A Low-code Development Framework for Cloud-native Edge Systems

WENZHAO ZHANG, YUXUAN ZHANG, and HONGCHANG FAN, College of Computer Science, Zhejiang University
YI GAO and WEI DONG, College of Computer Science, Zhejiang University and Alibaba-Zhejiang University Joint Institute of Frontier Technologies

Customizing and deploying an edge system are time-consuming and complex tasks because of hardware heterogeneity, third-party software compatibility, diverse performance requirements, and so on. In this article, we present TinyEdge, a holistic framework for the low-code development of edge systems. The key idea of TinyEdge is to use a top-down approach for designing edge systems. Developers select and configure TinyEdge modules to specify their interaction logic without dealing with the specific hardware or software. Taking the configuration as input, TinyEdge automatically generates the deployment package and estimates the performance with sufficient profiling. TinyEdge provides a unified development toolkit to specify module dependencies, functionalities, interactions, and configurations. We implement TinyEdge and evaluate its performance using real-world edge systems. Results show that: (1) TinyEdge achieves rapid customization of edge systems, reducing 44.15% of development time and 67.79% of lines of code on average compared with the state-of-the-art edge computing platforms; (2) TinyEdge builds compact modules and optimizes the latent circular dependency detection and message routing efficiency; (3) TinyEdge performance estimation has low absolute errors in various settings.

CCS Concepts: • **Networks** → **Cloud computing**; • **Software and its engineering** → **Development frameworks and environments;**

Additional Key Words and Phrases: Edge computing, low-code development, cloud-native

ACM Reference format:
Wenzhao Zhang, Yuxuan Zhang, Hongchang Fan, Yi Gao, and Wei Dong. 2023. A Low-code Development Framework for Cloud-native Edge Systems. *ACM Trans. Internet Technol.* 23, 1, Article 15 (February 2023), 22 pages.
https://doi.org/10.1145/3563215

This work is supported by the National Key R&D Program of China under Grant No. 2019YFB1600700, the National Science Foundation of China under Grant No. 62072396 and 61872437, and Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under Grant No. LR19F020001, and the Fundamental Research Funds for the Central Universities (No. 226-2022-00087).
Authors' addresses: W. Zhang, Y. Zhang, and H. Fan, College of Computer Science, Zhejiang University, 38 Zheda Rd, Hangzhou, China; emails: {wz.zhang, yx.zhang, fanhc}@zju.edu.cn; Y. Gao and W. Dong (corresponding author), College of Computer Science, Zhejiang University and Alibaba-Zhejiang University Joint Institute of Frontier Technologies, 38 Zheda Rd, Hangzhou, China; emails: {gaoy, dongw}@zju.edu.cn.

1533-5399/2023/02-ART15 $15.00
https://doi.org/10.1145/3563215

1 INTRODUCTION

Recently, edge computing systems emerge as a promising approach to achieving low-latency computing and better privacy protection. Edge computing can be applied in a wide range of applications including video surveillance [77], autonomous vehicles [46], AR/VR [45], and so on.

Recent edge computing platforms adopt cloud-native techniques such as virtual machines and containers to hold multiple services and provide isolation. There exist several edge platforms both in academia and industry. For example, ParaDrop [47] is a specific edge computing platform that provides computing and storage resources at wireless APs. EdgeX [61] is an open-source project whose primary purpose is to build an interoperable platform to enable an ecosystem of plug-and-play components for industrial IoT applications.

While these platforms have already shown their success in a number of applications, we observe the existing approaches are still insufficient in solving the following problems:

(1) *Low-code customization of edge systems.* Different from edge applications, an edge system can be seen as the aggregation of modules. The more modules an edge system has, the more applications it can support. However, hosting a rich set of modules is usually not feasible, as edge computing devices (e.g., wireless APs) have limited hardware resources compared with cloud platforms. It is essential that only application-required modules can be quickly deployed on the edge devices. Moreover, it is also vital to allow users to easily specify the interactions among these modules.

(2) *Accurate performance estimation.* Edge devices are in close proximity to IoT devices. As such, the resource consumption and performance of the entire system depends on the specific deployment strategy, e.g., the types of hardware the edge system is deployed upon, and the types of communication protocols connecting IoT and edge devices. It is important that we can estimate the resource consumption and performance of the entire system and give guidance to developers on how to deploy the customized systems on edge devices.

