised of more than one network, the same process is adopted for each mesh network.

D. Monitoring

Fogify embraces the sidecar architecture paradigm to perform seamless and efficient monitoring [35]. The sidecar paradigm enables Fogify to add monitoring capabilities to an IoT service with no additional configuration to the actual business logic. An overview of the Fogify monitoring flow is depicted in Figure 8 with an agent located on each host and operating as a long-running process that retrieves monitoring data. Both performance and app-level metrics, along with their periodicity, can be customized accordingly by the user.

Fogify captures performance metrics directly from the containerized process by inspecting the containers' *sysfs (pseudo-files)*, without any interruption or overhead to running services. Pre-defined metrics available by Fogify include CPU time, memory usage, disk I/O, and network traffic, among others. Users can expose application-level metrics for their IoT services. This is achieved by exposing metric updates, in JSON format, to a file (*fogify.metrics.json*) via the container root directory interface. Fogify will then retrieve them via a lightweight monitoring probe. The monitoring agent calls the probe in each interval, with the probe "reading" metric updates. When the agent retrieves the data, it stores them to the monitoring storage.

Since the user can define metrics that are not known beforehand, we have created an extensible and metric agnostic storage scheme that consists of two entities, the `Record` and the `Measurement`. A `Record` has two fields, the Fog node `Label`, as described in the modeling section, and the `timestamp` of the interval that the measurements are captured. Furthermore, `Measurement` has two fields: a `Metric Name`, that characterises both basic and application-level metrics, and its `value` that is a numeric value that the metric has at the specific timestamp. One `Record` has multiple measurements so there is a one-to-many relation between them. This allows Fogify to have an arbitrary number of monitoring metrics in each interval. Furthermore, we set a compound range index (identifier-timestamp), which increases the performance in time-range queries for specific instances that are the most common queries in our system. Finally, the stored data can be retrieved via the Fogify REST API or the SDK. A monitoring metric request can include time-range queries and filters for specific services or instances. The fast filtering minimizes the response time of Fogify and reduces the size of data that needs to be processed from analytic tools.

VI. Evaluation

This section introduces a study evaluating the Fogify feature set. First, we provide experiments to assess Fogify's ability to realistically emulate computing resources, network links and data processing tasks on top of different testbeds. Next, our experiments focus on the deployment and runtime scenario assessment of an IoT microservice-based application utilizing a real-world workload for geo-distributed vehicle tracking.

A. Realistic Emulation

1) Computing Performance Emulation: In this experiment, we investigate how Fogify handles user-desired computing

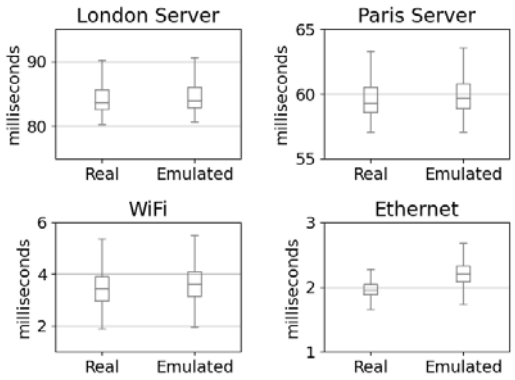

Fig. 12: Network Delays of different connections

preferences by emulating a Fog node equipped with 4 cores @ 1.4GHz and 1GB RAM (Raspberry Pi 3 features), on top of a host featuring 16 cores @ 2.4GHz and 16GB RAM. According to the CCR metric (Section V), the emulated Fog node will occupy 15% of the host's total processing power, ample for running cpu-intensive workloads. We inject CPU stress actions accounting for 25%, 50%, 75% and 100% CPU utilization of the emulated Fog node for a duration of 5min for each action. Figure 9 depicts the results by providing a box-plot per stress test where the median CPU usage is denoted by the line in the box, while the box length extends between the first and third quantile. From the results, despite small deviations, we observe favorable results when comparing the desired CPU with the emulated utilization for the 25%, 50% and 75% stress tests (medians are 25.80%, 50.44%, 74.31%). A slight performance deviation is observed for the 100% CPU stress test, where the median of the emulation is 92.29%. Moreover, since the host environment is more powerful than the emulated node, Completely Fair Scheduler (CFS), which is the default linux kernel CPU scheduler responsible for limiting CPU utilization, can "borrow" CPU cycles from future periods, so we even observe some utilization values over 100% of the available quota. This test shows that *the Fogify emulator can mimic the utilization of emulated computing resources quite accurately, with only small performance deviations observed on emulated Fog nodes for workloads approaching 100% CPU usage*.

