Pre-print of article to appear in the ACM Transactions on Cyber Physical Systems, 2021, https://doi.org/10.1145/3446346

VIoLET: An Emulation Environment for Validating IoT Deployments at Large-Scales

SHREY BAHETI*, Cargill, India
SHREYAS BADIGER†, University of California, USA
YOGESH SIMMHAN‡, Indian Institute of Science, India

Internet of Things (IoT) deployments have been growing manifold, encompassing sensors, networks, edge, fog and cloud resources. Despite the intense interest from researchers and practitioners, most do not have access to large-scale IoT testbeds for validation. Simulation environments that allow analytical modeling are a poor substitute for evaluating software platforms or application workloads in realistic computing environments. Here, we propose VIoLET, an *emulator* for defining and launching large-scale IoT deployments within cloud VMs. It allows users to declaratively specify container-based compute resources that match the performance of native IoT compute devices using Docker. These can be inter-connected by complex topologies on which bandwidth and latency rules are enforced. Users can configure synthetic sensors for data generation as well. We also incorporate models for CPU resource dynamism, and for failure and recovery of the underlying devices. We offer a detailed comparison of VIoLET's compute and network performance between the virtual and physical deployments, evaluate its scaling with deployments with up to 1000 devices and 4000 device-cores, and validate its ability to model resource dynamism. Our extensive experiments show that the performance of the virtual IoT environment *accurately* matches the expected behavior, with deviations levels within what is seen in actual physical devices. It also *scales* to 1000s of devices, and at a modest cloud computing costs of under 0.15% of the actual hardware cost, per hour of use, with minimal management effort. This IoT emulation environment fills an essential gap between IoT simulators and real deployments.

CCS Concepts: • **Computing methodologies** → **Distributed computing methodologies**; • **Computer systems organization** → **Embedded and cyber-physical systems**; • **Networks** → **Network performance evaluation**; • **Hardware** → *Emerging simulation*; • **Software and its engineering** → *Simulator / interpreter*; *Software verification and validation*;

Additional Key Words and Phrases: Internet of Things, Edge computing, Fog computing, Cloud computing, Virtual Environment, Emulation, Distributed Systems, Scalability

ACM Reference Format:
Shrey Baheti, Shreyas Badiger, and Yogesh Simmhan. 2020. VIoLET: An Emulation Environment for Validating IoT Deployments at Large-Scales. *ACM Transactions on Cyber-Physical Systems* 1, 1, Article 1 (January 2020), 38 pages. https://doi.org/10.1145/3446346

*Based on work done as a graduate student at the Indian Institute of Science.
†Based on work done as a staff member at the Indian Institute of Science.
‡Corresponding Author

Authors' addresses: Shrey Baheti, Shrey_Baheti@cargill.com, Cargill, Bangalore, India; Shreyas Badiger, University of California, Irvine, USA, sbadiger@uci.edu; Yogesh Simmhan, simmhan@IISc.ac.in, Indian Institute of Science, Bangalore, India.

XXXX-XXXX/2020/1-ART1 $15.00
https://doi.org/10.1145/3446346

1 INTRODUCTION

The *Internet of Things (IoT)* is expanding rapidly as diverse domains deploy sensors, communication networks, and gateway infrastructure to support applications such as smart cities, personalized health and autonomous vehicles [1–3]. Such Cyber-Physical Systems (CPS) are also accelerating the need for, and the use of *edge, fog and cloud resources*, in a coordinated manner [4]. Edge gateway devices such as Raspberry Pis and smart phones have non-trivial resource capabilities, and can run a full Linux stack on 64-bit ARM processors. Fog devices such as NVidia's TX1 and Dell's Edge Gateways have power-efficient Atom processors or low-end GPUs to coordinate and complement the capabilities of edge devices [5]. The need for such resources comes from the availability of large volumes of IoT sensor streams that have to be analyzed closer to the edge to conserve bandwidth (e.g., video surveillance), or for processing data streams with low latency (e.g., smart grids) [6, 7]. At the same time, captive edge and fog devices do not have the seemingly infinite on-demand resource capacity of the cloud, which is necessary for scalable processing by some IoT applications.

Production deployments of IoT are seen as part of smart city utilities, intelligent transportation and personalized healthcare [8]. Here, edge and fog devices, along with sensors and actuators, are deployed as part of the physical infrastructure by the city, by service providers or by consumers, and this will accelerate as 5G communication gets rolled out [9]. At the same time, there is also active research at the intersection of IoT, CPS, and edge, fog and cloud computing that is investigating application scheduling, resiliency, big data platforms, and so on [10, 11]. However, a critical gap is the ability to *validate these deployments at-scale* before they go on the field, or to *verify the research outcomes* on real or realistic IoT environments. *Simulation environments* make many idealized assumptions and do not allow actual applications to be deployed [12–15]. Access to *large-scale IoT testbeds* with 100s of edge and fog resources (let alone 1000s present in a city) is limited, given the high cost and the overheads for maintenance and customization. Manually launching and configuring containers for *virtual IoT deployment* is time-consuming and error-prone.

At present, there does not exist any virtualized IoT environment that emulates the computing and network ecosystem of a real deployment without the need to purchase, configure and deploy the edge, fog and networking devices. Such a "cyber" environment is often common to many CPS domains. Here, we propose *VIoLET, a Virtual Environment for validating Internet of Things at Large-scales*. VIoLET is not a simulator but an *emulator* that allows real software to execute and real data to flow through the network links. The VIoLET Virtual emulation Environment (VE) offers several essential features that make it valuable for CPS researchers and planners.

(1) It emulates *realistic compute resources* comparable to edge, fog and cloud resources present in IoT deployments using virtualized containers, and includes models for *resource dynamism and device failures*.
(2) It allows the *easy definition* of diverse network topologies, and imposes *bandwidth and latency limits* between containers.
(3) VIoLET supports *virtual sensors* that generate data with various distributions within the containers, and can interface with external physical system simulators, mimicking local sensing devices.
(4) It can *actually run* real applications and platforms for software evaluation and validation.
(5) It runs on top of cloud VMs or commodity clusters, allowing it to *scale to 1000s of devices*, provided cumulative compute capacity and network bandwidth is available on the host machines.
(6) Lastly, the VE is *reproducible*, allowing accurate comparisons between systems and "what if" validation.

Besides its novel use of container-technology, VIoLET also proposes *placement strategies* for containers to VMs to efficiently pack devices. These help set up a validation environment that matches the behavior of city-scale IoT deployments in a fast, reproducible and cost-effective manner. *VIoLET is the first emulation environment to enable such validation of IoT, edge and fog computing.*

This article significantly extends our previous work [16] by including support for models of CPU dynamism, and of device failure and recovery (Section 3.5). It also includes substantially more detailed experiments that have been completely rerun on Microsoft Azure VM, and additionally reports micro-benchmarks on the physical devices and host VMs (Section 4.1), compares VIoLET against a real-world IoT deployment (Section 4.2), scales the VE to 1000 devices and $>$ 4000 device-cores (Section 4.3), and validates the dynamism and reliability features as well (Section 4.4), besides detailed Appendices with supplementary statistics. The updated *VIoLET v1.2* is available for download from https://github.com/dream-lab/VIoLET.

The rest of this article is organized as follows. We motivate various requirements for VIoLET in Section 2, describe its architecture design that meets these requirements, along with its implementation, in Section 3, present detailed results on deploying and scaling VIoLET for different IoT topologies in Section 4, compare it with related literature and tools in Section 5, and finally present our conclusions and future work in Section 6.

2 DESIGN REQUIREMENTS

Here, we present the high-level requirements for a *Virtual emulation Environment (VE)* like VIoLET, based on the needs of researchers and developers of applications, platforms and runtime environments, for IoT, CPS, edge, and fog resources.

Compute environment. The VE should provide the ability to configure computing resources that capture the performance behavior of *heterogeneous IoT resources*, such as edge devices, gateways, fog and even cloud resources. Key resource capabilities to be controlled include CPU rating, memory and storage capacity, and network. Further, the *environment should be able to host* software and applications within these resources [17, 18]. Virtual Machines (VM) have traditionally offered such capabilities, but are too heavy-weight for the resource-constrained and plentiful IoT devices. *Containers* are much more light-weight and provide similar capabilities as VMs. One downside is the inability to change the underlying Operating System (OS) as it is coupled with the Linux kernel of the host machine. However, we expect most IoT devices to run a flavor of Linux.

Networking. Communication is central to CPS, and the networking layer is sensitive to various deployment limitations on the field. Wired, wireless and cellular networks are common, each with different *bandwidth and latency characteristics* [18–20]. There is also a distinction between *local and wide area networks*, and *public and private networks* – the latter can limit the visibility of devices to each other due to firewalls. These affect the platforms and applications in the computing environment, and can decide who can connect to whom and if an indirection service is required. The VE needs to capture such diverse network topologies and performance behavior, to evaluate common application-layer IoT protocols such as *MQTT*, *Constrained Application Protocol (CoAP)*, etc.

Sensing and Data Streams. Sensors (and actuators) form the third vital component of IoT. These are often connected to the edge computing devices by physical links, *ad hoc* wireless networks, or may even be on-board the device. They form the source of the distributed, fast-data streams that are intrinsic to IoT deployments. The VE should provide the ability to simulate the generation of sensor event streams with various sampling rates and distributions at the compute devices for consumption by hosted or even remote applications [18]. These sensors should also be able to interface with external *physical system simulators [21, 22]* to complement the "cyber emulation" with "physical co-simulation" to support different CPS applications [18].

Application Environment. IoT, edge and fog computing runtime and applications should be able to execute on the virtual devices and network. Such IoT middleware, platforms and applications are often pre-loaded and pre-configured on the devices so that potentially 1000s of devices do not have to be reconfigured across the wide area network. The VE should allow such platforms and applications to be pre-deployed and ready-to-use to validate their performance and even security [17, 19, 23]. Users should not be forced to individually configure each compute resource, though they should have the ability to do so if required.

(Un)reliability and Dynamism. Cyber-resources in the real-world do not behave identically to their theoretical performance ratings and exhibit resource variability. Similarly, devices and networks are prone to failures due to the commodity nature of IoT resources [20]. Since service providers may manage these, failed resources may also be able to recover based on the Service Level Agreement (SLA) offered. These factors are important since we may need to validate the IoT middleware or applications upon such variations on the field. The VE should capture resource dynamism, failure and recovery, with diverse models that can be plugged-in based on the actual devices and the SLA.

Scalable. IoT deployments can be large, with 1000s of devices and sensors present in 100s of private and public networks with complex topologies [20, 23]. A VE should be able to scale to much large deployments with minimal resource and developer overheads. Simultaneously, these devices offer a real computing environment that requires underlying compute capacities to be available on the host machine(s). Hence, the VE should *weakly scale* as long as the underlying infrastructure provides adequate cumulative compute and network capacity for all the devices. The use of elastic cloud resources as the host can enable this.

Reproducible. Simulators limit resource realism and the ability to run real applications, but offer accurate reproducibility of runs. Physical deployments are hard to get access to and suffer from transient variability that affects reproducibility. A VE should offer a balance between running applications within a realistic deployment while being reproducible at a later point in time [23]. This also allows easy sharing of deployment recipes for accurate comparisons.

Cost-effective. Clouds offer a lower cost per compute unit due to economies of scale at data centers and can be leased. But IoT devices, while being commodity devices, are costlier to purchase, deploy and manage on the field or even in the lab, and are capital rather than an operational expense. VEs must match the IoT deployment's resource performance but at a cheaper compute cost [18, 19, 23]. They should offer a pay-as-you-go model that can be deployed on-demand and released after completing an experiment or validation.

Ease of Design and Deployment. Users should be able to configure large IoT deployments with ease, and have them deploy automatically and rapidly [23]. It should be possible to both compose realistic real-world topologies or automatically generate synthetic ones for testing purposes.

3 ARCHITECTURE

We first give an overview of VIoLET's architecture and our high-level approach, and subsequently discuss individual components and novel aspects of the design. Fig. 1 shows the top-level architecture of our framework. Users compose their IoT VE topology as a set of *JavaScript Object Notation (JSON) configuration documents* (Appendix A) that declaratively capture their requirements. A `device_types.json` document lists the available devices, their types (e.g., Raspberry Pi 3B, NVidia TX1) and their resource performance and reliability. A `sensor_types.json` document provides the virtual sensors and their configurations that are present. The network configuration document, `infra_gen.json`, defines the number of private and public networks and their topology, the numbers and types of devices and sensors present in a network, and the applications hosted on each. Lastly, the *deployment document*, `infra_config.json`, provides the actual runtime mapping from

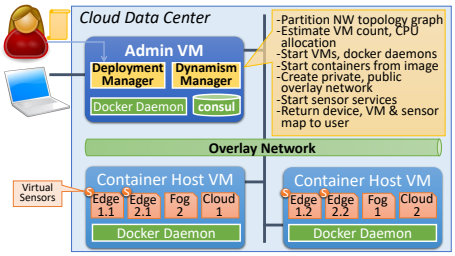
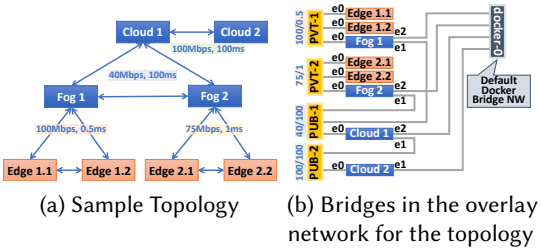


Fig. 1. High-level Architecture Design of VIoLET



(a) Sample Topology (b) Bridges in the overlay network for the topology

Fig. 2. Network Topology and Docker Overlay Network

devices to networks, binds them to IP addresses and subnets, and specifies their bandwidths and latencies. The deployment documents can be automatically generated by *visually composing* the IoT topology using a WebGML interface, *manually specified* by the user, or *synthetically generated*. The latter allows users to quickly launch sample synthetic deployments with minimal specification and reduce the time for validation.

The VIoLET *admin service* takes these documents and determines the number of cloud VMs of specified configurations necessary to host the device containers, with cumulative resources equivalent to the required number of devices. It also decides the mapping from devices to VMs while meeting the compute/memory/disk capacity of the devices, and the network bandwidth and latency needs of the topology, relative to what is made available by the host VMs. The user provides details of the available VM types for making these *resource provisioning* decisions in `vm_types.json`, and the specific VM instances for performing the deployment in `vm_instances.json` (Appendix A).

During the deployment, containers are configured and launched for each device on these VM instances using *Docker*, and inter-connected through an *overlay network*. This allows different private and public networks to be created in the VE. Further, Traffic Control (TC) and Network Address Translation (NAT) rules are set in each container to ensure that the requested network topology, bandwidth and latency limits are enforced. If specified, *virtual sensors* are then started on each device and their streams made available on a local network port in the container. Application environments or startup scripts are also configured or launched, if defined.

After this, VIoLET returns a `deployment_output.json document` to the user with details of the mapping from the logical device names in their deployment document to the matching container's physical device IPs, and the VMs on which the containers are placed on. Users can access these devices using the Docker `exec` command. Further, the port numbers at which various logical sensors streams are available on each device are also reported to the user in this JSON document. Together, these give full access to the deployed runtime environment to the user.

If the devices are configured with *dynamism and reliability models*, then the VIoLET admin service periodically samples from these distributions and modifies the resource behavior of the devices, terminates device containers to simulate faults, or restores terminated devices based on the recovery model.

Next, we examine these various activities and strategies in detail.

3.1 Compute Devices

Containers are emerging as a light-weight alternative to VMs for resource partitioning and multi-tenancy within a single host. They use Linux kernel's `cgroups` feature to offer the benefits of a custom software environment (except the OS kernel), and resource allocation and isolation, while having trivial overheads compared to VM hypervisors. They are well-suited for fine-grained resource allocation and software sand-boxing among trusted applications.

Computing devices in VIoLET are modeled as containers and managed using the *Docker* automation framework. There are four parts to this: *resource allocation and mapping*, *network configuration*, *software configuration*, and *management of dynamism*. The first three are done once at deployment time, and the last continues to occur while the deployment is active. Docker allows resource constraints to be specified on the containers [24]. We use this to limit a container's capacity to match the *CPU and memory* available on a native physical device. We perform CPU benchmarks on the native devices and the host (virtual) machines to decide the compute allocation, and use their memory and disk capacities as additional constraints. The commonly used *CoreMark®* [25] benchmark is currently supported for an Integer-heavy workload, while *Whetstone* [26] or other benchmarks like *LinPack* [27] are possible future alternatives for floating-point heavy applications. One subtlety is that while we use the multi-core benchmark rating of the device for the CPU scaling, this may map to fewer (faster) cores of the host machine. VIoLET pre-defines several common devices such as Raspberry Pi edge and NVidia Jetson fog devices, and others can be easily added.