To address the above issues, we present TinyEdge, a holistic framework for rapid customization of edge systems and IoT applications. TinyEdge inherits many designs from existing works of literature: (1) Container-based system architecture to achieve good extensibility at a low cost. (2) Cloud-based backend through which application required edge services can be flexibly downloaded and customized to the edge devices. (3) Integration of popular modules like device connector, time-series database, edge intelligent data analysis services, visualization, and so on.

However, we have several unique considerations.

First, existing container-based modules can not provide enough flexibility in terms of customization. They rely on RESTful API [61] or serverless function [56] to call different functionalities or build interactions, which requires a certain level of expertise and necessary configuration information; and their configuration format is not so friendly to the novice, either. TinyEdge abstracts different module configurations, enables cross-module configuration sharing to reduce the configuration overhead, and provides a unified domain-specific language to reduce the effort of application coding.

Second, the current cloud-based backend falls short in multi-dimensional consideration of performance modeling. State-of-the-art industrial edge platforms like Azure IoT Edge [56] only provide a coarse-grained cost model for each module without considering the performance metrics like workload or latency of an edge system. TinyEdge builds a multi-dimensional performance model by considering unique features of different modules through sufficient profiling, and selects from specific machine learning algorithms to obtain higher accuracy.

We evaluate TinyEdge using real-world edge systems. Results show that: (1) TinyEdge achieves rapid customization of edge systems, reducing 44.15% of customization time and 67.79% lines of code on average compared with that of state-of-the-art edge platforms; (2) TinyEdge builds a precise and practical performance model, considers multi-dimensional information for different modules.

The contributions of this work are summarized as follows:

— We present TinyEdge, a holistic framework for the low-code development of edge systems. TinyEdge offers various customizable and compact modules together with a **Domain Specific Language** (**DSL**). We have shown in our article that TinyEdge is beneficial to both non-expert and expert developers.
— We propose a general methodology to build accurate workload and latency models for the edge system. This model can be useful for supporting high-level system management including auto-scaling and application scheduling.
— We implement TinyEdge and evaluate it using three concrete and representative edge systems. Results show that TinyEdge achieves low code customization and generates accurate performance estimation results in terms of various metrics.

The rest of this article is organized as follows: Section 2 presents related works. Section 3 illustrates the system overview of TinyEdge, including TinyEdge usage and design overview. Sections 4 and 5 explicate the TinyEdge customization and performance estimation service respectively. Section 6 presents the implementation details of the TinyEdge system. Section 7 shows the evaluation results. Section 8 concludes this work.

2 RELATED WORK

2.1 IoT, Edge Computing, and 5G/6G

In recent years, IoT technologies have been widely exploited more than ever in many areas, e.g., industry [35, 60], 5G [13, 43], and 6G [38, 50, 51]. Such a rapid development of IoT and mobile Internet leads to the explosive growth of data at the extreme edge of the network. As a result, cloud computing can no longer satisfy the need for real-time, secure, and low-energy cost analysis of big data. Under this scenario, edge computing has emerged as a new paradigm that executes computation at the edge of the network. Different from cloud computing, edge computing can provide computation resources close to the data sources, which equips with the characteristics of low latency and high privacy.

Moreover, **mobile edge computing** (**MEC**) is a key technology for new generations of mobile communication such as 5G [28, 30, 66, 68] and 6G [1, 6, 34, 48]. With the advances in telecommunication infrastructures, functions, and applications, we can define more advanced air interfaces in 5G/6G networks [30]. The emergence of edge computing helps advance the transformation of the mobile broadband network into a programmable world and contributes to satisfying the demanding requirements of 5G/6G networks in terms of expected latency, scalability, and automation [30]. Another key technology to enable 5G/6G-based application development is the virtualized radio access network, i.e., vRAN. There is already a bunch of research on this topic to ensure real-time vRAN tasks [20], provide reliability [23], strike the tradeoff between performance and energy consumption [3], and conduct other optimizations [7, 16, 37, 74]. With the vRAN technology, developers can build IoT applications that are equipped with in-depth communication and resource optimizations.