2) Network Link Emulation: In this experiment, we investigate how accurately does Fogify emulate network link behavior. First, we setup a *real* infrastructure comprised of the following: (i) a Raspberry Pi 3 with 4 cores @ 1.4GHz and 1 GB RAM connected to a router via Ethernet; (ii) a laptop equipped with an i7-6500U processor with 4 cores @ 2.5GHz connected via WiFi with the same router; (iii) a VM allocated from the Azure datacenter in London with 4 vCPUs and 8GB RAM; and (iv) a VM allocated from the AWS datacenter in Paris with the same capabilities. The router, raspberry and laptop are physically located in the same room. Next, we measure, with perf3 [36], the network bandwidth of the laptop-raspberry (local network) and laptop-London (internet) connections. The mean measured bandwidth is 94Mbit/s and 3Mbits/s , respectively. Afterwards, we proceed with emulating the aforementioned topology using Fogify and measuring network link bandwidth as emulated by Fogify. Figure 11 depicts the comparison of the real and emulated deployment for both the local network and Internet connections. From this figure, we observe that the emulated links' bandwidth, 89.1Mbit/s and 2.92Mbit/s respectively, follow the real connections with only modest deviations ($\sim$3%).

In the next experiment, we focus on replicating real delay distributions. For this, we use the Linux ping tool [37] over 2000 intervals and capture the response time between: (i) the laptop and London VM; (ii) the laptop and Paris VM; (iii) the laptop and local router via wifi; and (vi) the raspberry and local router via Ethernet. Afterwards, we inject the captured delay traces distributions to Fogify, and reproduce the experiment over the emulated topology. Figure 10 depicts the probability density of the real and emulated delay traces for the Paris VM. We clearly identify that the real and emulated trace are very close with the emulated trace missing only some extreme values that the real trace includes (in particular, 14 values with latency just above 75ms). To further evaluate the delay replication, Figure 12 features the box plots of all traces (real and emulated). We see that the emulation follows the real network delay, despite a small overhead ($\sim$200us). This overhead is due to establishing the virtual network stack and interfaces with the desired rules (Section V-C). However, for very low Ethernet delays ($\sim$2ms) the 200us account for 10-12%. Hence, when emulating low-latency links this overhead takes an evident toll which becomes more evident as we approach the limits of the physical network. In conclusion, *Fogify emulation achieves near to real-world network link capabilities, with only outliers not captured in emulated network traces and a slight overhead in low-latency connections*.

3) Data Processing Emulation Realism: For this experiment, we benchmark an application that conducts object classification using a deep learning model. Specifically, we evaluate two deployments: (i) Cloud-only, where the application sends images from the user device to the Cloud for classification; (ii) Edge-Cloud, where the application sends images to a local edge device for pre-processing (image resizing) prior to the cloud classification. For the classification, we adopt the YOLOv3 toolkit for fast object detection [38]. We specifically note that, *to run the classification process over Fogify no alterations to the codebase are required*. As infrastructure, we use the aforementioned London Server as the Cloud service, the Raspberry as the Edge device and the laptop as the user device. The workload generator submits to the application 2 images over 40 consecutive time intervals, with UltraHD and 1080p resolution respectively. For evaluation, Fogify monitored the mean processing and network time.

Figure 13 depicts the results for the different scenarios when deployed over the real and emulated infrastructure. In particular, Figure 13 depicts both the processing and network overhead for all scenarios. For the Edge-Cloud deployment, the edge processing time is depicted as well. From the results one can immediately identify that *the emulation results closely follow the real measurements with a deviation of less that 5%*.