Users can also specify the local *disk storage* capacity available to applications running in the container of a particular device type. We enforce this using the `pquota` module in Linux. However, the contents of the device's file system are only available when the container is running and not visible when the container is terminated. So, as a convenience, we allow users to specify a shared external directory on the VM that is mounted on the device for data persistence beyond the container's lifecycle, and for file-based access by external devices and applications. The configuration of network capabilities are discussed in Section 3.2.

Users also define the *application software environment* for their devices using a Docker image file (`Dockerfile`) that lists the software dependencies and other initialization parameters. This can be provided for each device type. Public Docker repositories have existing images for standard IoT middleware, platforms and applications (e.g., Eclipse Californium CoAP, Microsoft IoT Edge, RabbitMQ, EdgeXFoundry). VIoLET itself has minimal dependencies within a container for its framework configuration. This approach is similar to specifying a VM image, except that the users are limited to the host device's Linux kernel OS [1]. Hence, defining a compute device in VIoLET requires associating a device type for resources and a device image for the software environment.

3.2 Network Topology

Users describe the network topology for the devices based on three aspects: the *public or private* networks the device is part of; the *visibility* of devices to each other as enforced by firewalls; and the *bandwidth and latency* between pairs of devices. IoT networks are usually composed of numerous private networks that interface with each other and the public Internet through gateways. We allow users to define logical private networks and assign devices to them. For devices with multiple Network Interface Cards (NICs), each can be on a different network. Users can specify the IP ranges of these devices, and the subnet of the network. Each private network has a *gateway device* defined, and all traffic to/from devices in this subnet to external networks is routed through it. All gateway devices are part of one or more public networks, besides other devices that may be present on those public networks. E.g., a Raspberry Pi 3B+ has a Local Area Network (LAN) and a WiFi interface, and each can be part of its own network, say, one public and one private if it is a gateway.

By default, all devices in a private network are visible to each other, and have a common latency and bandwidth specified between pairs of devices; and similarly for all devices connected to a public network. Devices on different public networks can also reach each other and access the public Internet. However, users can override this visibility between any pair of devices or to the

[1] Docker recently introduced support for Windows and Linux containers hosted on Windows Server using the Hyper-V hypervisor. But this is more heavy-weight than Linux containers, and not used by us currently.

Internet and this is directional, i.e., $D1 \rightarrow D2$ need not imply $D1 \leftarrow D2$. These capabilities allow users to model the presence of firewalls and other network restrictions that are seen in real-world IoT deployments.

We implement the bandwidth and latency between devices using *Traffic Control (TC) rules* offered by Linux's `iproute2` utility, and the `network` service that we start on each container using `systemd` [28]. Here, every unique bandwidth and latency requirement gets mapped to a unique virtual Ethernet port (eth), and the rules are enforced on it. This Ethernet port is also connected to the bridge corresponding to the (private or public) network that the device belongs to. The bridges physically group devices that are on the same network, and also logically assign a shared bandwidth and latency to them. All devices on public networks are also connected to a common `docker-0` bridge for the VM they are present on, and which allows all-to-all communication between devices in the public networks, by default. Restricting the routing of traffic in a private network to/from the public network through its gateway device is enacted by `ip` commands and *NAT rules*. These rules redirect packets from the Ethernet port connected to the private network, to the Ethernet port connected to the public network. VIoLET fully automates these.

E.g., Fig. 2a shows a sample network topology specified by the user, and Fig. 2b the corresponding Ethernet ports and bridges configured by VIoLET to enact this. Here, the edge devices E1.1 and E1.2 are part of the private network PVT-1, with the fog device F1 as a gateway, and likewise E2.1, E2.2 and F2 form another private network, PVT-2. The bandwidth/latency within these private networks is uniform: 100Mbps/0.5ms for PVT-1, and 75Mbps/1ms for PVT-2. These are enforced on the Ethernet port (`e0, e1, e2` in Fig. 2b) connected to each bridge (yellow boxes). F1 and F2 fog devices further form a public network PUB-1 along with the cloud resource, C1, with a bandwidth/latency of 40Mbps/100ms. Similarly, the two cloud devices form another public network PUB-2, with 100Mbps/100ms. All these devices are on a single VM, and the public devices are also connected to the `docker-0` bridge for that VM. While the edge devices are connected to a single overlay network, the fog and cloud devices can be connected to multiple overlay networks, based on their bandwidth and latency requirements.

The Docker daemon running on a host allows us to define connectivity rules and *IP addressing* of containers present within that host machine using custom bridges. However, devices in VIoLET can be placed on disparate VMs and still be part of the same private network. Docker's existing *Swarm Mode* capability does not give us the flexibility to enforce fine-grained network and system parameters. Instead, we create a standalone basic container ("swarm classic") and define custom Docker *overlay networks* for communication between Docker daemons on different VMs [29]. For this, the host machines must be able to access a shared key-value store that maintains the overlay networking information and is queried by the Docker daemon on each VM for network routing. VIoLET uses the *Consul* discovery service as the key-value store, which is hosted in a separate container on an *admin VM*.

In summary, the Docker bridges and overlay network allow global routing across containers and VMs they are hosted on; the different Ethernet ports and NAT rules on them manage more fine-grained routing and network visibility between devices on different networks; and the TC rules on the ports enforce bandwidth and latency limits. As can be seen, configuring the required network topology is complex, time-consuming, and can be error-prone – if done manually for each IoT deployment. VIoLET helps translate the simple declarative document of the network topology provided by the user, and performs the heavy-lifting to define the overlay networks, eth interfaces, and the NAT and TC rules required to enact them. While some IoT deployments operate on *ad hoc* wireless networks where packet drops and signal attenuation need to be modeled, this is currently outside the scope of our work and may require us to interface with a network simulator such as *ns-3* [30] or an emulator like *mininet* [31].

3.3 Virtual Sensors

Edge devices are frequently used to acquire IoT sensor data over hardware interfaces like serial, UART or I2C, and make them available for applications and middleware to process locally and/or transfer to the cloud. Experiments and validation of IoT deployments require access to such large-scale sensor data. To enable this, we allow users to define virtual sensors that are collocated on the device containers. These virtual sensors simulate the generation of sensed events and stream them at a local network port of the device container, which acts as a proxy for a hardware interface to the sensor. Applications on that or a remote device can connect to this port, and read the observations for processing, as required.

A device can have multiple sensors, and we support various configuration parameters for these. The values for the sensor measurements themselves may be provided either as a text file with real data collected from the field, or defined as a statistical distribution, such as uniform random, Gaussian and Poisson, from which we sample and return synthetic values. In addition, the rate at which these values change or the events are generated is also specified by the user. Here too, we can set real relative timestamps or a distribution for the inter-event interval.

We implement the virtual sensors as a micro-service that is launched on each container as part of its startup. It is implemented in Python using the *Flask* micro-services framework and the *Gunicorn* Web Server Gateway Interface (WSGI) application server. The service pre-fetches, or generates and caches, the observations for the different virtual sensors for that device based on the specified values or distributions. When a client connects to this service and requests an observation for a sensor, the service returns the *current* reading for it based on the observed timestamp. For simplicity, this is reported as a Comma Separated Value (CSV) string consisting of a user-defined logical sensor ID, the observation timestamp and a sensed value, but can be easily modified. Access to the sensors by concurrent clients is also supported.

Notably, users can also provide their own implementation for the virtual sensor micro-service. The service can *push* events to an application instead of having it *pulled*. It can also interface with an external discrete-event physical systems simulator [21, 22] to provide events from the simulator to applications within the VIoLET containers accessing this virtual sensor. As future work, this can be extended to interface with actuation signals as well.

3.4 Resource Mapping and Deployment

A VIoLET service hosted on an *admin VM* receives the user's deployment documents as a REpresentational State Transfer (REST) request and enacts the deployment on other VMs in that data center (Fig. 1). Both Microsoft Azure and Amazon EC2 VMs are supported natively, and this can be trivially extended to other cloud providers or even a private cluster. To ensure the accurate performance behavior of the virtual emulation environment, we perform intelligent allocation of VM resources to a container and the mapping of containers to VMs to satisfy the deployment requirements.

Several factors affect these decisions. The containers represent compute devices, and the resource capacity of a container needs to be met by the underlying host VM or machine. At the same time, multiple containers can (and should) run on a single VM to allow it to be fully utilized and be cost-efficient, but without overwhelming the CPU, memory or disk capacity of the VM. Likewise, the bandwidth and latency between VMs are bounded, and we need to consider the network capacity required between containers on different VMs during their placement. Further, the bandwidth (or latency) required between containers can exceed (or be smaller than) that across VMs. In such cases, we need to consider placing these containers on the *same VMs* since the intra-VM bandwidth (latency) is much higher (lower) and may meet the requirement. Intuitively, this is a form of mapping

the *IoT deployment graph*, with devices forming vertices and edges the network connections, onto the *data-center graph* where VMs are vertices forming a clique, such that the compute and network needs of the former are met by the latter.

We address these constraints by using a *graph partitioning strategy*. Here, we label the vertices of the deployment graph with the device's CPU, memory and disk requirements, given as CPU benchmark metrics (e.g., iterations/sec for *CoreMark*), and GBs of RAM and disk capacity. These three form a vector of weights for the vertex. An edge exists between vertices if a source device can connect to a sink device, and this is weighted with the bandwidth required that network link. E.g., devices in a private network will form a clique, while there will exist edges between gateway devices from different private networks and with devices in the public network that are visible to each other.

We then make an *optimistic estimate* of the minimum number of VM instances we require of a given VM type. This is done by adding the vertex weights for CPU, memory and disk separately, and dividing each by the corresponding capacity of the VM type, and rounding up the largest of these values. This is the least theoretical number of VMs, say n, of a given type needed to meet the resource needs of all devices in the deployment.

Using this as the initial resource allocation, we *map devices to the VMs* by partitioning the device graph across these n VM hosts. We use `gpmetis` [32] for this, with the goal of balancing the three vertex weights (CPU, memory, disk usage) across the hosts and minimizing the sum of edge cuts (bandwidth usage) between hosts. This tries to collocate devices with high bandwidth inter-connects on the same host, while spreading the CPU, memory and disk requirements across VMs.

While CPU, memory, disk and network bandwidth are cumulative metrics (they increase as more containers are placed on/across VMs), latency is an independent metric (it can either be met or not). We perform the following checks to see if the partitioning gives a *valid solution*. We check if the sum of resource needs of all devices assigned to a VM is less than or equal to its respective resource capacities (CPU, memory, disk); if the sum of the bandwidths on the edge cuts between devices in each pair of VMs is within the available bandwidth capacity between the VMs; and if the latency required between any pair of connected devices is greater than the latency possible between the VMs that these devices are placed on. If these constraints are not satisfied, we increment n by 1 and repeat the partitioning using one more VM, and so on till we get a correct solution.

It may not be possible to meet the bandwidth or latency required between a pair of devices in a network in the deployment document by placing them across two VMs. E.g., the *inter-VM* bandwidth between two Azure D16 v3 VMs is seen at 7.2 Gbps and the latency at 0.54 ms, while for two containers on a single VM, the *intra-VM* bandwidth is 44.6 Gbps and the latency is 0.04 ms. So, as an additional optimization to achieve a feasible mapping, we "pin" all the devices of a private/public network, which have a greater bandwidth or smaller latency requirement than possible between two VMs, onto a single VM. This is done by replacing the vertices for those devices in the graph with a single *meta-vertex* that represents the entire network, and which has the sum of their resource requirements as its weights.

This greedy mapping tries to minimize the number of VM resources while ensuring that we meet the CPU, memory, disk, bandwidth and latency requirements of the deployment. A valid mapping may be infeasible if individual bandwidth, latency or resource requirements are more than what is supported on a single VM. If so, VIoLET will report this and a larger VM type has to be provided.

3.5 Resource Dynamism and Reliability

Real-world devices and networks suffer from variability in their performance relative to their theoretical rating. As we show later in the experiments for a 25-device physical IoT deployment (D25, Section 4.2), this dynamism is manifest in the performance of the CPU, network bandwidth

and latency [2] Such variability can cause the performance of the applications to be affected or cause the IoT and edge/fog middleware to respond, say, by rescheduling tasks to meet certain QoS guarantees [33]. Similarly, the devices and network links between them can fail due to the vagaries of field deployments and the equipment's commodity nature. Depending on the SLA offered by the IoT service provider, the devices may recover as well, say, using a standby device or manual intervention. VIoLET allows users to capture a subset of these dynamism and reliability scenarios.

3.5.1 CPU Dynamism. We offer users the ability to define a model for CPU dynamism. As part of the device type specification, they can indicate the *peak CPU variability*, δ, which is the fraction by which the CPU performance can drop below its theoretical rating ω, e.g., given by the peak CoreMark value for that device. While we can have different distributions to model the variability, for simplicity, we assume a uniform distribution of the CPU speed between the range $[(1-\delta)\times\omega,\ \omega]$. The user also specifies a *variability period* π, which is the time interval between which the CPU speed may change based on this distribution.

We coordinate the changes in CPU variability from the admin service. Importantly, this performance distribution is *independent* for each device, and the period is independent as well. So we must not change the performance for all the devices simultaneously. Hence, we introduce a *control interval* $\epsilon \ll \pi$ at which we probabilistically change the CPU speed for a subset of the devices. If n is the total number of devices of a particular type, at each time interval ϵ, we select $\frac{\epsilon}{\pi}\times n$ of the devices with equal probability, and for each of these, change their CPU speed to a value between $[(1-\delta)\times\omega,\ \omega]$, with uniform probability. We use Docker's ability to update the CPU rating for a container to enact this change, on the fly, without stopping the container.

While other forms of dynamism exist, such as network bandwidth and latency, these can leverage a similar decision-making model and be enacted using the respective capabilities. We leave these to future work.

3.5.2 Device Reliability. We allow users to define the *Mean Time to Failure (MTTF)* and *Mean Time to Recovery (MTTR)* for each device type. As is the usual definition, MTTF is the expected time taken by a device to fail since it was deployed/recovered. Similarly, MTTR is the expected time to fix the device and restore it, after it has failed. Setting MTTF to ∞ means that a device never fails, and setting MTTR to ∞ indicates that a device failure is permanent and it will not come back online. Formally, given the MTTF, μ, and MTTR, λ, for a device type, and a control interval $\epsilon \ll \mu$ and $\epsilon \ll \lambda$, as before, the VIoLET admin service performs independent and uniform sampling of a subset of running devices n of a given type to select $\frac{\epsilon}{\mu}\times n$ of them to terminate at each control interval. Similarly, it also selects a subset of all failed devices m of that type, $\frac{\epsilon}{\lambda}\times m$, with equal probability which will be restored. There are other forms of failures that are possible, such as network link failure or packet corruption, which we leave to future work.

3.6 Usability

Besides the command-line access to launching VIoLET deployments, we also define a Domain Specific Modeling Language (DSML) for defining the IoT devices and network topology of a deployment. This DSML uses the *Web-based Generic Modeling Environment (WebGME)* [34] that allows visual composition of the topology from the browser. Once composed, our VIoLET compiler generates the relevant JSON files from the DSML composition. This makes the deployment definition interactive and eases the definition of even complex IoT topologies. Fig. 12a in Appendix B shows a screenshot of configuring a private network with 20 devices using this interface.

[2]While these may also impact the performance of memory and disk I/O, we consider these resources more as capacity rather than performance limitations.