TinyEdge provides general modules and the DSL for the developers to easily build typical IoT applications with functionalities including device connection, edge processing, and edge-cloud connection. Although TinyEdge does not provide modules for applications scenarios such as MEC and 5G/6G vRAN, developers can still benefit from our system framework by following the TinyEdge schema and the corresponding building process. With the necessary modules, TinyEdge can further provide performance estimation services for optimizing the overall system performance.

2.2 Edge Computing Platforms and Middlewares

Due to the prominence of edge computing, a number of edge computing platforms with different purposes have emerged, targeting stream analysis [2, 63], data sharing [78], security [5], and most importantly, deploying systems by integrating cloud-edge-end resources [47, 56, 61, 64]. In this section, we focus on the last category. The comparison between our work and the existing works will also be discussed.

Cloudlet [64] is devised to instantiate customized service software on edge devices rapidly. Cloudlet uses VM overlay as a building block of edge computing system, which results in low extensibility as the OS-dependent nature, users need to create different overlays for the same system on alternative OSes. Cloudlet also requires OpenStack++ that runs on the Cloudlet Node to carry out baseVM importation, resumption, and overlay loading.

ParaDrop [47] is an edge computing platform that provides modest computing and storage resources at the "extreme" edge of the network, whose main purpose is to allow third-party developers to flexibly create new types of services. ParaDrop consists of three main components—a hosting substrate that supports multi-tenancy, a cloud-based backend that orchestrates computations across many ParaDrop APs, and an API that third-party developers can deploy and manage their own computing functions across different ParaDrop APs. However, ParaDrop does not provide resource and performance models, and its instance tool is supported on limited hardware or VM.

EdgeX [61] is an open-source project supported by Linux Foundry. Its main purpose is to build an interoperable platform to enable an ecosystem of plug-and-play components that unifies the marketplace and accelerates the deployment of IoT solutions across a wide variety of industrial and enterprise use cases. The main drawbacks of EdgeX are high redundancy and relatively low capability. When customizing new systems, EdgeX requires deploying all its core modules; EdgeX does not have data analysis services, and a lot of manual work is needed to integrate new ones.

Azure IoT Edge [56] is an industrial edge computing platform. There are also other competitors like AWS Greengrass [59], Alibaba LinkEdge [9], and Baidu IntelliEdge [4], but they are either not able to integrate third-party modules or are still in a preliminary stage. As an industrial platform with a powerful cloud backbone, Azure IoT Edge can basically satisfy all we need to deploy an edge system, but the high integration with the cloud makes it hard to use, users have to get familiar with a wide range of cloud services before getting hands-on Azure IoT Edge itself.

KubeEdge [10] is another open-source project supported by **Cloud Native Computing Foundation** (**CNCF**). KubeEdge aims at extending native containerized application orchestration capabilities to hosts at the edge. KubeEdge can be regarded as a mixture of EdgeX and Azure IoT Edge, which embraces the heterogeneity of IoT devices and leverages cloud resources. However, the configuration and application development process of KubeEdge is not so friendly to conventional users.

K3s [11] is a lightweight certified Kubernetes distribution built for IoT and edge computing, which is currently a CNCF sandbox project. Different from KubeEdge, K3s is an orchestration framework rather than a development platform. To develop an IoT application, users will have to build from the ground up using a container engine such as Docker or container.

WasmEdge [12] is a lightweight, high-performance, and extensible WebAssembly runtime for cloud native, edge, and decentralized applications. Users can write a WasmEdge app in various programming languages such as Rust, JavaScript, Go, and Python. WasmEdge will provide a safe, secure, isolated, and containerized runtime to execute the written applications.

Unlike Cloudlet [64] and ParaDrop [47], TinyEdge leverages the container-based design to achieve high extensibility. Compared with EdgeX [61], Azure IoT Edge [56], and KubeEdge [10], TinyEdge provides additional key techniques such as configuration sharing and dependency checking, greatly accelerating the customization process. Different from K3s [11] and WasmEdge [12] which require building edge systems and IoT applications from the ground up, TinyEdge provides general modules and the DSL to facilitate the development. Furthermore, we also offer profiling-based performance models that can help to give useful guidance for system deployment and performance optimization.

2.3 Low-code Development Framework

IoT applications usually require developers to have certain-level expertise in both hardware and software. There are also many sub-areas like sensing, smart home, smart factory, and so on, which makes the design considerations of an IoT application more sophisticated. Recent years have witnessed the growth of IoT devices, and researchers in both academia and industry begin paying efforts to low-code development for IoT applications.