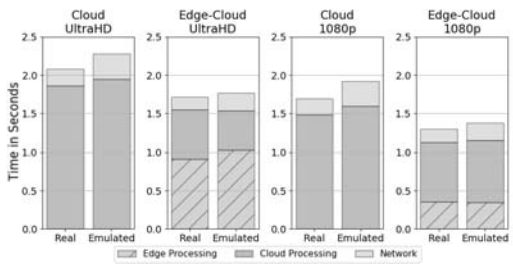

Fig. 13: Application in different deployments

The only exception is the network overhead increment in the Cloud-only scenarios which resulted in a 8% deviation of the overall experiment time. This limitation is due to bandwidth emulation, where in the previous experiment we showed that emulating low-latency links presents a performance penalty. Consequently, when the volume of the data increases, the overall network overhead becomes slightly more obvious.

B. IoT Microservice Evaluation

Next, we evaluate the use of Fogify in the emulation of an envisioned IoT application that is driven by real-world data. We assume a scenario with a fictional taxi-cab company that would like to collect and analyse location-based data from their taxi fleet in an attempt to optimize, in real-time, operations, cab routing and subsequently increase revenue. For this, the company purchases five MECs and places them in five different regions inside a city (*region-{1-5}*), with taxis sending sensed data to nearby MEC. Taxis can also travel in the *suburbs*, where there are no reachable MEC, and in which case data are forwarded directly to the Cloud via mobile internet. An abstract overview of the topology is described in Section IV and is depicted in Figure 3. The MECs are used for data pre-processing and for producing in-time area-level analytic insights based on the current area-operating taxis (e.g. *the number of occupied taxis per neighborhood*). The entities which take place in the experiment are the following:

- **IoT workload-generator**: reproduces the workload generated by each taxi, emulating the sensing data dispatch to a nearby MEC or to the Cloud. Unless stated otherwise, we use *workload-generators* in two different 'profiles': one for taxis moving in the city connected with a MEC, and one for taxis moving in the suburbs and communicating via mobile internet. Both generators produce a static rate of 10 req/sec. The Car node definition is presented in Fig. 5 ①, and each Car node is emulated as a Fog node with 1 core @ 700MHz and 256MB RAM;
- **MEC Node service**: captures incoming requests, computes insights, in this use-case area-level analytics, and forwards the results to Cloud. The MEC node blueprint is illustrated in Fig 5 ② with its capabilities are shown in Fig. 4 ② (4 cores @ 1.4GHz and 4GB RAM).
- **Cloud server**: gathers area-level data and computes the final results. Fig. 5 ③ depicts the node definition (8 cores @ 2.4GHz and 8GB RAM).

MECs and taxis placed in the same area, communicate with each other through regional networks (one per area), while the

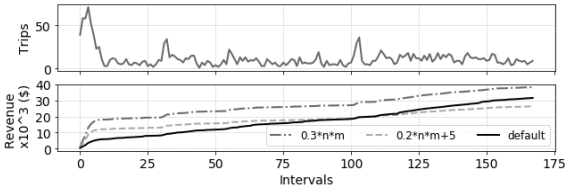

Fig. 14: App-level metrics extracted and compiled by Fogify

MECs and suburb taxis communicate with the Cloud through mobile connections. The definition of the regional networks (e.g. *mec-net-1*) is depicted in Fig. 4 ③. The regional network latency is 15ms and the maximum bandwidth is 10Mbps, while the Edge-Cloud connection has 100ms delay and up to 5Mbps bandwidth. According to the parameters of the emulated application, the Cloud service queries the MECs for updated area-level insights to produce a global overview every 5min. A MEC is considered unreachable when the delay of any sensing (or query) response exceeds 10s.