Table 1. Expected performance of Devices (top) and VM hosts (bottom) used in the deployments

Device	Cores⋆	Speed⋆	CoreMark†	Mem. (GB)⋆	Disk (GB)⋆	BW (Mbps)	Lat. (ms)†	US$
Pi 2B [3]	4	900 MHz	11,557	1	1	⋆LAN: 100	0.6	50
Pi 3B [4]	4	1.2 GHz	15,457	1	1	⋆LAN: 100	0.6	50
Pi 3B+ [5]	4	1.4 GHz	17,888	1	1	⋆LAN: 300 ⋆WAN: 300	0.5 1.3	50
NVidia Jetson TX1 [6]	4	1.9 GHz [7]	27,070	4	4	⋆LAN: 1,000 ⋆WAN: 867 [8]	0.7 8.7	499
Softiron 3000 [9]	8	2 GHz	77,739	16	16	⋆LAN: 10,000	0.2	2,400
Azure D16 v3 VM [10]	16	2.3 GHz	163,767	64	400	†Inter: 7,373 †Intra: 45,670	0.54 0.04	0.84/hr
Azure D32 v3 VM	32	2.3 GHz	301,269	128	800	†Inter: 15,930 †Intra: 47,897	0.58 0.04	1.68/hr
Azure D64 v3 VM	64	2.3 GHz	598,261	256	1,600	†Inter: 17,749 †Intra: 52,300	0.54 0.02	3.36/hr

⋆ Based on device specifications. † Based on 99^{th} percentile observed during device micro-benchmarks in Fig. 3a.

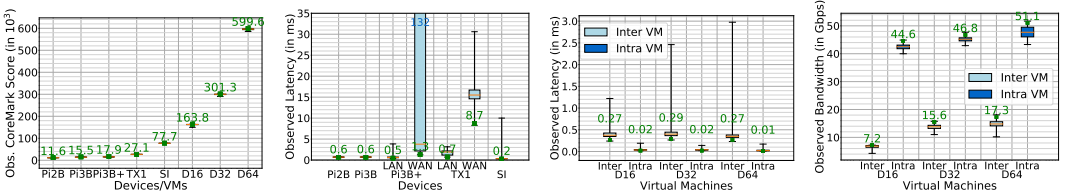
VIoLET also offers a *portal* to start and manage the deployment of a VE, designed using the Angular framework. It executes a series of Flask micro-services in the admin VM that enact the various stages of the deployment pipeline, such as (optionally) generating a synthetic deployment from high-level specifications, defining the mapping and placement, starting the Docker containers, instantiating the sensors, and (optionally) running benchmarks to validate the deployment. It also provides simple visualization capabilities about the container placement on VMs and resource usage. Fig. 12b in Appendix B shows a screenshot of using this UI to monitor the partitioning and mapping phase of the deployment for a 25-device setup.

4 EVALUATION

We offer detailed experiments to validate the efficacy of VIoLET, and specifically focus on its *accuracy* in reproducing the performance of the specified devices and network topology, and its *scalability*. Table 1(top) summarizes the different edge and fog devices used in our deployments.

We use three generations of Raspberry Pis as *edge devices* as they are one of the most popular gateways for IoT deployments. The *Pi 2B* (released in 2015) has 4×900 MHz ARM A7 32-bit CPU cores, while the *Pi 3B* (2016) and *Pi 3B+* (2018) have 4 ARM A53 64-bit cores rated at 1.2 GHz and 1.4 GHz, respectively. All three have 1 GB RAM, and 1 GB of nominal disk space for user applications. In addition, we have two *fog-class devices* – an *NVidia Jetson TX1* with 4×1.9 GHz ARM A57 64-bit cores [11] and 4 GB RAM, and a *Softiron Overdrive 3000 server (SI)* with an AMD A1100 CPU having 8×2 GHz ARM A57 64-bit cores and 16 GB RAM. All these devices have on-board LAN Ethernet ports rated at between 100 Mbps – 10 Gbps. The Pi 3B+ and TX1 have an additional WiFi interface on-board that allows them to be part of two networks, with one interface connected to the LAN and the other to the Wide Area Network (WAN). While these are

[3] https://www.raspberrypi.org/products/raspberry-pi-2-model-b/
[4] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[5] https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/
[6] https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/
[7] https://devblogs.nvidia.com/nvidia-jetson-tx1-supercomputer-on-module-drives-next-wave-of-autonomous-machines/
[8] http://images.nvidia.com/content/tegra/embedded-systems/pdf/JTX1-DevKit-Product-sheet.pdf
[9] https://softiron.com/wp-content/uploads/2015/11/SoftIron-Overdrive-3000-Data-Sheet-Final.pdf
[10] https://azure.microsoft.com/en-in/pricing/details/virtual-machines/linux/
[11] While the TX1 has an on-board GPU, we currently do not expose it as part of the container due to technology limitations, and it is planned for future work.



(a) Baseline *CPU CoreMark* (b) Baseline Device *Latency* (c) Baseline VM *Latency* (d) Baseline VM *Bandwidth*

Fig. 3. Micro-benchmarks measuring peak performance on physical devices and VMs. Green labels indicate the 99^{th} *percentile peak performance* – highest value for CPU and bandwidth, and lowest for network latency. A blue label at the top indicates the maximum value of the whisker, if truncated.

the device ratings, the network behavior also depends on the network switch they are connected to, as discussed for relevant experiments. E.g., when using 1 Gbps switches for the LAN, the SI only offers a bandwidth of 1 Gbps and not the 10 Gbps of its NIC. These devices are also configured with a nominal available user disk space in our experiments. The Pis cost $\approx US\$50$ for a working device, the TX1 development kit costs $US\$499$ and the SI server costs $US\$2,400$, at the time of purchase.

We use Microsoft Azure's VM instances as our host machines for deploying VIoLET. We use three different VM types to offer the best price to performance ratio for the different deployment capacities we attempt. All are part of the Standard D v3 series deployed in the Central India data center. The *D16, D32 and D64* VMs have 16, 32 and 64 CPU cores each, running Intel Xeon E5-2673 v4 at 2.3 GHz, with 64 GB, 128 GB and 256 GB of RAM, respectively. While Microsoft provides an expected network speed for these VMs of 8, 16 and 30 *Gbps* [12], we use micro-benchmarks to empirically estimate their observed performance. The bandwidth and latency, both *intra-VM* between containers on the same VM, and *inter-VM* between multiple VMs, are measured and reported. This gives a more accurate estimate of the available network capacity that is used during resource mapping. In our earlier study [16], we have use similarly-sized Amazon EC2 VM instances, and the results are comparable.

4.1 Micro-benchmarks

While some of the device and VM performance metrics are part of their data-sheets, others need to be micro-benchmarked to find their baseline performance. Also, specifications like the CPU clockspeed and cores have to be translated to their practical application performance for comparison. Lastly, there are some limitations on the possible combinations of bandwidth and latency that are valid for a given network link. Here, we offer detailed micro-benchmarks to evaluate these baseline parameters missing from data-sheets, before discussing different VIoLET deployments. Such micro-benchmarks take a few hours to run per device, and are also performed on each new IoT device or new VM type that VIoLET needs to support in its deployment configuration.

CPU Benchmarks. CPU clockspeeds cannot be used to uniformly compare application performance across processor types or architectures. Instead, we express the CPU performance of the devices and the VMs using the CoreMark benchmark [25]. CoreMark internally implements standard algorithms such as list and matrix operations, stream processing and CRC using ANSI C, in an architecture and library independent manner that is reproducible across embedded and server-class CPUs. Specifically, we use it to understand the *peak performance* for each of these

[12] Microsoft Azure General purpose virtual machine sizes, https://docs.microsoft.com/en-us/azure/virtual-machines/linux/sizes-general

compute resources in order to provision adequate capacity, and also examine their *deviation* from the expected CPU performance later on.

The CoreMark benchmark is compiled on each of the devices and VMs to run in a multi-threaded mode, with as many threads as the number of CPU cores. We run the benchmark in a continuous loop for 1 *hour* on multiple instances of each device type and an instance of each Azure VM type. We perform about 100–300 CoreMark runs per device and VM type, and report their scores as a Box and Whiskers plot in Fig. 3a. We also label the top 99^{th} percentile score for each device/VM type, and use this value as the "ideal" or expected *peak CPU performance* for that resource (see *CoreMark* column in Table 1).

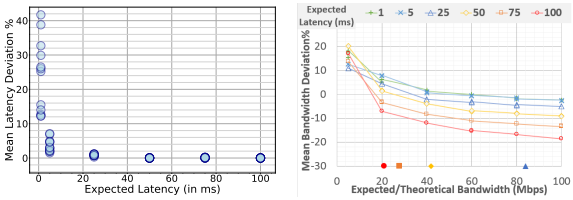
The CoreMark trends are similar to what is expected from the respective CPUs and the number of cores. The three types of Pis have a 99^{th} percentile CoreMark score between $11.56k$ to $17.89k$. The Pi 2B has an older ARM architecture, while the 3B and 3B+ vary only in their clock speeds. In fact, the ratio of the clockspeeds of 3B and 3B+ match the ratio of their CoreMarks. The TX1 has a newer and faster ARM processor than the 3B, and has a score of $27.07k$. All these devices have 4 cores. The Softiron is a low-end server, and its AMD CPU based on the ARM architecture is much faster and has 8 cores, resulting in a score of $77.74k$. The three VM types we use as the host machines run server-grade Intel CPUs and have a higher per-core performance. They also exhibit fairly linear scaling with the number of cores. The D16, D32 and D64 VMs have 99^{th} percentile scores of $163k$, $301k$ and $599k$. The scores from all runs on a device type fall within a narrow band, indicating that these are predictable and reproducible. So we expect low variability and deterministic compute performance from these devices and VMs.

Latency Benchmarks. Latency values are not typically given as part of device or VM specifications. Hence, we need to rely on empirical measurements. Latency is also affected by the two devices that communicate with each other and the network switch they pass through. We measure the latency distribution for the LAN and Wireless LAN (WLAN) network interface for each device and VM, as applicable, and report the 99^{th} percentile smallest value as the lower bound of the expected latency for the resource.

For the *physical devices*, we use the network topology described later for the *D25 setup* in Section 4.2 and Fig. 5. Here, the Pi 2B and 3B use their LAN ports, the Pi 3B+ is connected to two separate networks using its LAN and WLAN ports, respectively, and the TX1 is connected using its WLAN port. Separately, we also run experiments using the LAN ports of the TX1 and SI with a Pi 3B+ device to ensure that all devices and their available network ports are evaluated. These devices are connected through WiFi routers that support the best protocol supported by the devices' WLAN ports, or through Gigabit LAN switches. We perform all-to-all `fping` between devices in the same public or private network which are visible to each other, with devices in each network connected to a common switch. We measure $\approx$ 100–2000 pings for each device type and port type.

Fig. 3b shows a Box and Whiskers plot of the latency value distributions (in ms), with the 99^{th} percentile *smallest latency* labeled in green – this is an indicator of the shortest latency that is achievable. We see that the LAN interfaces of all devices offer a tighter latency distribution than the WLAN interfaces, while also being lower with a *sub-millisecond* 99^{th} lowest percentile latency for all devices. The WLAN's latency values are both higher and wider. The Pi 3B+ has a 99^{th} least percentile latency of 1.3 *ms* but with $Q1$–$Q3$ quartile values that span a 52 *ms* range, while the TX1's WLAN has a higher 99^{th} percentile of 8.7 *ms* but a relatively narrower distribution that is within 31 *ms*.

The *Azure VMs* do not report details on their network port or switch configuration within the data center. We use two VMs of the same type to perform `fping` 1000 times for the inter-VM network latency. For the intra-VM latencies, we create two containers on a single device equivalent



(a) *Mean latency deviation%* for different expected latencies (b) *Mean bandwidth deviation%* for various expected bandwidth–latency pairs.

Fig. 4. Mean deviation% of bandwidth and latency between containers, for different bandwidth–latency configurations

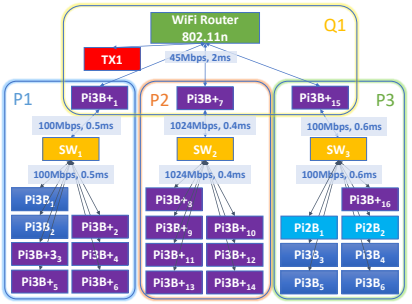


Fig. 5. D25 deployment setup and network configuration

to a Pi 3B+ device and do 1000 `fpings` between them. Fig. 3c shows the box plot of observed intra and inter VM latencies (in ms) for each VM type, with the 99^{th} percentile lowest latency labeled. The latencies do not vary much as the VM type changes, with inter-VM at 0.5 *ms* and intra-VM at ≈ 0.03 *ms*. The latency distributions are also tight. The intra-VM latency is useful for VIoLET to co-locate containers on a VM to mimic very low latency IoT networks.

Bandwidth Benchmarks. We do not perform bandwidth micro-benchmarks for the physical devices since these are specified as part of the network interfaces, switches and routers. Later, we evaluate their real-world accuracy in Section 4.2. The Azure VMs have an expected inter-VM bandwidth of 8, 16 and 30 *Gbps* specified for the three VM types. We empirically verify these inter-VM bandwidths, and further measure the intra-VM bandwidths. As before, we use two VMs of the same type for the inter-VM bandwidth experiments, and run the multi-threaded `iperf3` TCP benchmark between them for over 300 runs ($\approx$ 1 *hour*), with file sizes of $\approx$ 5 *MB*. The results using `iperf3` are comparable to Microsoft's `ntttcp` bandwidth benchmark, which they use within Azure [35]. For the intra-VM experiments, we create two Pi3B+ class containers on the same VM, and as before run the `iperf3` benchmark for 1 *hour*. We use 2 threads, which gives the best bandwidth.

The inter and intra VM bandwidth distributions are shown as box plots in Fig. 3d, with the 99^{th} percentile *highest bandwidth* indicated, which we use as a measure of their peak expected bandwidth used during container mapping. All bandwidth values have a tight distribution. We see that the inter-VM bandwidth for D16 and D32 are close to their rated performance, at 7.2 *Gbps* and 15.6 *Gbps* at their 99^{th} percentile. However, the D64 VM achieves only 56% of its rated performance at 17.1 *Gbps*, only modestly above D32 rather than double its value. This may be due to multi-tenancy or hops across under-provisioned network switches. This is unlike the CPU performance that increased linearly with the VM size. The intra-VM bandwidths are much higher but do not vary dramatically across the VM types, with the 99^{th} percentile values ranging from 44–51 *Gbps*. These appear to be memory bound since such bandwidths are comparable to the peak performance of DDR3 memory modules that are expected on these machines. This indicates that IoT networks with high network bandwidths of up to 50 *Gbps* can be virtually deployed on a single VM, and those with a total cross-section bandwidth of up to 17 *Gbps* can be formed across VMs.

Network Configuration Micro-benchmarks. Being able to model network behavior accurately is essential for IoT VEs. Here, we perform more detailed experiments that evaluate the behavior of specific bandwidth and latency values between different containers that are enforced by the TC rules. These micro-benchmarks help specify viable network configuration options for larger VEs. Specifically, we try out 36 different combinations of latency and bandwidth pairs, formed from

$\{1, 5, 25, 50, 75, 100\}$ *ms* $\times$ $\{5, 20, 40, 60, 80, 100\}$ *Mbps*. Each *latency–bandwidth* pair is configured for all devices in a private network in a VIoLET deployment, with each network having 12 devices selected from among those in Table 1. Each deployment has eight private networks placed on five D32 VMs, and nine such deployments are conducted. For each private network, we execute multiple `fping` and `iperf` runs between the devices in the network. We calculate the difference between the observed latency or bandwidth and the expected (i.e., configured) performance as the *deviation%*, given as: $deviation\% = \frac{(Observed - Expected)}{Expected}\%$.

Fig. 4a shows the mean of the deviation% of the observed latency on the Y axis as the expected latency increases on the X Axis, for each combination of latency–bandwidth pairs. This shows that the latency deviation is sensitive to the expected latency value. For small latency values of 1 *ms*, it ranges between 12–42%, and when the expected latency increases to 5 *ms*, this variation range drops to 1.5–7.1%. The deviation asymptotically reduces for higher latencies, with those over 50 *ms* having just a 0.1% skew. Since we can achieve a sub-millisecond latency across Azure VMs and even better within a VM, the higher deviation% for low latencies are attributed to the challenges of enforcing such limits using the software TC rules, where even a small variation of 100 μs in absolute latency can cause a larger effect on the deviation%. For higher latency values, the deviation% remains small, and independent of the bandwidth of the network that are tested.