Academia works. TinyLink series [15, 24, 26, 41] are state-of-the-art systems for rapidly developing IoT applications, which provide easy-to-use APIs that can help users develop IoT applications (running on IoT devices) without worrying about hardware specifications and complex interactions between end and cloud. Target at sensing applications, Shen et al. propose Beam [65], a framework that can let developers specify "what should be sensed or inferred" by introducing the key abstraction of an inference graph. To simplify development, Beam requires developers to first manually construct inference graphs with prior expert knowledge and then build sensing applications upon the graphs. Target at smart spaces, Fu et al. propose dSpace [22], an open and modular programming framework that aims at simplifying and accelerating smart space application development by introducing two abstractions (i.e., digivice and digidata) and three primitives (i.e., Mount, Pipe, and Yield). dSpace, however, mainly focuses on the rapid development of developing/-composing high-level "virtual entities" like a room or a house, rather than building out-of-the-box edge systems.

Industry solutions. Tuya IoT platform [33] provides interconnectivity standards to support: (1) zero-code development for COTS devices in a drag-and-drop manner and (2) low-code development for MCU-based devices. Home Assistant [32] is an open-source home automation platform that can connect and control a wide range of COTS IoT devices. Defining automation, i.e., policies is the key development method in Home Assistant, which requires little coding expertise and lines of code. Node-RED [21] is a lightweight flow-based programming tool that connects hardware devices, APIs, and online services. Users can easily build an IoT application in a web front-end with Node-RED modules. IFTTT [31] is the leading no-code platform on mobile where users can define rules intuitively. A rule can be triggered by different events (e.g., receive a new e-mail, the light is opened) and take diverse reactions such as sending a message or actuating an IoT device.

TinyEdge focuses on the rapid composition of edge systems (running on edge devices), which is orthogonal to solutions that focus on IoT devices (e.g., TinyLink [15, 24, 41] and Tuya IoT platform [33]). Compared with cloud-device integrated solutions (e.g., TinyLink 2.0 [26], Beam [65], and dSpace [22]), TinyEdge provides essential services (e.g., connector services) for IoT devices that do not have direct Internet access (e.g., via BLE or ZigBee). Compared with general-purpose development frameworks using IFTTT policies or flow-based programming (e.g., Home Assistant [32], IFTTT [31], and Node-RED [21]), TinyEdge's DSL has higher expressiveness in that it can describe complex data- and control-flow besides simple "if this then that" rules.

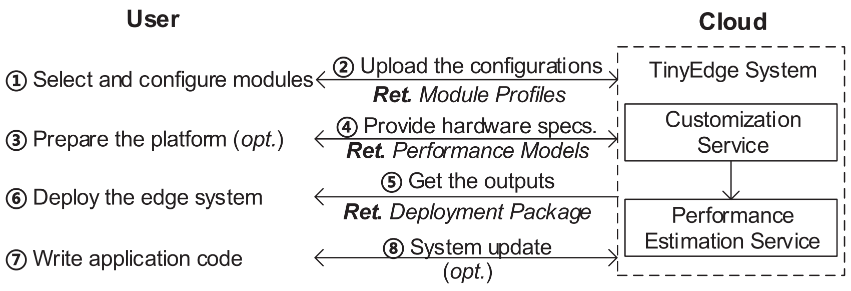


Fig. 1. Workflow overview of TinyEdge.

2.4 Performance Modeling

Performance modeling plays an important role in various systems. It not only provides essential information for system developers about how well a system operates but also gives users awareness of distinct aspects of a system. Although there are a number of performance metrics that are valuable to analysis a system, we focus on the latency and workload in our scenario.

MobiQoR [44] is an optimization framework that minimizes service response time and app energy consumption, giving the offloading strategy for a series of edge nodes. The service response time and app energy consumption are modeled in the white-box manner, considering features like time of data transfer, task processing, and power. But MobiQoR does not take the effect of multi-thread execution and dynamic workload into consideration, to tackle this issue, Guan et al. propose Queec [25]: a QoE-aware edge computing system, where they use regression techniques to model the execution time and workload of specific edge computing applications like speech and face recognition. Maheshwari et al. [53] present a scalable edge cloud system model that incorporates M/M/C queuing model as its computation model and divides the overall latency into transmission delay, routing node delays, and the cloud processing time.