We utilize a *publicly available, real-world dataset* with 44M routes from 13.5K NYC taxi cabs in the first half of 2018 [39]. Each vehicle is equipped with GPS tracking to record data for each route. The dataset includes passenger number, charged amount, tip, pickup/dropoff location, etc; in total, each record captures 18 data items. Based on this application, we explore the *usability*, *extensibility*, and *observability* of Fogify with 4 different scenarios. Initially, we show an example of the extensibility of Fogify in terms of app-level metrics (Scenario 1). Then, we examine the monitoring and profiling capabilities of Fogify (Scenario 2) and for the last two scenarios we focus on (i) network uncertainties (Scenario 3), and (ii) scaling actions and workload alterations (Scenario 4). In terms of deployment effort, we were able to deploy the IoT application with no changes to its codebase and a less than 30% extra configuration written on application's docker-compose file. *With Fogify model composition the deployment effort is substantially lower compared to a real fog deployment.*

1) Scenario 1 - Dynamic Taxi Pricing: Suppose that the business analyst of the taxi-cab company wants to explore alternative pricing models based on realistic data about taxi rides. To produce a pricing model, the analyst needs to collect the total number of trips per hour for each operating region and for different operational scenarios. Fogify can compile analytic insights from data produced by the emulation of realistic scenarios, incorporate insights into pricing models and visualise the results. To this end, we define and deploy on Fogify the previous topology and the application components that collect sensor data from the taxi-cabs, push them to Fog nodes and the Cloud, and implement alternative pricing models. The emulation is driven by realistic data fed into the emulated application by a *workload-generator* that reproduces the real dataset for a week (01/01-08/01). The workload-generator exposes the total number of trips per hour that occur in region-1 as an app-level metric that is fed into alternative pricing functions. The top plot of Figure 14 depicts the number of trips per hour and the bottom plot the running revenue of 3 different pricing models, namely: (i) $0.2 \times n \times m + 5$, where

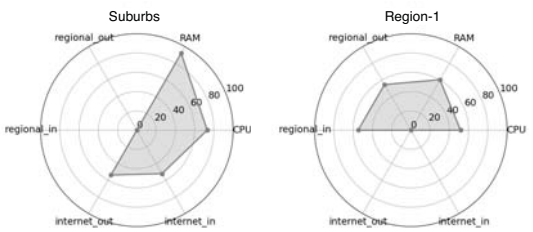

Fig. 15: Fog node profiling when moving between regions

n and m is the number of trips and the cumulative distance of all trips, respectively, increased by a static initial charge ($5) (ii) $0.3 \times n \times m$ that follows the same notation and (iii) the default pricing model. In conclusion, taxi-cab *operators can easily employ Fogify to produce and evaluate cost models and analytic insights under a variety of operational conditions, implementing adequate application-level metrics*. This analysis does not require deployment of the models onto the actual infrastructure. Subsequent integration of the model of choice onto the infrastructure is straightforward and does not require additional, significant implementation effort.

2) Scenario 2 - Fog Node Profiling: In this scenario, we investigate the impact towards Car node resource utilization, as Cars move between areas w/o MEC communication. Figure 15 depicts the mean resource utilization for a Car node when operating in the *suburbs* and then moving to *region-1*. From this, we immediately observe that despite a node being imposed to the same workload, *how sensed data are disseminated to recipients significantly impacts resource utilization*. In this case, a Car in the suburbs does not have the luxury of region-level data offloading to a MEC and thus, must propagate data to the cloud via a mobile Internet connection. However, moving to region-1, we see a drop in both memory (56%) and CPU utilization (36%) by propagating data to the cloud via a regional connection established with the region-1 MEC. *Node profiling insights are highly beneficial to engineers not just for capacity planning, but also for optimizing service and resource placement (e.g., where and how many MECs to place).*

3) Scenario 3 - Network Uncertainties: In this scenario let us assume that the emulated topology after 2min of proper operation, presents a problem with the region-1 MEC causing a temporal network latency of 2s for a 3min interval. Next, the MEC returns to the desired state for 2min, however, the problem reoccurs with the network latency reaching 20s for another 3min before returning again to normal state. Afterwards, technicians disconnect the MEC for repairing. Figures 16 and 18 illustrate the scenario timeline (including Fogify Actions) and runtime resource utilization of the topology components. Initially, we identify a throughput degradation between the 25-30th intervals in both *MEC.region-1.local-net* and *Car.region-1.local-net* plots. However, after the 30th interval, network traffic increases to the previous level, since the latency does not exceed the pre-defined 10s disconnect limit. Another observation is that the return to normal state results in a data transfer spike (60-65th intervals) for the MEC and Car nodes due to queued requests. On the 75th interval, when the network