Next, we examine the effect of the bandwidth–latency pairs on their bandwidth deviation%. We report that the deviation in bandwidth is a function of both the expected latency and the bandwidth. In fact, it is also a function of the *TCP window size*, which by default is set to 262, 144 *bytes* in the containers. The *Bandwidth–Delay Product (BDP) [36]* is defined as the product of the bandwidth and latency. For efficient use of the network link, the TCP window size should be greater than this BDP, i.e., $Window \geq Bandwidth \times Latency$. In other words, given a fixed latency and TCP window size, the $Peak\ Bandwidth = \frac{Window}{Latency}$. Fig. 4b plots the mean deviation% of the observed bandwidth on the Y axis, for different expected latencies, as the expected bandwidth increases on the X axis. A marker on the bottom of the X axis shows the *peak theoretical bandwidth*, for that latency using the default TCP window size. We observe that for low latencies of 1–25 *ms* (green plus, light blue cross, dark blue triangle), the bandwidth deviation is limited and falls between -5.1 and 19.4% for all bandwidths evaluated, from 5–100 *Mbps*; a positive deviation indicates that a higher bandwidth was observed than was configured. With the default window size, even a latency of 25 *ms* supports a peak bandwidth of 83 *Mbps*, and lower latencies support an even higher peak bandwidth. The positive deviation is also high for low bandwidth values and low for high bandwidth values. This is because for low values of expected bandwidth, even small changes in the absolute bandwidth causes a larger change in the relative deviation%.

As the latency increases, the negative deviation% increases as the bandwidth increases. In particular, as we cross the peak bandwidth value on the X axis, the deviation% becomes more negative, indicating an inability to meet the expected bandwidth given the window size. E.g., at 75 *ms*, the peak bandwidth supported is only 28 *Mbps*, and we see the bandwidth deviation% for this latency worsens from -3.2% to 12.2 when the bandwidth configuration increases from 20 *Mbps* to 80 *Mbps*. This behavior is as expected from the BDP equation, and indicates that the users of the container need to tune the TCP window size in the container to enforce bandwidths more accurately, or select a bandwidth–latency pair that conforms to the default window size. In all the experiments we subsequently report, we ensure that the latter holds.

4.2 Comparison with Physical Setup (D25)

A key goal for VIoLET is for its *virtual environments* to have a resource performance that closely matches the *devices and networks in the real-world* that they model. However, even for physical

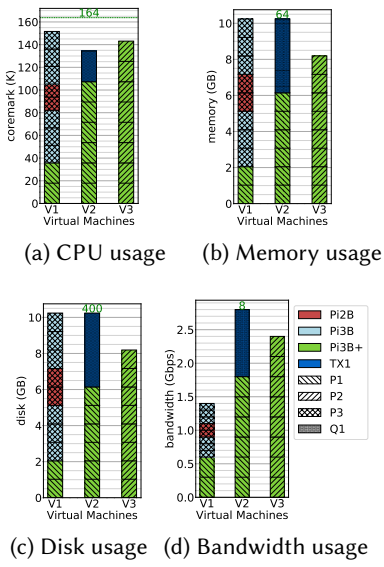
Table 2. Devices, their mapped VMs and the rated performance, for four different deployment setups

Deployment Setup→	**D25**			**D100**			**D400**			**D1000**		
Device/ Host	Count	∑CM (k)	∑Mem (GB)	Count	∑CM (k)	∑Mem (GB)	Count	∑CM (k)	∑Mem (GB)	Count	∑CM (k)	∑Mem (GB)
Pi 2B	2	23	2	51	589	51	255	2,947	255	542	6,263	542
Pi 3B	6	93	6	36	556	36	96	1,484	96	292	4,513	292
Pi 3B+	16	286	16	11	197	11	47	841	47	151	270,1	270
TX1	1	27	4	1	27	4	1	27	4	10	270	40
Softiron	-	-	-	1	78	16	1	78	16	5	389	80
Total devices	**25**	**429**	**28**	**100**	**1,447**	**118**	**400**	**5,377**	**418**	**1,000**	**14,136**	**1,226**
D16 VM	3	491	1,200	-	-	-	-	-	-	-	-	-
D32 VM	-	-	-	5	1,506	4,000	20	6,025	16,000	-	-	-
D64 VM	-	-	-	-	-	-	-	-	-	25	14,956	40,000

resources, their *rated performance* may not match their *observed performance*. Here, we compare and contrast the rated performance for the physical devices and the network, against the observed performance of the physical devices and also the observed performance of the VIoLET deployment.

4.2.1 Setup. We use a real IoT deployment, which we term **D25**, consisting of 25 physical devices connected using four networks, as illustrated in Fig. 5 and described in Table 2. It reflects the heterogeneity typical of a real-world IoT setup, and comprises of 2 Pi 2Bs, 6 Pi 3Bs, 16 Pi 3B+s, and 1 NVidia TX1. These are grouped into three private networks (*P1–P3*) and one public network (*Q1*). The private networks have 8 devices each connected using their LAN ports, with a Pi 3B+ serving as a *gateway* to the public network on its additional WLAN port. Devices in P1 and P3 are connected using the LAN ports of two *TP-Link TL SF 1008D* routers which support 100 *Mbps* network bandwidth (SW_1 and SW_3 in Fig. 5) while devices in P2 are connected using the LAN ports of a *TP-Link TL SG 1008D* router which supports 1 *Gbps* (SW_2). *The effective bandwidth supported is the lower of the rated bandwidths of the device NIC and the router.* While P2 has 8 homogeneous Pi 3B+ devices, P1 and P3 have multiple Pi types present. The 3 Pi 3B+ gateway devices and the standalone TX1 on the public network Q1 are connected using their WLAN ports to a *TP-Link TL WR 940N* WiFi router which supports the 802.11n protocol at 450 *Mbps*. We place the devices on WiFi proximate to the router to reduce interference and maximize the network performance achieved. But the median bandwidth of the WiFi links established by devices in the public network is only 135 *Mbps*, which is set as the expected bandwidth of the network. The expected latency values for these networks are based on the 99^{th} lowest percentile supported by the devices on the network, given by the earlier micro-benchmarks. These expected bandwidth and latency values for the four networks are labeled in Fig. 5.

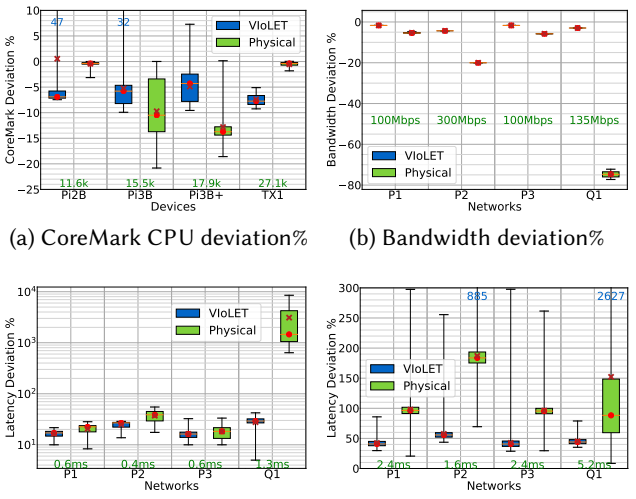
We replicate this deployment as a VE on VIoLET, with matching CPU CoreMark rating, memory and disk capacity (as per Table 1), and network connectivity and performance (as shown in Fig. 5). We map the containers for these 25 devices onto 3 D16 VMs, and Fig. 6 shows the corresponding CPU, memory, disk and bandwidth usage on these three VMs. The device types are denoted by their color, their heights indicate the relevant metric, and their shading pattern represents the network(s) they belong to. The gateway devices that belong to both the private and public networks are shown with the shading of the private network. The green label at the top gives the capacity of the VMs for that resource. As we see, the container placement is constrained by the CPU capacity of each VM (CoreMark of $\approx 164k$), while there is excess memory (64 *GB*), disk (400 *GB*) and bandwidth (8 *Gbps*) available on them. Fig. 6a indicates that devices of the private network P2, which hosts only the faster Pi 3B+s, are mapped to VM V3 exclusively, while devices of P3, having a mix of fast and slow Pis, are placed on VM V1 along with two devices from P1. The rest of the devices from P1



(a) CPU usage (b) Memory usage

(c) Disk usage (d) Bandwidth usage

Fig. 6. *Resource usage* on each of the three VMs by the 25 VIoLET containers after mapping, for the D25 deployment. The VM's capacity for each metric is shown in green text on top. The legend is common to all plots.



(a) CoreMark CPU deviation% (b) Bandwidth deviation%

(c) Latency deviation% (Log) (d) Ping-pong application latency deviation%

Fig. 7. Box plot of *deviation%* for the VIoLET containers and the physical devices from their expected performance, in the D25 setup. The expected performance is labeled in green at the bottom. The median is a red circle and the mean a red cross. A blue label at the top indicates the maximum value of the whisker, if truncated.

and TX1 on the public network Q1 are on VM V2. This reflects the graph partitioning algorithm's preference for co-locating devices from the same network on the same VM to reduce edge-cuts. Each virtual device also has three on-board virtual sensors configured.

The cost of the physical compute devices, excluding network routers, is $\approx US\$1700$. In contrast, the VMs for the D25 VIoLET setup cost $\approx US\$2.52/hour$ to rent. More details are in Appendix E.

4.2.2 Analysis of CPU Performance. We run three baseline CPU and network **validation benchmarks** on the performance of the *physical and virtual deployments*. These benchmarks are also used in the evaluation of other deployments discussed in later sections. For the *observed CPU performance*, we run *CoreMark* on all the devices concurrently 15 times, which takes ≈ 7 *mins* to complete for the D25 setup. Next, we uniformly sample a subset of $\frac{n}{2}$ links from $n(n-1)$ possible ones in each private and public network, and run `iperf3` between them to measure the *observed bandwidth*, with each concurrent run taking ≈ 10 *secs*. This benchmark is limited to fewer links since the overhead for an all-to-all run is high for larger networks. Lastly, we run `fping` between a random $2n$ of the $n(n-1)$ pairs of devices within each private and public network, and measure the *observed latency*. Each run takes < 1 *secs*. Figs. 7a, 7b and 7c show box plots of the deviation% between the expected CoreMark, bandwidth and latency, respectively, and the physical and the VIoLET deployments, grouped by the device types (Table 1) taken cumulatively across networks. A distribution of their absolute measurements and absolute deviations are given in Appendix D.

In Fig. 7a, the box for the VIoLET containers closely matches the expected 99^{th} percentile *CoreMark* (in *thousands*) of the physical device micro-benchmarks. All values in the $Q1$–$Q3$ quartiles are within a -2% to -9% deviation range for all device types. Among the three Pi devices, we see that the box is slightly wider for Pi 3B+, partly because it has more number of devices spread across

all three VMs, compared to the other two device types. We also see that the CoreMark deviation% of the VIoLET containers is comparable to the physical devices themselves. The Pi 2B and TX1 devices have a low deviation of close to 0%. However, this is wider for Pi 3B and 3B+. We observe that individual Pi devices have a narrow distribution of their CoreMark values, within a 5% range, but different Pi devices of the same type vary in their CoreMark values by up to 20%. Since we have many Pi 3B and 3B+ devices in the D25 physical deployment, their deviation% are worse than the 1 TX1 device and the 2 Pi 2B devices. A statistical *sign test* of the CoreMarks of the physical and emulated devices shows that the Pi3B and Pi3B+ virtual devices statistically out-perform the physical devices, while Pi2 and TX1 under-perform (more details in Appendix D.3). Figure 13a of Appendix D also indicates that the absolute CoreMark of VIoLET's device containers are close to or better than the physical devices. *So the compute capability of the VIoLET container devices is comparable to a physical device, and our IoT VE can serve as a reasonable replacement for a physical deployment.*

4.2.3 Analysis of Network Performance. Fig. 7b shows that in all cases, except for the physical deployment of the public network Q1, the *bandwidth deviation%* is tight, indicating deterministic performance. The two exceptions are both for the physical networks. For the private physical network P2, which has a 300 *Mbps* effective connectivity between the Pi3B+ device and router, they are unable to meet this theoretical bandwidth and show a 20% (60 *Mbps*, Fig. 14b in Appendix D) lower performance than expected. The P2 network for the VIoLET containers is only marginally affected. The public physical network Q1 shows a much higher deviation, with a mean reduction of 74.7% (100.9 *Mbps*). This is due to the wireless network whose link connects at $\approx$ 135 *Mbps* but the achieved bandwidth is much lower at a median of $\approx$ 45 *Mbps*. The WLAN also shows a marginally higher variability in network performance compared to the LAN. One of the future works would be for VIoLET to be (counter-intuitively) *less deterministic* to capture the variability of the real-world wireless networks more accurately.

The *network latency deviation%* in Fig. 7c are greater than for CPU and bandwidth, for both the VIoLET and the physical networks, though small in absolute value (Figs. 13c and 14c in Appendix D). For VIoLET, the median deviation% across the four networks ranges from 16.2% to 29.3% (a mean reduction of -0.1 to -0.36 *ms*), with a $Q1$–$Q3$ quartile variation of no more than 6%. The deviation inversely correlates with the absolute expected latency values, with lower latency networks having a higher deviation, as we saw in Fig. 4a. But the Q1 public network has a slightly higher median deviation% than P1 and P3, despite having a higher absolute latency. This is because the containers of Q1 are spread across all three VMs while those for each of the three private networks are placed within a VM. So there is a marginally higher latency variability introduced across VMs for Q1. For the three private physical networks, the median deviation% are comparable to VIoLET at between 16.3% to 38.8% (a mean drop of -0.12 to -0.31 *ms*). However, the latency for the Q1 public WLAN network is very high at a median of 1457%, with high $Q1$–$Q3$ quartile variability that spans 3193%, i.e., while the expected latency based on the device rating was 1.3 *ms*, the observed median latency was 20.25 *ms*. This too is consistent with the earlier micro-benchmarks in Fig. 3b, where the WLAN latencies were much more variable. We provide additional statistical validation in Appendix D.3.

4.2.4 Analysis of Pub-Sub Application and Virtual Sensors. CPS systems often use a publish-subscribe model (pub-sub) for sensing and communication with the analytics layer [20]. We run an application layer *ping–pong* benchmark to measure the effect of the network performance on such pub-sub user services deployed on the VE and the physical setup. Here, we deploy the *Eclipse Mosquitto MQTT broker*, which uses the MQTT messaging protocol popular for IoT applications, on each of the four networks' gateway device. We then randomly choose device pairs in each network such that all devices are either a publisher or a subscriber. The publisher reads an observation

from its local virtual sensor (VIoLET) or file (physical) and initiates sending a message of length 68 *bytes* to a unique topic at the broker, and the subscriber receives this. On receipt of a message, the subscriber publishes it back to a matching topic in the broker, which in turn is subscribed to by the first publisher. The messages are read and sent every 1 *sec*, and repeated for 3 *mins* on the VIoLET and the physical deployments. Each message effectively passes through four network hops end-to-end: *publisher—broker—subscriber—broker—publisher*. Its observed end-to-end latency is compared against the expected minimum latency to report the deviation% shown in Fig. 7d, and the absolute values given in Figs. 13d and 14d of Appendix D.