Unlike the above works, TinyEdge proposes a hybrid solution to combine the merits of both black- and white-box methods and integrates the resource constraint into the model, making the estimation results more reliable.

3 TINYEDGE OVERVIEW

In this section, we will give an overview of TinyEdge in terms of system usage (Section 3.1) and design (Section 3.2).

3.1 TinyEdge Usage

In this subsection, we will illustrate how to use TinyEdge. We use a typical IoT system as an example to present the overall process when using TinyEdge. This system can pull sensor data through the MQTT protocol, store the data in InfluxDB and visualize them by Grafana. As shown in Figure 1, a user needs to perform the following eight steps:

- **Step ① and ②**. Select MQTT Connector, InfluxDB, and Grafana then provide necessary configurations ((Figure 4), detailed in Section 4.1). The user will get a snapshot of module profiles ((Figure 6), detailed in Section 5.1) that contain the approximate minimal storage and memory consumption of the customized system.
- **Step ③ and ④**. Based on the resource consumption, the user can choose to prepare satisfactory hardware or virtual platform to run the customized system and upload the hardware specifications to get specific performance models rather than general ones with default values.

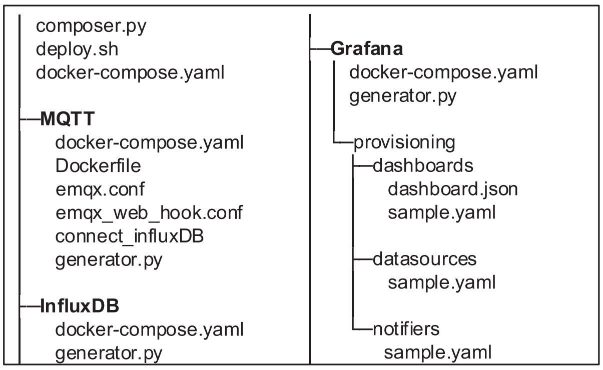


Fig. 2. Deployment package directory of the IoT system.

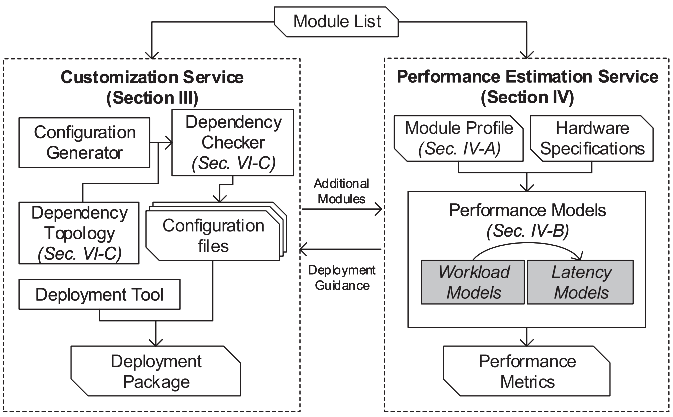

Module List
Customization Service (Section III)
Configuration Generator
Dependency Checker (Sec. VI-C)
Dependency Topology (Sec. VI-C)
Configuration files
Deployment Tool
Deployment Package
Additional Modules
Deployment Guidance
Performance Estimation Service (Section IV)
Module Profile (Sec. IV-A)
Hardware Specifications
Performance Models (Sec. IV-B)
Workload Models
Latency Models
Performance Metrics

Fig. 3. Architecture overview of TinyEdge.

- **Step ⑤ and ⑥**. Then the users will get the deployment package that consists of an OS-specific deployment tool (detailed in Section 6) with other necessary supplementary configuration files for each module. Figure 2 shows the deployment package directory architecture in this case. Necessary configuration files are stored under separate directories for each module. *composer.py* in the root directory is a script that aggregates module configurations; while *deploy.sh* is the deployment tool to set up the environment and the customized edge system.
- **Step ⑦ and ⑧**. After deploying the system, the user writes application code ((Figure 5), detailed in Section 4.2) to specify the interaction logic, which may trigger a system update.