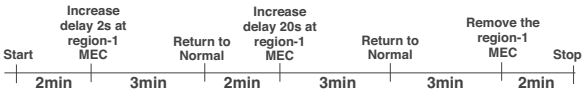

Fig. 16: Scenario for Network Uncertainties

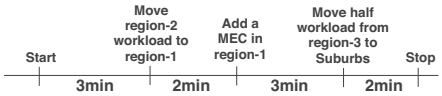

Fig. 17: Scenario for Scaling Actions and Workload Changes

latency is boosted to 20s, every Car node request to the MEC fails. Since requests by Cars in region-1 experience delays over 10s, they transmit sensed data directly to the Cloud (75-110th intervals *Car.region-1.internet*). In turn, we observe very low traffic on the *Car.region-1.local-net* which is reasonable due to the requests exchanged with the Cloud over the internet. Moreover, the *Cloud.cpu* and *Cloud.internet* plots depict the effect of directly receiving requests from the taxis operating in region-1. Finally, on the 130th interval of the experiment, the technicians remove the MEC in region-1. In this phase, we observe a similar increase in the cloud CPU load and network traffic as in the previous period (75-110th interval). *In conclusion, with network alterations and fault injections, users comprehensively evaluate the execution of their services under extreme conditions while identifying unpredictable outcomes of imposed uncertainties to the service behavior.*

4) Scenario 4 - Scaling Actions & Workload Changes: In this scenario let us assume that during events (i.e., marathons), a region's taxis are restricted to neighboring areas. This situation is modeled by moving region-2 taxis to region-1 after 3min of normal execution via a *scale-in action on region-2's workload-generator and a scale-out action on region-1's workload-generator*. Due to increased traffic, the Operations Team decides to add another gateway (*scale-out*) for the region-1 MEC at the 5th min. Finally, on the 8th min, half of the workload of region-3 is transferred to the suburbs due to an accident in region-3's main highway. Figures 17 and 19 depict the scenario timeline and the resources utilization of the topology components. By inspecting *MEC.region-1.cpu* and *MEC.region-1.local-net* plots, on the 30th interval (load moved from region-2 to region-1), the utilization in both are increased. Furthermore, the region-2 MEC for the rest of the experiment has zero network I/O in both *MEC.region-2.local-net* and *MEC.region-2.internet*, whereas *MEC.region-2.cpu* gives us the baseline utilization for the MEC service. On the 50th interval, the scenario introduces (*scale-out* action) a new instance of the region-1 MEC and, as we observe on *MEC.region-1.cpu* and *MEC.region-1.local-net*, the CPU utilization decreases. Finally, on the 90th interval, the increased number of data, which goes directly from the suburbs to the cloud, causes additional cpu and network utilization to the Cloud (after 90th interval on plots *Cloud.cpu* and *Cloud.internet*). *To this end, down-time injections, scaling actions and varying workload released insights about service performance and resources utilization.*