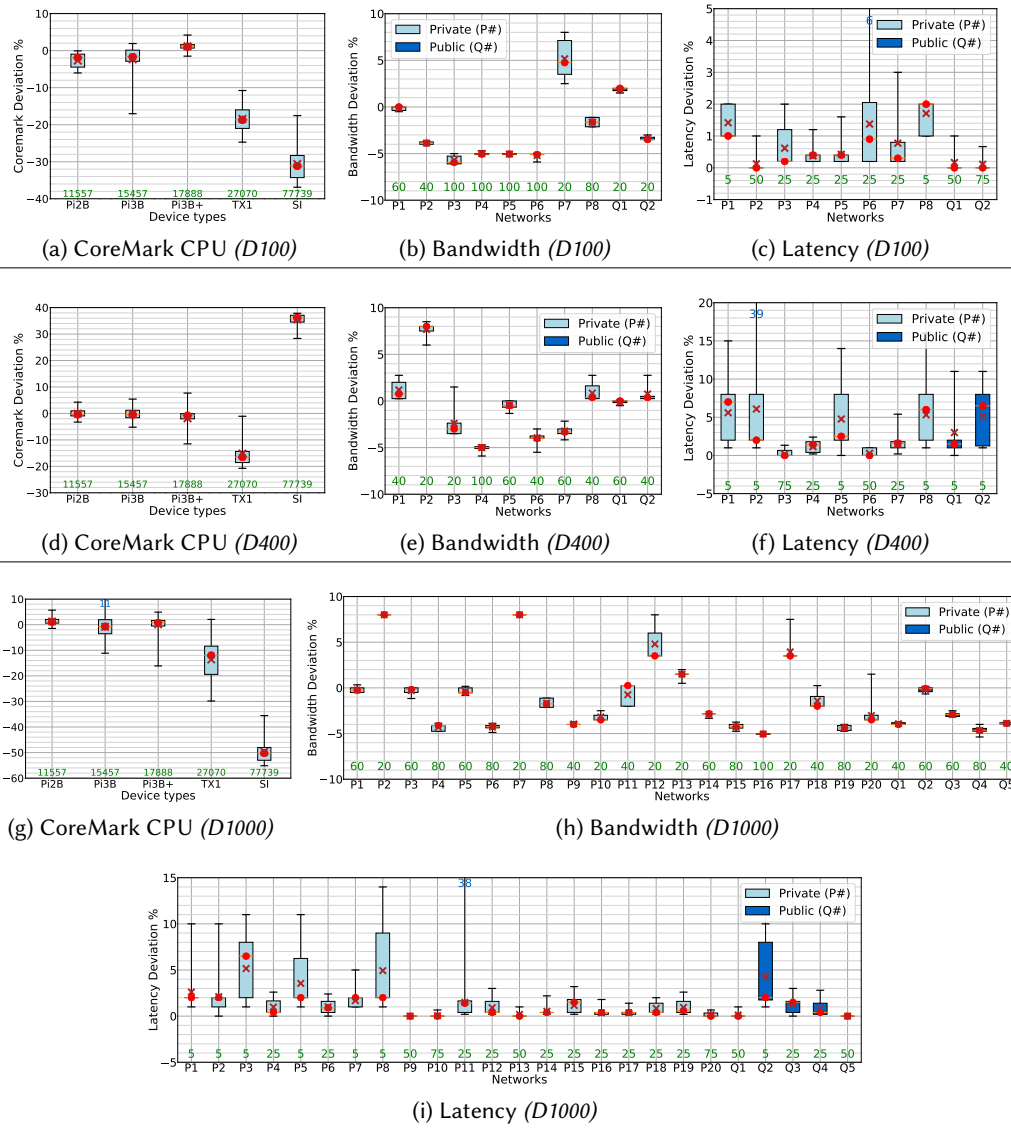
For *VIoLET*, the median deviation% ranges from 40.6% to 55.7%, which in absolute deviation is from 0.8–2.3 *ms* (Figure 14d). While the latency errors can accumulate per network hop, these values are well within 4× the median latency deviation% seen for the latency experiments above. Further, the expected ping–pong latency does not take into account the nominal message processing time, which can add to the deviation. The $Q1$–$Q3$ quartile deviation% also falls within a modest 9% band. For the *physical setup*, the median latency deviation% spans between 88.3% to 183.7% (2.3–4.6 *ms*, Figure 14d). While this too is broadly within the 4× cumulative deviation, it tends to the higher end of the range, i.e., it exhibits a higher deviation% across the board compared to VIoLET. In future, client/server experiments can be done for application protocols like CoAP too.

In summary, these CPU and network performance results validate that VIoLET can support a virtual IoT deployment comparable in behavior to the physical deployment, often erring on the side of deterministic performance rather than variability, and tending relatively closer to the peak performance. This gives confidence to users on the practical viability of our framework.

4.3 Evaluation of VE with D100, D400 and D1000 Devices

A key proposition of VIoLET is its ability to model large device deployments in an effortless and cost-effective manner. We evaluate this by configuring three VE deployments with 100, 400 and 1000 devices each (*D100* and *D400* and *D1000*). These deployments are generated using our synthetic infrastructure generator tool where we specify the number of networks and devices, mix of device types, and the possible bandwidths and latencies for the links. In these deployments, D100 and D400 have 8 private networks and 2 public networks, while D1000 has 20 private and 5 public networks. Each private network has 12 devices for D100, and 48 devices for D400 and D1000. Since the accuracy of modeling the network is sensitive to the BDP, identify the following *(bandwidth, latency)* pairs with deviations within −5% to 10% from the benchmarks, and randomly pick one for each private and public network: $[(20, 5), (40, 5), (60, 5), (80, 5), (100, 5), (20, 25), (40, 25), (60, 25), (80, 25), (100, 25), (20, 50), (40, 50), (20, 75)]$. Table 2 lists the count of device of each type, with ≈ 50% being Pi 2Bs, ≈ 30% Pi 3Bs and the rest a mix of Pi 3B+, TX1 and Softiron. These are hosted on 5 and 20 D16 VMs for D100 and D400, and 25 D64 VMs for D1000, with the latter having a cumulative *1600 Intel vCPUs on Azure* modeling over *4000 device cores*. For the largest D1000 deployment, the VMs together cost US$84/hour, which is much cheaper than the cost of US$66,000 for purchasing these devices, on an hourly basis. More details of these VEs are given in Appendix C, and their costs in comparison to an equivalent physical deployment are provided in Appendix E.

We run *CoreMark* for 20 minutes on all devices concurrently and report the violin plot for the deviation% for each of the five device types in Figs. 8a, 8d and 8g, for D100, D400 and D1000. We see that for the three types of Pis on all deployments, the median CPU deviation% is within ±3 and having a narrow box distribution as well. This indicates that even with 985 Pi containers running across the 25 VMs for D1000, the VE is able to provide accurate CPU performance concurrently across all the devices as measured by CoreMark. This deviation is even better than seen in the D25 micro-benchmarks in Fig. 6a.



(a) CoreMark CPU *(D100)* (b) Bandwidth *(D100)* (c) Latency *(D100)*

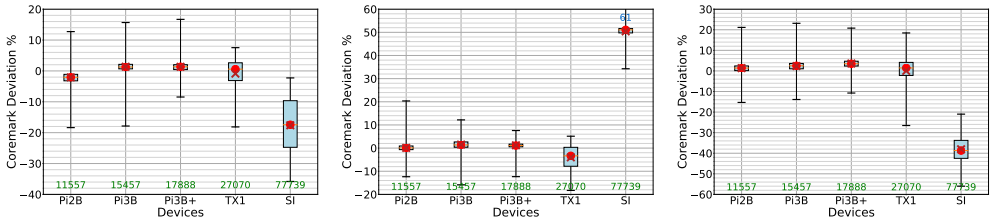
(d) CoreMark CPU *(D400)* (e) Bandwidth *(D400)* (f) Latency *(D400)*

(g) CoreMark CPU *(D1000)* (h) Bandwidth *(D1000)*

(i) Latency *(D1000)*

Fig. 8. Box plot of *deviation%* of the VIoLET containers from their expected performance, in the **D100** (top row), **D400** (second row) and **D1000** (bottom rows) setups. The expected performance is labeled in green at the bottom. The median is a red circle and the mean a red cross. A blue label at the top indicates the maximum value of the whisker, if truncated.

However, the deviation is worse for TX1 and SI. For TX1, we see a consistent under-performance in all three deployments, with a median deviation% of up to −18.7%, seen in D400. The median deviation% for SI is as bad as −49.6% for D1000, but exhibits an over-performance for D400, with a median 36.1%. There are two factors at play here. As the CoreMark and CPU allocation for a container increases, the deviation also increases. This was also seen in the earlier micro-benchmark where the deviation of TX1 was worse than the Pis, and this is further exacerbated for SI. Another factor is the CPU utilization of the VMs based on the mapping of containers to a VM. We observe



(a) D100 (b) D400 (c) D1000

Fig. 9. Box plot of CoreMark deviation% by device types, for all VE deployments with CPU dynamism

that with low CPU utilization, the containers over-perform and vice versa. This is why the single SI device in D400 was faster than expected – the partitioning logic assigned a single SI container exclusively to the entire VM.

We conduct network experiments similar to D25 (Section 4.2.3). We measure the *observed bandwidth* using `iperf3` from $\frac{n}{2}$ links uniformly sampled from $n(n-1)$ links for each network, and measure the *observed latency* using `fping` between a random $2n$ of the $n(n-1)$ pairs of devices within each private and public network. We plot the bandwidth and latency distributions for all devices in each private or public network as box plots, in Figs. 8b, 8c, 8e, 8f, 8h and 8i, for the three VE deployments.

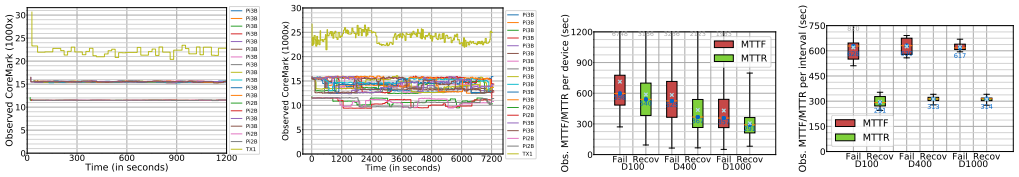
The *median bandwidth deviation* falls within ±5% with a narrow distribution in most cases, for all the deployments. This is consistent with the results from the earlier micro-benchmarks for D25 (Fig. 7b), and scales to such larger deployments with many more devices and networks as well. Occasionally, the bandwidth observed is greater than expected and the median falls between 5–10%, for network P2 in D400 and network P2 and P7 in D1000. These three are the only networks in all the deployments with a low bandwidth and a low latency setup of $20Mbps/5ms$. So, for a given BDP and low latency, the bandwidth relaxes to a slightly higher value but causes a larger % change. This too is similar to Fig. 4b from before.

The *latency deviation* is also fairly small and tight at ±2% for the median in most networks and ±7% for all of them, across deployments. The few networks for which the latency is at the higher end are consistently the ones where the latency is low at 5 *ms* and the bandwidth is higher at $60Mbps$ or $80Mbps$. As seen earlier in Fig. 4a, lower latencies tend to exhibit a larger % deviation as they are sensitive to even small absolute changes. Interestingly, the latency and bandwidth deviation% do not increase as the number of VM on which the containers of a network are placed increases. The low intra-VM latency of < 1 *ms*, high bandwidth of 46–51 *Gbps* and the network-aware placement algorithm we use is responsible for this.

In summary, the key performance characteristics of CPU and network behavior are consistent in these large scale VIoLET deployments of up to 1000 devices and 48 networks on up to 25 VMs. The high device and network accuracy matches the observations from the smaller benchmarks. This confirms the scalability of our framework to large-scale real-world VE deployments that is otherwise infeasible.

4.4 Evaluation of Dynamism and Reliability

4.4.1 CPU Dynamism of Devices. We evaluate the ability of VIoLET to introduce dynamism in the CPU performance for the devices in the VE at runtime, as described in Section 3.5.1. We configure the three VEs with a peak CPU variability of $\delta = 0.2$ and a variability period of $\pi = 600$ *secs* for all device types. So each device is expected to switch its CPU performance to a value between



(a) Without dynamism (b) With dynamism (a) Per device (b) Per interval

Fig. 10. Observed CoreMark, with and without *CPU dynamism*, for 19 devices placed on VM #3 of D100

Fig. 11. MTTF and MTTR observed, per device (space) and per interval (time)

80–100% of its rated performance every 10 *mins*. The control interval is $\epsilon = 30$ *secs*, i.e., a random subset of devices will exhibit CPU dynamism every 30 *secs*. We run the experiment for 2 *hours*, with a prior 20 *mins* of running the VE without dynamism enabled. Within each device container, we run CoreMark continuously in a loop for the entire period and also observe the CPU usage for each container from the host VM. We plot the deviation% of the CoreMark for each device when considering its dynamic CPU performance, i.e., if a device is supposed to operate at 93% of its speed due to the dynamism during a time period, then the expected performance is 93% and the observed is compared against this for the deviation. We plot the CoreMark deviation% per device type at every control interval as a distribution in Fig. 9 for D100, D400 and D1000.

For all device types except SI, the deviation% for dynamic CPU performance is ±4% in all three deployments. So the VIoLET framework is able to accurately introduce dynamism as configured by the user model. However, for SI, the deviation% is worse but consistent with its behavior in the non-dynamic scenario (Fig. 8). While D100 and D1000 show a median deviation of $\approx -20\%$ and $\approx -40\%$, D400 exhibits $\approx +50\%$ due to the container being scheduled exclusively on a VM. Interestingly, in the latter case, despite a higher deviation%, the Pearson's correlation coefficient between the expected and the observed CoreMark is a median of $\rho = 0.98$ across the execution time period, i.e., the performance shifts-up by a constant but the dynamism is enacted on this offset value accurately. The median is $\rho \geq 0.95$ for the Pi devices, and $\rho = 0.98$ for TX1, in D400.

In fact, Fig. 15 in Appendix F shows a timeline of the observed CPU% utilization, as measured at the host VM, against the expected CPU performance, for four sample containers of different device types during the 2 *hour* dynamism period for D100. As we see, the Pi devices have a CPU usage that closely matches the expected, while the TX1 broadly follows the trend. The SI shows more muted changes that are not correlated with the expected dynamism. We also see that the CPU speed changes 10–13 times in this 120 *min* time period (discrete steps in the *expected CPU* line), matching the expectation of about one change every 10 *mins* based on the uniform sampling.

This active variation is even more apparent in Fig. 10b where we plot the observed CoreMark for all 19 devices placed on one of the five D32 VMs in the D100 deployment. Here, we see that the dynamism is triggered every 30 *secs* across the entire VE. The three bands that are formed, at $\approx 10k$, $15k$ and $25k$ CoreMark indicate the 80–100% range for the three device types, Pi2B, Pi3B and TX1 shown here. In contrast, Fig. 10a illustrates the prior 20 *mins* for these same devices with dynamism disabled, and their CoreMark variation is barely existent for the Pis and episodic for the TX1. So there is a real impact of modeling and introducing CPU dynamism.

Also, these changes happen instantaneously, i.e., there is no delay between the CPU variation being requested at a control interval by VIoLET from a container and the change occurring. The applications running within the container are also not impacted in any manner and continue running, other than observing a CPU performance variation. So this makes VIoLET an effective

framework to model and validate the behavior of applications, IoT platforms and even scheduling strategies under diverse conditions of CPU variations in the VE.

4.4.2 Device Reliability. Lastly, we validate the ability of VIoLET to simulate device failures and recovery, as was discussed in Section 3.5.2. The experiment setup is similar to the one for CPU dynamism. We set the MTTF for all devices as $\mu = 600$ *secs*, and the MTTR as $\lambda = 300$ *secs*. The control interval is $\epsilon = 30$ *secs*. So each device has a $\frac{\epsilon}{\mu} = 5\%$ chance of failing at each interval, and these are independent for each device in the deployment. Similarly, each failed device has a $\frac{\epsilon}{\lambda} = 10\%$ chance of recovering at each interval. The experiment is run for 2 *hours* for each of the D100, D400 and D1000 deployments.

Fig. 11a shows the distribution of the average observed MTTF and MTTR for devices in each deployment during the complete run. Further, Fig. 11b shows the distribution of the average observed MTTF and MTTR for devices selected for failure and for recovery, at each of the 240 control intervals. The median observed MTTF for the devices is 598 *secs*, 622 *secs* and 623 *secs* for the D100, D400 and D1000 deployments, and similarly, MTTR is 292 *secs*, 304 *secs* and 302 *secs*. These closely match the expected values of 600 *secs* and 300 *secs* across a device's lifetime. There are some long-tail outliers seen by the whiskers, which indicate that some devices did not fail for a long time. The observed MTTF and MTTR for devices selected at each control interval are also comparable. These suggest that these reliability metrics operate independently across space and time, as is typically the case in the real-world. The timeline of failures and recovery for all devices is shown in Appendix G for a similar experiment conducted on the D25 setup.

5 RELATED WORK

The growing interest in IoT and edge/fog computing has given rise to several *simulation environments* [37, 38]. *iFogSim* [13] extends the prior work on CloudSim [39] to simulate the behavior of applications over fog devices, sensors and actuators that are connected by a network topology. Users define the compute, network and energy profiles of fog devices, and the properties and distributions of tuples from sensors. Users can define DAG-based applications with tasks consuming compute capacity and bandwidth, and their execution over the fog network is simulated using an extensible resource manager. The goal is to evaluate different scheduling strategies synthetically. *YAFS* is another discrete-event simulator that allows for the definition of complex graph-based IoT network topology, with fog nodes as entities with resource characteristics. Application DAGs, messages and placement policies for tasks to devices can also be specified. They offer event logging mechanisms for posterior analysis. Our VIoLET emulator similarly lets users define devices, complex network graphs and sensors, but *actually instantiates* them rather than simulate them. While we do not expose an application model, this is not necessary since users can bring in their actual applications, schedulers, policies, etc. to be evaluated on the virtual environment that matches reality.