We can see that TinyEdge workflow is clean and short; users can customize and deploy typical edge computing systems quickly. Performance models can provide a rough estimation of the customized system, which help users to choose a satisfactory hardware prototype more easily. Furthermore, TinyEdge deployment tool can automate the process of system setup and save time.

3.2 TinyEdge Design

Figure 3 depicts the overview of TinyEdge, including the customization service (Section 4) and the performance estimation service (Section 5). The module list serves as the input of TinyEdge and it is passed to both services:

— Inside the customization service, the configuration generator automatically generates the low-level configurations by considering the typical configuration format of Dockerfiles and Docker-compose files for different modules, as well as the dependency between them. The deployment package for the target customized system will then be generated.
— Inside the performance estimation service, the performance models takes module profiles as inputs, which include the workload and latency model of different modules. The performance metrics under different hardware specifications for the target customized system will then be generated.

Note that the container-based system architecture makes it possible to integrate other important edge services. For example, we can easily add off-the-shelf secret store, API gateway and user-role access control service to enhance system security. Furthermore, the incorporation of Kubernetes enables multi-tenant coordination among heterogeneous edge devices and better utilization of extended computing resources like GPU.[1] The above features make TinyEdge a more flexible and reliable system.

4 CUSTOMIZATION SERVICE

The main purpose of TinyEdge customization service is to provide a software and hardware agnostic, easy-to-use interface that users can easily ensemble a scenario-specific edge system with much less effort. In this section, we will explicate this service with respect to system-level customization (Section 4.1), and application-level customization (Section 4.2).

4.1 System-level Customization

In the system-level customization phase, a user needs to select modules and provide the necessary configurations for them. To facilitate system-level rapid customization, TinyEdge proposes configuration reduction methods to reduce the system configuration time.

Configuration classification. For container-based modules, existing edge computing platforms not only require users to learn module-specific configurations that have many items but also to master the complex format of container-specific configurations. To enable rapid configuration, we pack up module configurations into (1) basic configurations that a module needs to operate normally and (2) advanced ones that deal with more complex situations or contain performance requirements with default values. To handle sophisticated configuration formats, we maintain a configuration mapper between TinyEdge and Dockerfile/Kubernetes. As a result, users only need to provide basic configurations to run the customized system and fill in the advanced ones only when necessary with much less effort without knowing the detailed configuration format of Dockerfile/Kubernetes.

We revisit the example shown in Section 3.1. Before applying TinyEdge configuration classification technique, a novice user will first need to learn the configuration format of both InfluxDB and Docker/Kubernetes, then provide the information according to his own requirement from all configuration items listed on the left side of Figure 4, and repeat this procedure for all modules before he can finally deploy and run the customized system. While after applying the technique, the user only needs to follow the instruction of each TinyEdge module with a unified configuration form.

Global configuration sharing. In modular settings, sometimes one module will share parts of its configurations with another, especially for database modules. While existing edge computing platforms treat each Docker-based module as a standalone daemon, they either use RESTful API

[1] https://kubernetes.io/zh/docs/concepts/configuration/manage-resources-containers/#extended-resources.