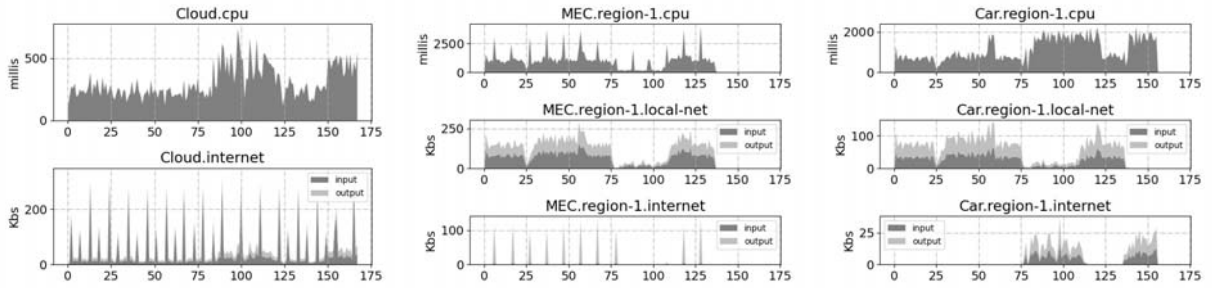

Fig. 18: Metrics from Network Uncertainties Scenario

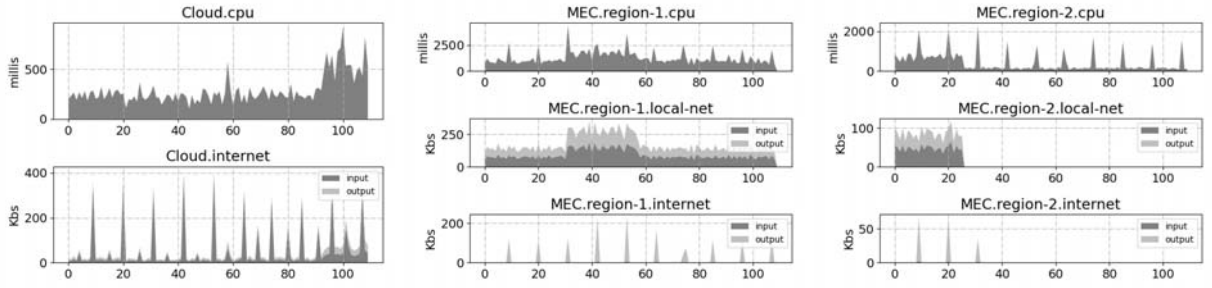

Fig. 19: Metrics from Workload and Scaling Scenario

VII. Related Work

D' Angelo et al. [10], identify scalability, capacity planning, network configuration, "what-if" scenario assessment and security testing as key challenges involved in prototyping IoT simulation and emulation tools. To date, the majority of IoT simulators are implemented as direct extensions of cloud simulators. For example, iFogSim [9], EdgeCloudSim [14], and IoTSim-Edge [40], are based on CloudSim [41], a popular open-source cloud simulator. iFogSim [9] provides IoT device and fog resource modeling to enable the simulation of scheduling policies based on several QoS criteria. Similarly, FogTorchPi [8] is an IoT simulator that enables fog resource modeling and the expression of QoS criteria, but it also determines valid service placements by exploiting Monte Carlo simulations. However, these toolkits do not take into account fog node mobility, network uncertainty, as well as, energy consumption modeling; key pains for today's IoT services.

On the other hand, EdgeCloudSim and IoTSim-Edge [40] provide modeling abstractions for fog mobility. FDK [7] goes on step further and provides high-level interfaces for allocating compute and network resources, abstracting the complexities of fog computing from developers and enables the rapid development of fog systems. To enable more realistic testing of network configurations, a number of fog emulators have been proposed by academia. For example, FogBed [11], EmuFog [12], and EmuEdge [13], extend notable network emulators (e.g., MiniNet [42]) to also support fog resource and network heterogeneity. Nonetheless, since they are direct descendants from network emulators, they inherit their restrictions, such as strict modeling (e.g., configuration of routers, gateways, IP masks, etc). Most importantly, current emulators feature bounded scalability, restricting the testbed to be run on a single host (e.g., the developers' laptop) and preventing large-scale experimentation. Moreover, MockFog [43] is a fog emulator that enables users to inject network faults at runtime to evaluate the proper execution of an application under faulty configurations. However, MockFog is tightly bounded to OpenStack. Thus, *despite the plethora of toolkits, no current framework provides support for real-time monitoring and reproducable and dynamic "what-if" scenario assessment*. Most importantly, *all are limited to a single host testing environment and cannot scale to support large-scale experimentation.*

VIII. Conclusion

In this work, we introduced Fogify: an all-in-one Fog emulator that facilitates the rapid prototyping, deployment, and experimentation of IoT microservices over fog realms. Fogify features a powerful modeling framework for Fog topology definition that extends the docker-compose specification along with enablers for large-scale experimentation. Furthermore, a detailed description of the implementation aspects, such as resource management, network shaping and monitoring, is given. The evaluation of emulation accuracy displayed 3-8% deviation between emulated and real Infrastructures. Finally, we presented two real-world IoT workloads that demonstrate the usability, extensibility, and observability of Fogify.