Edgecloudsim [14] offers similar simulation capabilities but also introduces mobility models for the edge. They simulate network characteristics like transmission delay for LAN and WAN, and also task failures due to mobility for a single use-case. In a similar spirit, *MobFogSim* also extends iFogSim to simulate the mobility of devices and the migration of services among fog devices. It is used to validate the usefulness of different service migration strategies [40]. VIoLET allows services to directly be run on the device containers and migrated, and enables CPU dynamism and device failures. But it does not support device mobility, and in the future, this can be reflected in a network dynamism model.

In the commercial space, city-scale simulators for IoT deployments in smart cities are available [15]. These mimic the behavior of not just devices, sensors, actuators and the network, but also application services like MQTT broker and CoAP services that may be hosted. These offer

a comprehensive simulation environment for city-planners to perform what-if analysis on the models. Our proposed VIoLET environment goes a step further and permits realistic devices and networks to be emulated on elastic cloud VMs and applications themselves to be executed without physically deploying the field devices.

There has been some work on *emulation of IoT and CPS environments*. They either focus on the computing layer and emulate devices using virtual machines [17, 20], TinyOS [18] and Docker [23], or emphasize accurate the network emulation [19, 38]. *ELIoT* emulates an IoT platform to validate the CoAP stack, where Docker containers host the CoAP servers implemented using an Eclipse Leshan M2M server. While this helps validate CoAP execution patterns, payloads like JSON and CBOR, and DTLS-based security, it is limited only to CoAP applications. Also, there is no network topology, device/network performance control or resource dynamism. Others [20] combine a VM running the ContikiOS to emulate an embedded device with the COOJA wireless network simulator [41]. This allows CoAP applications on the VM to communicate through COOJA that handles network packet control and transmission based on different models. Here again, there is no attempt to accurately model the device performance or dynamism, or scale this to a large number (100s) of devices. VIoLET attempts to balance both these goals. It uses light-weight Docker containers benchmarked to real IoT devices and run real IoT and CPS applications, combined with diverse network topologies and bandwidth/latency shaping, for 100s of devices at scale.

Simulators and emulators are also popular in other domains, such as cloud, network and Software-Defined Network (SDN) simulators [31, 38, 39, 42]. Discrete event *network simulators* like *ns-3* attempt to mimic the performance of diverse networks and Internet protocols, in a controlled and reproducible manner [30]. But they do not really send data packets between processes. *Mininet* [31] is a popular network virtualization tool that allows complex network topologies and SDNs to be designed, and real data flows through. But its use of Linux *process groups* for executing applications does not restrict the CPU and memory for a host. VIoLET's use of Linux *cgroups* in Docker enables such resource constraints that emulate the performance of physical devices. We also allow the use of custom device images with pre-deployed libraries and software. Also, support for executing on distributed machines is limited in MiniNet, with alternatives like *MaxiNet* proposed. VIoLET is designed to efficiently scale to 1000s of realistic devices on 10–100s of VMs. Others like *COOJA* [41] extend the network simulation to the application layer for the Contiki OS. In the future, one could combine the capabilities of advanced network simulators and VIoLET in a single platform.

Digital twins and *device shadows* are emerging concepts that allow a simulation or a virtual service to act as a proxy for the live physical devices in an IoT or CPS deployment [43]. They enable lifecycle management of the physical asset by just interacting with the digital twin, which in turn may have uni- or bi-direction communication with the physical asset [44]. A VE like VIoLET can serve as a core building-block to model such digital twins, and enable what-if analysis or to replicate faults for root-cause analysis.

Others have proposed IoT data stream and application workloads for evaluating big data platforms, particularly stream processing ones. Here, large-scale sensor data is simulated with realistic distributions [45, 46]. These can be used in place of the synthetic sensor streams that we provide. Our prior work has proposed complex event processing application workloads for IoT domains [11]. These can use VIoLET for evaluating their execution on edge, fog and cloud resources.

Google's Kubernetes [47] is a multi-node orchestration platform for container life-cycle management. It schedules containers across nodes to balance the load, but is not aware of network topology overlays on the containers. It can potentially be a replacement for Docker in VIoLET.

In summary, the uniqueness of VIoLET lies in its ability to emulate the cyber-infrastructure of IoT and CPS domains at large-scales, allowing users to actually deploy and run their application within device containers whose performance matches physical devices, and move data between

Table 3. Feature matrix of using physical systems, simulators and the VIoLET emulation environment

Type	Deploy-ability	Manage-ability	Flexibility	Low CapEx	Low OpEx	CPU Accuracy	NW Accuracy	Run Code	Move Data
Physical system	○	○	○	○	◐	●	●	●	●
Simulation	●	●	●	●	●	◐	◐	○	○
VIoLET	●	●	●	●	○	●	●	●	●

them over network overlays whose behavior resembles physical networks. Table 3 summarizes the relative benefits of a testbed with physical devices, an emulation environment like VIoLET, and a simulator. It contrasts them along the dimensions of: accuracy of the devices and network; ease of deployment; ease of management; flexibility in designing custom environments; capital and operational costs; ability to run real software; and ability to move real data between devices. As we see, VIoLET attempts to achieve the best of physical systems – with high accuracy in the compute and network capabilities, and allowing real code to run; and of simulations – with ease of deployment, management and flexibility, and low capital costs. Its modest operational costs are much less than the capital costs for a physical setup (Appendix E) and only billed when used.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have motivated the design requirements for an IoT *emulation* environment and proposed VIoLET to meet these needs. VIoLET allows users to declaratively create virtual edge, fog and cloud devices as containers that are connected through user-defined network topologies. This allows developers to run real cyber-physical platforms and applications, with real data flowing through the network, and including virtual sensors that can connect to external physical system simulators. This also offers first-hand knowledge of the performance, scalability and security of the user's applications, IoT platforms or scheduling algorithms, similar to a real IoT deployment, unlike other *simulators* that currently exist. It is as simple to deploy and run as a simulation environment, but provides the realism of physical systems, with ease and reproducibility. Our extensive experiments show that VIoLET is *accurate*, with performance statistically matching physical IoT deployments, and it scales to 1000s of devices. It is also affordable, costing just US$84/hour to deploy 1000 devices on the Cloud. VIoLET is an essential tool for CPS researchers to validate their cyber-environment, and for IoT managers to virtually test various software stacks and network deployment models.

There are several extensions possible to VIoLET. One of our limitations is that only devices for which Docker can launch container environments are feasible. While any device container that runs a standard Linux kernel using `cgroups` (or even a Windows device [13]) can be run, this limits the use of edge micro-controllers like Arduino, or wireless IoT motes that run real-time OS. Also, leveraging Docker's support for GPUs in the future will help users make use of accelerators present in devices like NVidia TX1 [14]. It may also be useful to migrate to Kubernetes for container orchestration rather than Docker due to its growing support [47]. There is also the opportunity to pack containers more efficiently to reduce the cloud costs, including over-packing when devices will not be pushed to their full utilization [48].

Modeling the energy usage of devices and networks is vital for energy-constrained IoT devices. As future work, we propose to include energy profiling based on CPU, memory, IO and network usage by applications hosted within containers, along with a model for battery charging and discharging. This will help evaluate energy-aware schedulers and energy-performance trade-offs.

[13] Docker for Windows, https://docs.docker.com/docker-for-windows/

[14] GPU-enabled Docker Containers, https://github.com/NVIDIA/nvidia-docker

Our network configurations focus on the visibility of public and private networks, and the bandwidth and latency of the links. However, it does not yet handle protocols at the transport layer or below, which is necessary to emulate IoT wireless networks like LoRA, 6LoWPAN, etc. This will require interfacing VIoLET with a network emulator such as mininet in the future. Modeling variability in bandwidth, latency and link failures, and device mobility are also planned.

7 ACKNOWLEDGMENTS

This work is supported by research grants from IUSSTF, VMWare and MHRD, and by cloud credits from Microsoft Azure. We thank DREAM:Lab members, Rahul Bhope, Shriram R., Siva Prakash, Aakash Khochare and Abhilash Sharma, for design discussions and assistance with experiments. We also thank the reviewers of Euro-Par 2018 for their detailed comments on the conference version of the article, and for recognizing it as a *Distinguished Paper*.

REFERENCES

[1] Y. Simmhan, P. Ravindra, S. Chaturvedi, M. Hegde, R. Ballamajalu, Towards a data-driven iot software architecture for smart city utilities, Software: Practice and Experience 48 (7) (2018) 1390–1416. doi:10.1002/spe.2580.

[2] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky, et al., An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance, in: ACM/IEEE Symposium on Edge Computing, 2017, p. 14.

[3] D. F. Pigatto, M. Rodrigues, J. V. de Carvalho Fontes, A. S. R. Pinto, J. Smith, K. R. L. J. C. Branco, The Internet of Flying Things, John Wiley & Sons, Ltd, 2018, Ch. 19, pp. 529–562. doi:10.1002/9781119456735.ch19.

[4] P. Varshney, Y. Simmhan, Characterizing application scheduling on edge, fog and cloud computing resources, Software: Practice and Experience 50 (5) (2020) 558–595. doi:10.1002/spe.2699.

[5] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: ACM Workshop on Mobile Cloud Computing (MCC), 2012.

[6] Y. Simmhan, S. Aman, A. Kumbhare, R. Liu, S. Stevens, Q. Zhou, V. Prasanna, Cloud-based software platform for big data analytics in smart grids, Computing in Science and Engineering 15 (4) (2013) 38 – 47. doi:10.1109/MCSE.2013.39.

[7] M. Satyanarayanan, et al., Edge analytics in the internet of things, IEEE Pervasive Computing 14 (2) (2015) 24–31.

[8] M. Yannuzzi, F. van Lingen, A. Jain, O. L. Parellada, M. M. Flores, D. Carrera, J. L. Pérez, D. Montero, P. Chacin, A. Corsaro, et al., A new era for cities with fog computing, IEEE Internet Computing 21 (2) (2017) 54–67.

[9] F. Van Lingen, M. Yannuzzi, A. Jain, R. Irons-Mclean, O. Lluch, D. Carrera, J. L. Perez, A. Gutierrez, D. Montero, J. Marti, et al., The unavoidable convergence of nfv, 5g, and fog: a model-driven approach to bridge cloud and edge, IEEE Communications Magazine 55 (8) (2017) 28–35.

[10] A. V. Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh, R. Buyya, Internet of Things: Principles and Paradigms, Morgan Kaufmann, 2016, Ch. Fog Computing: Principles, Architectures, and Applications.

[11] R. Ghosh, Y. Simmhan, Distributed scheduling of event analytics across edge and cloud, ACM Transactions on Cyber-Physical Systems 2 (4) (2018) 24.

[12] I. Lera, C. Guerrero, C. Juiz, Yafs: A simulator for iot scenarios in fog computing, IEEE Access 7 (2019) 91745–91758.

[13] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, 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 47 (9) (2017) 1275–1296.

[14] C. Sonmez, A. Ozgovde, C. Ersoy, Edgecloudsim: An environment for performance evaluation of edge computing systems, in: Fog and Mobile Edge Computing (FMEC), 2017.

[15] J. Leland, Deploy scalable smart city architectures confidently with network simulation, Tech. rep., Insight Tech (2017).

[16] S. Badiger, S. Baheti, Y. Simmhan, Violet: A large-scale virtual environment for internet of things, in: International European Conference on Parallel and Distributed Computing (EuroPar), 2018, Distinguished Paper Award.

[17] Y. Zhang, F. Xie, Y. Dong, G. Yang, X. Zhou, High fidelity virtualization of cyber-physical systems, International Journal of Modeling, Simulation, and Scientific Computing 4 (02) (2013) 1340005.

[18] B. Wang, J. S. Baras, Hybridsim: A modeling and co-simulation toolchain for cyber-physical systems, in: IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, 2013, pp. 33–40.

[19] D. Antonioli, N. O. Tippenhauer, Minicps: A toolkit for security research on cps networks, in: ACM workshop on cyber-physical systems-security and/or privacy, 2015, pp. 91–100.

[20] A. W. Abbas, S. N. K. Marwat, Scalable emulated framework for iot devices in smart logistics based cyber-physical systems: Bonded coverage and connectivity analysis, IEEE Access 8 (2020) 138350–138372.

[21] D. Krajzewicz, J. Erdmann, M. Behrisch, L. Bieker, Recent development and applications of sumo-simulation of urban mobility, International journal on advances in systems and measurements 5 (3&4).
[22] J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, O. Von Stryk, Comprehensive simulation of quadrotor uavs using ros and gazebo, in: International conference on simulation, modeling, and programming for autonomous robots, 2012.
[23] A. Mäkinen, J. Jiménez, R. Morabito, Eliot: Design of an emulated iot platform, in: IEEE International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), 2017, pp. 1–7.
[24] Docker Inc., Docker resource constraints, https://docs.docker.com/config/containers/resource_constraints/ (2019).
[25] Embedded Microprocessor Benchmark Consortium (EEMBC), Coremark v1.0, http://coremark.org (2017).
[26] R. Longbottom, Whetstone benchmark history and results, http://www.roylongbottom.org.uk/whetstone.htm (2019).
[27] J. J. Dongarra, P. Luszczek, A. Petitet, The linpack benchmark: past, present and future, Concurrency and Computation: practice and experience 15 (9) (2003) 803–820.
[28] M. A. Brown, Traffic control in linux, v1.0.2, https://www.tldp.org/HOWTO/Traffic-Control-HOWTO/ (2019).
[29] Docker Inc., Multi-host networking with standalone swarms, https://docs.docker.com/network/overlay-standalone.swarm/ (2019).
[30] ns-3 manual: A discrete-event network simulator, Tech. rep., University of Washington NS-3 Consortium, https://www.nsnam.org/docs/release/3.32/manual/ns-3-manual.pdf (2020).
[31] B. Lantz, B. Heller, N. McKeown, A network in a laptop: Rapid prototyping for software-defined networks, in: Workshop on Hot Topics in Networks, 2010.
[32] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. on Scientific Computing 20 (1) (1998) 359–392.
[33] P. Ravindra, A. Khochare, S. P. Reddy, S. Sharma, P. Varshney, Y. Simmhan, Echo: An adaptive orchestration platform for hybrid dataflows across cloud and edge, in: International Conference on Service-Oriented Computing, Springer, 2017, pp. 395–410.
[34] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Levendovszky, Á. Lédeczi, Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure, MPM@ MoDELS 1237 (2014) 41–60.
[35] Microsoft, Ntttcp-for-linux, a linux network throughput multiple-thread benchmark tool, https://github.com/Microsoft/ntttcp-for-linux (2019).
[36] V. Jacobson, Congestion avoidance and control, in: ACM SIGCOMM computer communication review, Vol. 18, ACM, 1988, pp. 314–329.
[37] D. P. Abreu, K. Velasquez, M. Curado, E. Monteiro, A comparative analysis of simulators for the cloud to fog continuum, Simulation Modelling Practice and Theory 101 (2020) 102029.
[38] M. Chernyshev, Z. Baig, O. Bello, S. Zeadally, Internet of things (iot): Research, simulators, and testbeds, IEEE Internet of Things Journal 5 (3) (2017) 1637–1647.
[39] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, R. Buyya, Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Software: Practice and experience 41 (1) (2011) 23–50.
[40] C. Puliafito, D. M. Gonçalves, M. M. Lopes, L. L. Martins, E. Madeira, E. Mingozzi, O. Rana, L. F. Bittencourt, Mobfogsim: Simulation of mobility and migration for fog computing, Simulation Modelling Practice and Theory 101 (2020) 102062.
[41] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, T. Voigt, Cross-level sensor network simulation with cooja, in: IEEE Conference on Local Computer Networks, 2006, pp. 641–648.
[42] T. R. Henderson, S. Roy, S. Floyd, G. F. Riley, Ns-3 project goals, in: Workshop on Ns-2: The IP Network Simulator, 2006.
[43] W. Kritzinger, M. Karner, G. Traar, J. Henjes, W. Sihn, Digital twin in manufacturing: A categorical literature review and classification, IFAC-PapersOnLine 51 (11) (2018) 1016–1022.
[44] M. Dietz, G. Pernul, Digital twin: Empowering enterprises towards a system-of-systems approach, Business & Information Systems Engineering 62 (2) (2020) 179–184.
[45] M. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, B. Vandiver, Iotabench: an internet of things analytics benchmark, in: International Conference on Performance Engineering (ICPE), 2015.
[46] L. Gu, M. Zhou, Z. Zhang, M.-C. Shan, A. Zhou, M. Winslett, Chronos: An elastic parallel framework for stream benchmark generation and simulation, in: IEEE International Conference on Data Engineering (ICDE), 2015.
[47] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes, Borg, omega, and kubernetes, ACM Queue 14 (1).
[48] U. Awada, A. Barker, Improving resource efficiency of container-instance clusters on clouds, in: Cluster, Cloud and Grid Computing (CCGRID), 2017.

APPENDIX

A JSON Configuration Document

Listing 1. Device Types, `device_types.json`

```
"device_types": {
  "Pi2B": {
    "core_count": "4",
    "coremark": "11557",
    "whetstone": "-1",
    "memory_mb": "1024",
    "disk_mb": "1024",
    "nic_out_bw_mbps": "100",
    "docker_image": "violet:local",
    "host_mount": "/tmp/log",
    "reliability": {
      "mttf_sec": "600",
      "mttr_sec": "300",
      "cpu_var_max": "0.2",
      "cpu_var_period_sec": "600",
      "out_bw_var_max": "0.1",
      "out_bw_var_period_secs": "1200",
      "out_lat_var_max": "0.05",
      "out_lat_var_period_secs": "1200",
      "out_packet_loss_ratio": "0.00001",
      "out_packet_corruption_ratio": "0.0001"
    }
  },
  "Pi3B": {
    "core_count": "4",
    "coremark": "15457",
    "whetstone": "-1",
    "memory_mb": "1024",
    "disk_mb": "1024",
    "nic_out_bw_mbps": "100",
    "docker_image": "violet:local",
    "host_mount": "/tmp/log",
    "reliability": {
      "mttf_sec": "600",
      "mttr_sec": "300",
      "cpu_var_max": "0.2",
      "cpu_var_period_sec": "600",
      "out_bw_var_max": "0.1",
      "out_bw_var_period_secs": "1200",
      "out_lat_var_max": "0.05",
      "out_lat_var_period_secs": "1200",
      "out_packet_loss_ratio": "0.00001",
      "out_packet_corruption_ratio": "0.0001"
    }
  },
  ...
}
```

Listing 2. Sensor Types, `sensor_types.json`

```
"sensor_types": [
  {
    "type": "accelerometer",
    "id": "true",
    "timestamp": "true",
    "dist_rate": "uniform",
    "rate_params": {
      "lower_limit": "1",
      "upper_limit": "10",
      "unit": "s"
    },
    "dist_value": "normal",
    "value_params": {
      "mean": "150",
      "variance": "50",
      "min_value": "25"
    }
  },
  {
    "type": "custom",
    "id": "true",
    "timestamp": "true",
    "dist_rate": "user_defined",
    "rate_params": {
      "path": "sensors_data_gen/data/time.csv",
      "unit": "s"
    },
    "dist_value": "user_defined",
    "value_params": {
      "path": "sensors_data_gen/data/data.csv"
    }
  },
  ...
]
```

Listing 3. Infrastructure Generation, `infra_gen.json`

```
"infra_gen": {
  "public_networks":
  {
    "violet_public_1":
    {
      "devices":
      [
        {
          "device_type":"Pi2B",
          "number_devices":"4",
          "number_sensors":"5"
        },
        {
          "device_type":"Pi3B",
          "number_devices":"2",
          "number_sensors":"5"
        }
        ],
        "private_networks_list":
        [
          "violet_private_1",
          "violet_private_2",
          "violet_private_3",
          "violet_private_4"
        ]
      },
      ...
  }
  "private_networks":
  {
    "violet_private_1":
    {
      "device_type":"Pi2B",
      "number_devices":"11",
      "gateway_device_type":"Pi2B",
      "number_sensors":"5"
    },
    "violet_private_2":
    {
      "device_type":"Pi3B",
      "number_devices":"11",
      "gateway_device_type":"Pi3B",
      "number_sensors":"5"
    },
    ...
  },
  "network":
  {
    "public_networks":
    {
      "bandwidth_mbps":["10", "15", "20", "25", "30",
            ↪ "35", "40", "45", "50"],
      "latency_ms":["10", "15", "20", "25", "30", "35",
            ↪ "40", "45", "50"],
      "window_size_bits":"524288"
    },
    "private_networks":
    {
      "bandwidth_mbps":["10", "15", "20", "25", "30",
            ↪ "35", "40" ,"45", "50"],
      "latency_ms":["10", "15", "20", "25", "30", "35",
            ↪ "40", "45", "50"],
      "window_size_bits":"524288"
    }
  }
}
```

Listing 4. Infrastructure Configuration, `infra_config.json`

```
"infra_config": {
  "public_global_network": {
    "subnet": "10.0.3.0/24",
    "bandwidth_mbps": 100,
    "ip_range": "10.0.3.0/24",
    "devices": [
      "Fog-1.1", "Fog-1.2", "Fog-1.3", "Fog-1.4",
            ↪ "Fog-1.5", "Fog-1.6", "Fog-2.1",
            ↪ "Fog-2.2", "Fog-2.3", "Fog-2.4",
            ↪ "Fog-2.5", "Fog-2.6"
    ],
    "latency_ms": 5
  },
  "block_network_route": [

  ],
  "public_networks": {
    "violet_public_1": {
      "subnet": "10.0.1.0/24",
      "bandwidth_mbps": 20,
      "ip_range": "10.0.1.0/24",
      "devices": [
        "Fog-1.1", "Fog-1.2", "Fog-1.3", "Fog-1.4",
            ↪ "Fog-1.5", "Fog-1.6"
      ],
      "latency_ms": 50
    },
    ...
  },
  "private_networks": {
    "violet_private_1": {
      "subnet": "192.168.1.0/24",
      "bandwidth_mbps": 60,
      "latency_ms": 5,
      "devices": [
        "Edge-1.1", "Edge-1.2", "Edge-1.3", "Edge-1.4",
            ↪ "Edge-1.5", "Edge-1.6", "Edge-1.7",
            ↪ "Edge-1.8", "Edge-1.9", "Edge-1.10",
            ↪ "Edge-1.11"
      ],
      "ip_range": "192.168.1.0/24",
      "gateway": "Fog-1.1"
    },
    ...
  },
  "devices": {
    "Edge-5.8": {
      "sensors": [
      {
        "count": 2,
        "sensor_type": "accelerometer"
      },
      {
        "count": 2,
        "sensor_type": "gyroscope"
      },
      {
        "count": 1,
        "sensor_type": "custom"
      }
      ],
      "port": 37893,
      "device_type": "Pi2B"
    },
    ...
  }
  ...
}
```

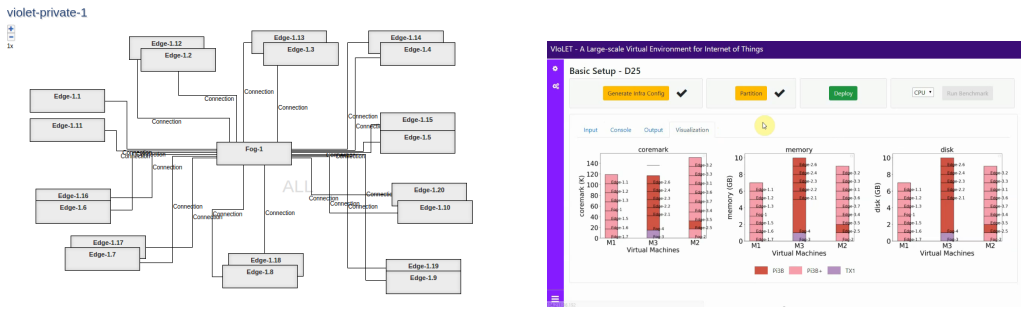
Listing 5. VM Types, `vm_types.json`


```
"vm_types": {
  "Standard_D4_v2": {
    "core_count": "8",
    "coremark": "102341",
    "whetstone": "-1",
    "memory_mb": "28672",
    "disk_mb": "409600",
    "out_bw_mbps": "2048",
    "out_latency_ms": "1.11",
    "local_bw_mbps": "39116",
    "local_latency_ms": "0.053",
    "shared_mount": "/tmp/log/"
  },
  "Standard_D16_v3": {
    "core_count": "16",
    "coremark": "163767",
    "whetstone": "-1",
    "memory_mb": "63488",
    "disk_mb": "409600",
    "out_bw_mbps": "2048",
    "out_latency_ms": "1.11",
    "local_bw_mbps": "39116",
    "local_latency_ms": "0.053",
    "shared_mount": "/tmp/log/"
  },
  "Standard_D32_v3": {
    "core_count": "32",
    "coremark": "301269",
    "whetstone": "-1",
    "memory_mb": "128000",
    "disk_mb": "819200",
    "out_bw_mbps": "2990",
    "out_latency_ms": "0.45",
    "local_bw_mbps": "28364",
    "local_latency_ms": "0.018",
    "shared_mount": "/tmp/log/"
  },
  ...
}
```


Listing 6. VM Configuration, `vm_instances.json`


```
"vm_config": {
  "admin_VM": {
    "VIoLET_admin": {
      "hostname_ip": "10.0.0.4",
      "key_path": "/home/centos/id_rsa",
      "user": "centos"
    }
  },
  "container_VM": {
    "AEP-1": {
      "hostname_ip": "10.0.0.9",
      "key_path": "/home/centos/aep_rsa",
      "user": "centos",
      "vm_type": "Standard_D32_v3"
    },
    "AEP-2": {
      "hostname_ip": "10.0.0.10",
      "key_path": "/home/centos/id_rsa",
      "user": "centos",
      "vm_type": "Standard_D32_v3"
    },
    "AEP-3": {
      "hostname_ip": "10.0.0.11",
      "key_path": "/home/centos/aep_rsa",
      "user": "centos",
      "vm_type": "Standard_D32_v3"
    },
    "AEP-4": {
      "hostname_ip": "10.0.0.12",
      "key_path": "/home/centos/aep_rsa",
      "user": "centos",
      "vm_type": "Standard_D32_v3"
    },
    "AEP-5": {
      "hostname_ip": "10.0.0.13",
      "key_path": "/home/centos/aep_rsa",
      "user": "centos",
      "vm_type": "Standard_D32_v3"
    }
  }
}
```


B Composition and Deployment User Interfaces



(a) Visual composition using WebGME DSML (b) Deployment Management Portal

Fig. 12. Web-based user interfaces for enhanced usability of VIoLET

C VE Configuration for the Larger Deployments

Table 4. Configuration of private (P_i) and public (Q_i) networks in **D100** deployment

Network	Device type	Device count [15]	Expected Bandwidth (Mbps)	Expected Latency (ms)
P1	Pi2B	12	60	5
P2	Pi3B	12	40	50
P3	Pi2B	12	100	25
P4	Pi3B	12	100	25
P5	Pi2B	12	100	25
P6	Pi3B	12	100	25
P7	Pi2B	11	20	25
	TX1	1		
P8	Pi3B+	11	80	5
	SI	1		
Q1	Pi2B	4	20	50
	Pi3B	2		
Q2	Pi2B	3	20	75
	Pi3B	1		
	TX1	1		
	SI	1		

Table 5. Configuration of private (P_i) and public (Q_i) networks in **D400** deployment

Network	Device type	Device count [16]	Expected Bandwidth (Mbps)	Expected Latency (ms)
P1	Pi2B	48	40	5
P2	Pi3B	48	20	5
P3	Pi2B	48	20	75
P4	Pi2B	48	100	25
P5	Pi2B	48	60	5
P6	Pi3B	48	40	50
P7	Pi3B+	47	60	25
	TX1	1		
P8	Pi2B	47	40	5
	SI	1		
Q1	Pi2B	10	60	5
	Pi3B	1		
Q2	Pi2B	10	40	5
	Pi3B	1		
	TX1	1		
	SI	1		

[15]Note that the same device can be part of both a public and private network. Hence the sum of device counts will be ≥ 100 though only 100 distinct devices are present.
[16]Note that the same device can be part of both a public and private network. Hence the sum of device counts will be ≥ 400 though only 400 distinct devices are present.
[17]Note that the same device can be part of both a public and private network. Hence the sum of device counts will be ≥ 1000 though only 1000 distinct devices are present.

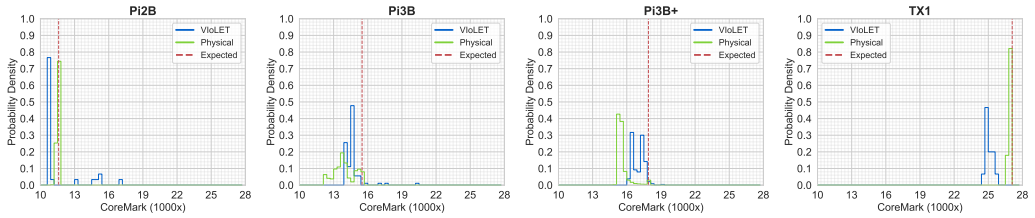
Table 6. Configuration of private (P_i) and public (Q_i) networks in **D1000** deployment

Network	Device type	Device count [17]	Expected Bandwidth (Mbps)	Expected Latency (ms)
P1	Pi2B	48	60	5
P2	Pi2B	47	20	5
	TX1	1		
P3	Pi3B	48	60	5
P4	Pi2B	48	80	25
P5	Pi3B+	48	60	5
P6	Pi2B	48	80	25
P7	Pi2B	47	20	5
	SI	1		
P8	Pi3B	48	80	5
P9	Pi2B	48	40	50
P10	Pi3B	48	20	75
P11	Pi3B+	48	40	25
P12	Pi2B	48	20	25
P13	Pi3B	48	20	50
P14	Pi2B	48	60	25
P15	Pi3B+	48	80	25
P16	Pi2B	48	100	25
P17	Pi2B	47	20	25
	TX1	1		
P18	Pi3B	48	40	25
P19	Pi2B	48	80	25
P20	Pi3B	48	20	75
Q1	Pi2B	5	40	50
	Pi3B	2		
	Pi3B+	2		
	TX1	2		
	SI	1		
Q2	Pi2B	5	60	5
	Pi3B	2		
	Pi3B+	2		
	TX1	2		
	SI	1		
Q3	Pi2B	5	60	25
	Pi3B	2		
	Pi3B+	2		
	TX1	2		
	SI	1		
Q4	Pi2B	5	80	25
	Pi3B	2		
	Pi3B+	2		
	TX1	2		
	SI	1		
Q5	Pi2B	5	40	50
	Pi3B	2		
	Pi3B+	2		
	TX1	2		
	SI	1		

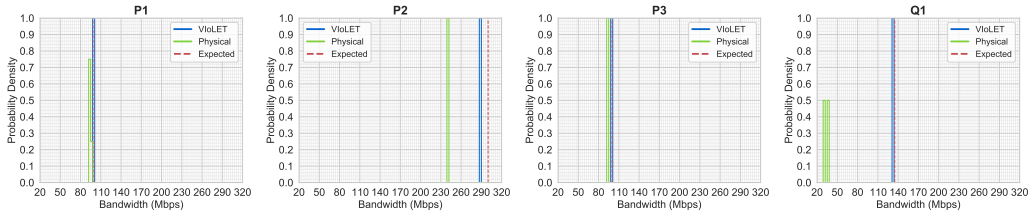
D Statistical Data for D25 Experiments

This section provides additional statistical details for the D25 physical and VIoLET experiments conducted in Section 4.2 of the main article.

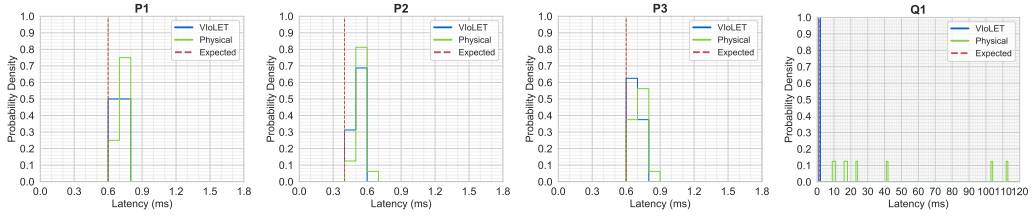
D.1 Distribution of Absolute Values. The Figures 13 show the *probability distributions* of the absolute values of CoreMark, bandwidth, latency and the application ping-pong performance benchmarks, for the D25 VIoLET and Physical configurations. These complement Figures 7 in the main article, that show the *% deviation* of these absolute values from the expected performance.