Our future work includes an extensive scalability evaluation, which is currently only bounded by the Fogify Controller with performance impacted by the number of Fogify agents, emulated, and associated runtime actions undertaken.

Acknowledgement. This work is partially supported by the EU Commission through RAINBOW 871403 (ICT-15-2019-2020) project and by the Cyprus Research and Innovation Foundation through COMPLEMENTARY/0916/0916/0171 project.

REFERENCES

[1] D. Trihinas, G. Pallis, and M. Dikaiakos, "Low-Cost Adaptive Monitoring Techniques for the Internet of Things," *IEEE Transactions on Services Computing*, pp. 1–1, 2018.

[2] Z. Dong, Y. Lu, G. Tong, Y. Shu, S. Wang, and W. Shi, "Watchdog: Real-time vehicle tracking on geo-distributed edge nodes," 2020.

[3] Q. Qi and F. Tao, "A smart manufacturing service system based on edge computing, fog computing, and cloud computing," *IEEE Access*, vol. 7, pp. 86 769–86 777, 2019.

[4] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiatowicz, "The cloud is not enough: Saving iot from the cloud," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, Jul. 2015.

[5] M. Symeonides, D. Trihinas, Z. Georgiou, G. Pallis, and M. Dikaiakos, "Query-Driven Descriptive Analytics for IoT and Edge Computing," in *Proceedings of IEEE International Conference on Cloud Engineering (IC2E 2019)*, 2019.

[6] M. Villari, M. Fazio, S. Dustdar, O. Rana, D. N. Jha, and R. Ranjan, "Osmosis: The osmotic computing platform for microelements in the cloud, edge, and internet of things," *Computer*, vol. 52, no. 8, pp. 14–26, 2019.

[7] C. Powell, C. Desiniotis, and B. Dezfouli, "The fog development kit: A platform for the development and management of fog systems," *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 3198–3213, 2020.

[8] A. Brogi, S. Forti, and A. Ibrahim, "How to best deploy your fog applications, probably," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, May 2017, pp. 105–114.

[9] 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.

[10] G. D'Angelo, S. Ferretti, and V. Ghini, "Simulation of the internet of things," in *2016 International Conference on High Performance Computing Simulation (HPCS)*, July 2016, pp. 1–8.

[11] A. Coutinho, F. Greve, C. Prazeres, and J. Cardoso, "Fogbed: A rapid-prototyping emulation environment for fog computing," in *2018 IEEE International Conference on Communications (ICC)*, May 2018.

[12] R. Mayer, L. Graser, H. Gupta, E. Saurez, and U. Ramachandran, "Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures," *CoRR*, vol. abs/1709.07563, 2017.

[13] Y. Zeng, M. Chao, and R. Stoleru, "Emuedge: A hybrid emulator for reproducible and realistic edge computing experiments," in *2019 IEEE International Conference on Fog Computing (ICFC)*, 2019, pp. 153–164.

[14] C. Sonmez, A. Ozgovde, and C. Ersoy, "Edgecloudsim: An environment for performance evaluation of edge computing systems," *Transactions on Emerging Telecommunications Technologies*, vol. 29, no. 11, 2018.

[15] M. Ficco, C. Esposito, Y. Xiang, and F. Palmieri, "Pseudo-dynamic testing of realistic edge-fog cloud ecosystems," *IEEE Communications Magazine*, vol. 55, no. 11, pp. 98–104, Nov 2017.

[16] Y. Jiang, Z. Huang, and D. H. K. Tsang, "Challenges and solutions in fog computing orchestration," *IEEE Network*, vol. 32, no. 3, pp. 122–129, May 2018.

[17] "Fogify documentation," https://ucy-linc-lab.github.io/fogify/.

[18] R. Morabito, "Virtualization on internet of things edge devices with container technologies: A performance evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017.