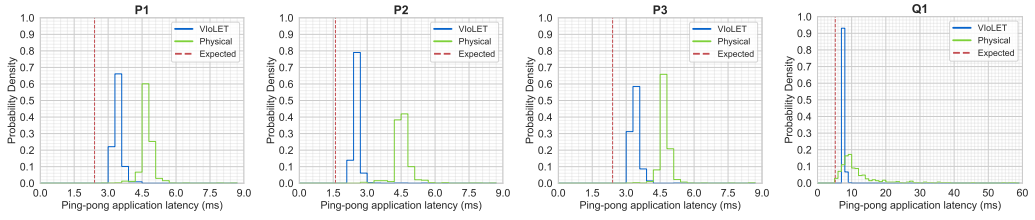
(a) CPU CoreMark



(b) Bandwidth



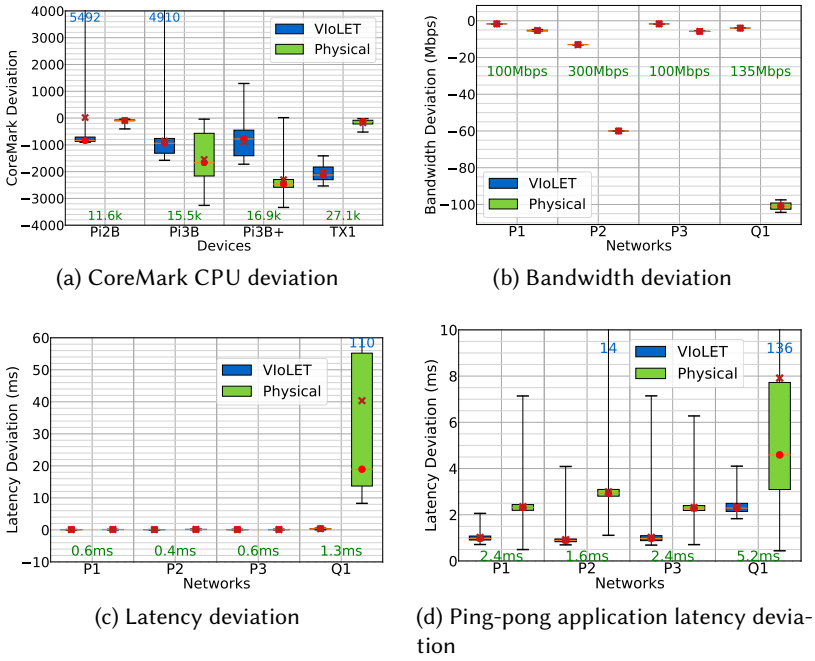
(c) Latency



(d) Ping-pong application latency

Fig. 13. *Probability distributions* for the performance benchmarks for VIoLET containers and the physical devices in the D25 setup. A red vertical dashed line indicates the expected performance.

D.2 Distribution of Absolute Deviations. The Figures 14 show the *absolute deviations* of the D25 VIoLET and Physical configurations from the ideal behavior. These complement the *% deviations* reported in the main article in Figures 7.



(a) CoreMark CPU deviation

(b) Bandwidth deviation

(c) Latency deviation

(d) Ping-pong application latency deviation

Fig. 14. Box plot of *absolute deviation* for the VIoLET containers and the physical devices from their expected performance, in the D25 setup. The expected ideal performance is labeled in green at the bottom. The median is a red circle and the mean a red cross. A blue label at the top indicates the maximum value of the whisker, if truncated.

D.3 Statistical Tests. The value distributions for the various performance metrics do not clearly correspond to a normal distribution. Further, the data points collected are not *paired* for the physical and virtual experiments since they run independently, and the number of data points collected also vary substantially, from < 10 to > 1000, depending on the experiments. So we perform a simple non-parametric *sign test* [18] to evaluate if the performance of the VIoLET virtual environment is statistically better than the median physical environment, and if the virtual and the physical environments are statistically performing within $f\%$ of the ideal performance. The *sign test* returns the statistical probability (p-value) of the hypothesis that a series of values $y \geq x$ is not just due to pure chance, on a binomial distribution.

CoreMark. We use the *sign test* to test the hypothesis that *the CoreMark of the virtual devices are greater than the median CoreMark of the physical devices.* For Pi2B and TX1, we get a p-value of $p < 0.95$, rejecting the hypothesis. In fact, the results statistically show that the physical devices out-perform the virtual devices. For Pi3B and Pi3B+, we get a p-value of $p \geq 0.99$, supporting the hypothesis and helping us conclude that the virtual devices out-perform the physical devices.

[18] *Applied Nonparametric Statistical Methods*, Ch: 2.3 The Sign Test, P. Sprent and N.C. Smeeton, 2001, Chapman and Hall/CRC

Next, we test the hypothesis that *the CoreMark of the virtual devices (or the physical devices) is greater than f% of the CoreMark of an ideal device*. The table below shows the largest values of f% at which this hypothesis holds with a p-value of $p \geq 0.95$ for the sign test. Greater the value of f% and closer it is to 100%, the better; cells shaded green denote the better of the virtual or the physical device. We have between $n = 15$–240 data points per VIoLET device, and $n = 201$–3196 data points per physical device.

Device Type	Pi2	Pi3B	Pi3B+	TX1
Physical	99%	89%	86%	99%
VIoLET	91%	94%	95%	92%

Bandwidth. We similarly use the sign test to test the hypothesis that *the bandwidth of the virtual network (or the physical network) is greater than f% of the bandwidth of the ideal network*. The table below shows the largest values of f% at which this hypothesis holds with a p-value of $p \geq 0.95$. Greater the value of f% and closer it is to 100%, the better. Here, we have a limited number of data points of $n = 2$–4 for the physical and the VIoLET networks.

Network Type	P1	P2	P3	Q1
Physical	94%	79%	94%	22%
VIoLET	98%	95%	98%	97%

Latency. We use the sign test to test the hypothesis that *the latency of the virtual network (or the physical network) is lower than (100 + f%) of the latency of the ideal network*. The table below shows the smallest values of f% at which this hypothesis holds. Smaller the value of f% and closer it is to 0%, the better. Here, we only have a few data points, between $n = 8$–16, for the physical and the VIoLET networks.

Network Type	P1	P2	P3	Q1
Physical	24%	45%	22%	7825%
VIoLET	19%	28%	18%	34%

Ping-pong application latency. We test the hypothesis that *the ping-pong latency of the virtual network (or the physical network) for the pub-sub application is lower than (100 + f%) of the latency of the ideal network*. The table below shows the smallest values of f% at which this hypothesis holds, using the sign test. Smaller the value of f% and closer it is to 0%, the better. Here, we have between $n = 360$–720 data points for the physical and the VIoLET applications.

Network Type	P1	P2	P3	Q1
Physical	98%	185%	96%	96%
VIoLET	42%	57%	42%	45%

E Monetary Costs for the Different Deployment Setups

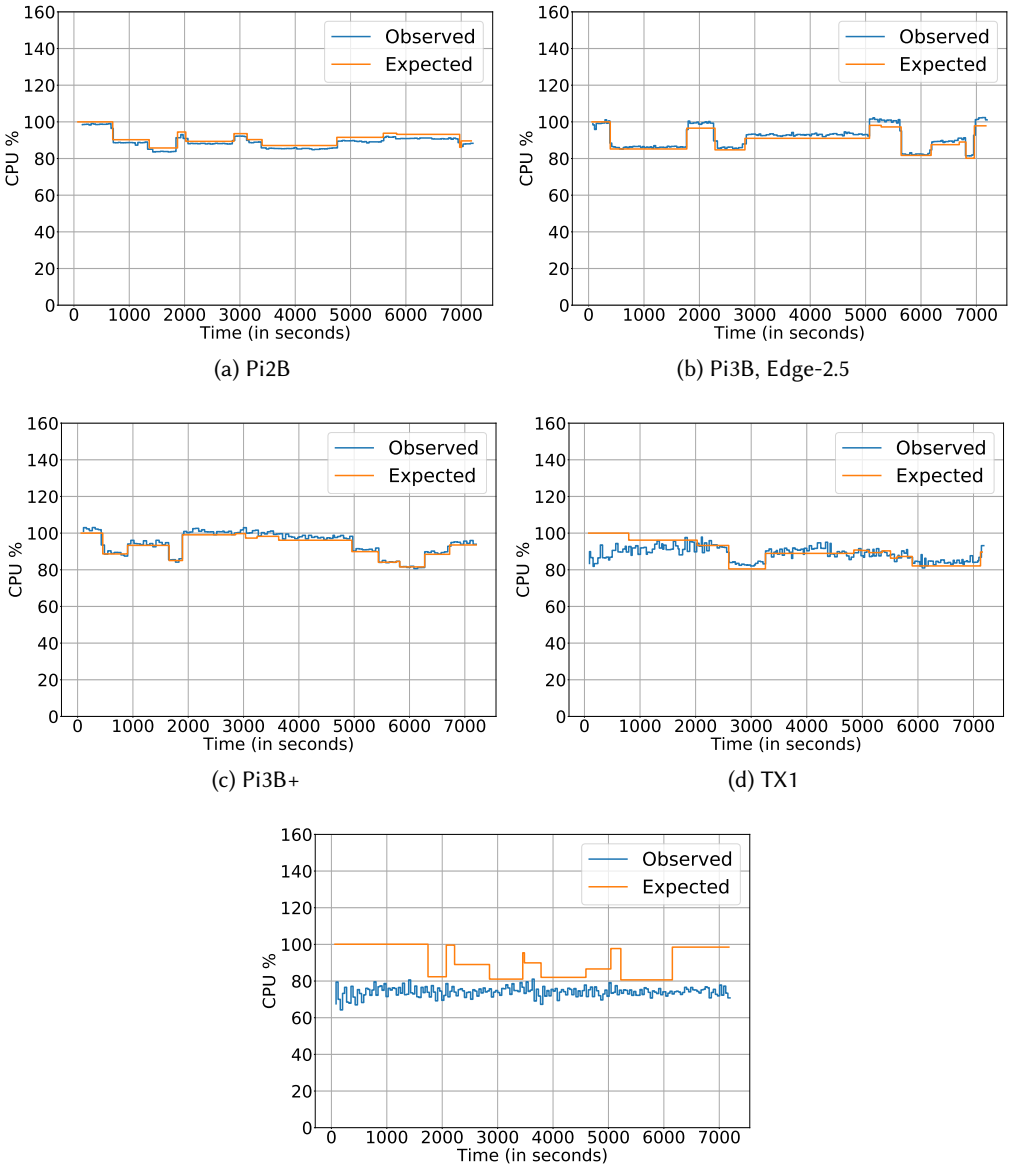
Table 7 shows the hourly VM rental costs for deploying the VIoLET containers and the expected costs for purchasing the devices for a corresponding physical deployment for the different setups. This draws upon details from Tables 1 and 2 in the main article. We also show the number of hours/days for which the VE has to be run for its cost to match the physical deployment cost.

As we can see, the capital costs for physical devices are high, costing over $1000 for just 25 devices, and over $66,000 for 1000 or more devices. There may be additional management and operational costs for maintaining the physical testbed that is not included here. In contrast, VIoLET is billed per hour (or even per minute) of its usage on the VMs. These cost only $2.52/*hour* for a 25-device setup, and $84/*hour* even with 1000 devices. In other words, one can run the VE for about 30 *days* continuously before its operational costs match the capital costs of a physical setup. Typically, the VE would be used for 10s of hours during testing, debugging and evaluation, and instantiated on demand for what-if and fault analysis.

Table 7. Devices, their mapped VMs and the actual cost, for four different deployment setups

Deployment Setup→	**D25**		**D100**		**D400**		**D1000**	
Physical Device/VM Host	Count	∑Cost ($)	Count	∑Cost ($)	Count	∑Cost ($)	Count	∑Cost ($)
Pi 2B	2	100	51	2,550	255	12,750	542	27,100
Pi 3B	6	300	36	1,800	96	4,800	292	14,600
Pi 3B+	16	800	11	550	47	2,350	151	7,550
TX1	1	499	1	499	1	499	10	4,990
Softiron	-	-	1	2,400	1	2,400	5	12,000
Total	**25**	**$1,699**	**100**	**$7,799**	**400**	**$22,799**	**1,000**	**$66,240**
D16 VM	3	2.52/hr	-	-	-	-	-	-
D32 VM	-	-	5	8.40/hr	20	33.60/hr	-	-
D64 VM	-	-	-	-	-	-	25	84/hr
VE Runtime for Phys. Cost	**674 hrs** (28 *days*)		**928 hrs** (37 *days*)		**679 hrs** (28 *days*)		**789 hrs** (33 *days*)	

F Observed vs. Expected CPU Performance with Dynamism Enabled



(a) Pi2B

(b) Pi3B, Edge-2.5

(c) Pi3B+

(d) TX1

(e) SI

Fig. 15. Observed CPU utilization% and expected CPU% performance for sample devices of four types, on D100

G Reliability Timeline for D25

Fig. 16 shows the reliability timeline for all 25 devices of the D25 VE for a 2 *hour* experiment with unreliability enabled, with a configuration of MTTF as $\mu = 600$ *secs*, MTTR as $\lambda = 300$ *secs* and control interval as $\epsilon = 30$ *secs*. An upward line shows the device is working and a downward line indicates that it has failed. The right Y axis indicates the average observed MTTF and MTTR for each device.

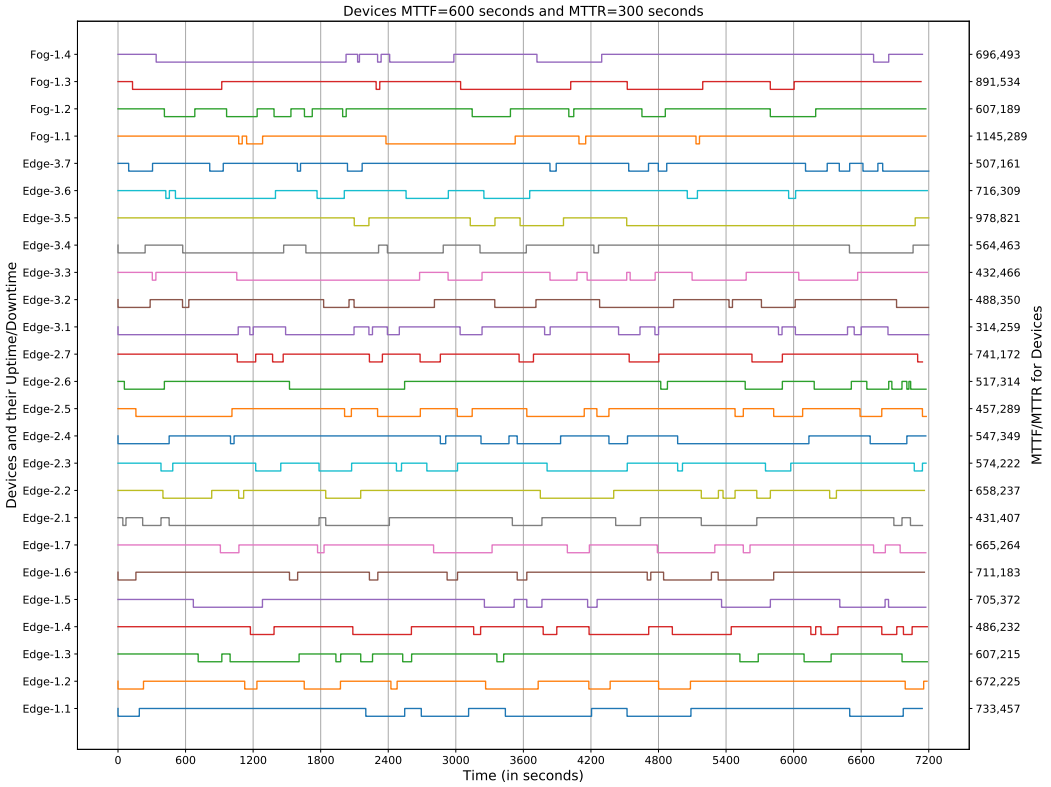


Fig. 16. MTTF/MTTR timeline plot for D25 deployment