[19] J. McChesney, N. Wang, A. Tanwer, E. de Lara, and B. Varghese, "Defog: Fog computing benchmarks," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, ser. SEC '19. New York, NY, USA: ACM, 2019, pp. 47–58.

[20] S. Bagchi, M.-B. Siddiqui, P. Wood, and H. Zhang, "Dependability in edge computing," *Commun. ACM*, vol. 63, no. 1, p. 58–66, Dec. 2019.

[21] C. Puliafito, E. Mingozzi, F. Longo, A. Puliafito, and O. Rana, "Fog computing for the internet of things: A survey," *ACM Trans. Internet Technol.*, vol. 19, no. 2, pp. 18:1–18:41, Apr. 2019.

[22] D. Trihinas, A. Tryfonos, M. D. Dikaiakos, and G. Pallis, "Devops as a service: Pushing the boundaries of microservice adoption," *IEEE Internet Computing*, vol. 22, no. 3, pp. 65–71, 2018.

[23] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate iot edge computing with lightweight virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102–111, Jan 2018.

[24] A. Brogi, S. Forti, and A. Ibrahim, *Predictive Analysis to Support Fog Application Deployment*. John Wiley & Sons, Ltd, 2019, ch. 9, pp. 191–221.

[25] M. Centenaro, L. Vangelista, A. Zanella, and M. Zorzi, "Long-Range Communications in Unlicensed Bands: the Rising Stars in the IoTand Smart City Scenarios," *IEEE Wireless Communications*, vol. 23, October 2016.

[26] W. Xia, P. Zhao, Y. Wen, and H. Xie, "A survey on data center networking (dcn): Infrastructure and operations," *IEEE Communications Surveys Tutorials*, vol. 19, no. 1, pp. 640–656, Firstquarter 2017.

[27] C. Nguyen, A. Mehta, C. Klein, and E. Elmroth, "Why cloud applications are not ready for the edge (yet)," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, ser. SEC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 250–263.

[28] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering*. O'Reilly, 2016.

[29] H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "CrystalNet: Faithfully Emulating Large Production Networks," in *SOSP '17 Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 599–613.

[30] Y. Barve, S. Shekhar, A. Chhokra, S. Khare, A. Bhattacharjee, and A. Gokhale, "Fecbench: An extensible framework for pinpointing sources of performance interference in the cloud-edge resource spectrum," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 331–333.

[31] D. Trihinas, G. Pallis, and M. Dikaiakos, "ADMin: adaptive monitoring dissemination for the internet of things," in *IEEE INFOCOM 2017*, Atlanta, USA, May 2017.

[32] "linux nsenter tool," https://www.man7.org/linux/man-pages/man1/nsenter.1.html.

[33] "token bucket filter," https://linux.die.net/man/8/tc-tbf.

[34] "linux tc-tool," https://linux.die.net/man/8/tc.

[35] B. Burns, *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*. " O'Reilly Media, Inc.", 2018.

[36] "linux perf3 tool," https://manpages.ubuntu.com/manpages /xenial/en/man1/iperf3.1.html.

[37] "linux ping tool," https://linux.die.net/man/8/ping.

[38] "Yolo: Real-time object detection," https://pjreddie.com/darknet/yolo/.

[39] "New york yellow cab dataset (jan-jun 2018)," https://on.nyc.gov/2OssELg.

[40] D. N. Jha, K. Alwasel, A. Alshoshan, X. Huang, R. K. Naha, S. K. Battula, S. Garg, D. Puthal, P. James, A. Y. Zomaya, S. Dustdar, and R. Ranjan, "IoTSim-Edge: A Simulation Framework for Modeling the Behaviour of IoT and Edge Computing Environments," pp. 1–19, 2019.

[41] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, 2011.

[42] B. Lantz, B. Heller, and N. Mckeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *In ACM SIGCOMM HotNets Workshop*, 2010.

[43] J. Hasenburg, M. Grambow, E. Grünewald, S. Huk, and D. Bermbach, "Mockfog: Emulating fog computing infrastructure in the cloud," in *2019 IEEE International Conference on Fog Computing (ICFC)*, June 2019, pp. 144–152.