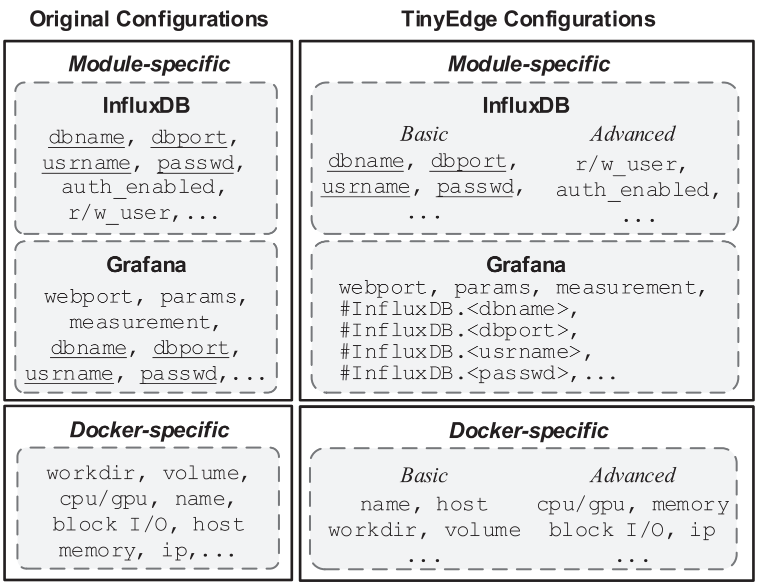

Original Configurations
TinyEdge Configurations
Module-specific
InfluxDB
dbname, dbport, usrname, passwd, auth_enabled, r/w_user,...
Grafana
webport, params, measurement, dbname, dbport, usrname, passwd,...
Module-specific
InfluxDB
Basic
dbname, dbport, usrname, passwd, ...
Advanced
r/w_user, auth_enabled, ...
Grafana
webport, params, measurement, #InfluxDB.<dbname>, #InfluxDB.<dbport>, #InfluxDB.<usrname>, #InfluxDB.<passwd>,...
Docker-specific
workdir, volume, cpu/gpu, name, block I/O, host memory, ip,...
Docker-specific
Basic
name, host workdir, volume ...
Advanced
cpu/gpu, memory block I/O, ip ...

Fig. 4. An example of configuration classification and sharing.

or serverless function to exchange parameters at runtime or configure it multiple times at startup. The problem is twofold: (1) the shared configurations need to be set multiple times; (2) if module A changes its shared configurations while B is not, B may fail to operate normally. TinyEdge uses a placeholder in the form of "#<module name>.<configuration name>" to replace the shared configurations so that the user can only fill in the shared configuration once and TinyEdge configuration generator will automatically handle the others. Moreover, the shared configuration can also be overloaded by using the "@overload" annotation.

We revisit the example shown in Section 3.1. If the user wants to use Grafana to visualize the time-series data stored in InfluxDB, he will need to provide `host`, `dbname`, `dbport`, `usrname`, and `passwd` for both InfluxDB and Grafana. As Figure 4 shows, Grafana needs the configuration of InfluxDB, and the user only needs to provide the shared configurations once in InfluxDB, those in Grafana will be generated by TinyEdge when used at runtime.

4.2 Application-level Customization

In the application-level customization phase, a user first writes application code using TinyEdge DSL. TinyEdge runtime will parse the code into different parts and generate message queue topics, then distribute each part to designated execution modules or engines.

TinyEdge DSL. Existing edge platforms require users to learn how to use each module and program with them, which usually needs to co-operate among different programming languages. To facilitate application-level rapid customization, TinyEdge proposes a DSL that abstracts TinyEdge modules' functionalities to provide a general coding form and reduce programming time. TinyEdge DSL can be divided into the following three parts:

① **Module function call.** TinyEdge creates a virtual class to wrap all functions that belong to a module. Like *get_data()* of MQTT_Connector and *visualize()* of Grafana in Figure 5.

② **Message routing.** Within this virtual class, there are two virtual functions that each module has to implement, which are *Pub()* and *Sub()*. The *Pub()* function takes charge of wrapping

```
from tinyedge.modules import mqtt, influxdb, grafana
from tinyedge.utils import Serverless
MQTT_Connector = mqtt("emq")
data = MQTT_Connector.get_data("id")
topic_id_1 = MQTT_Connector.Pub(data)
func = Serverless(lang="python3.6", pkgs=req_path, code_path)
func.Sub(topic_id_1)
data = func.get_results(**args)
topic_id_2 = func.Pub(data)
Influx = influxdb("influxdb")
data = Influx.Sub(topic_id_2)
InfluxDB.insert(data)
Grafana = grafana("grafana")
data = Grafana.Sub(topic_id_1)
Grafana.visualize(data)
```

Fig. 5. A python-like application code snippet of the IoT system.

data and creating topics in the TinyEdge message queue engine. While the *Sub()* function is responsible for subscribing topics from the message queue engine and unwrapping data.

③ **Serverless function.** Serverless is a class that used to define the programming language, version, and requirements. After filling in the serverless code template, function *get_results()* will pass the language configuration and application code to TinyEdge serverless engine, execute the serverless function and get the results.

The different alternatives of the serverless function are generally referred to as RESTful API. Although RESTful API shares the merit of ease-of-programming, the serverless function still shows better potential as it (1) has comprehensive infrastructures both in the industrial and open-source community; (2) thoroughly decouples functional and service code modules; (3) has a little requirement of module backend, which makes it easier to implement; (4) do not need to operate all the time.

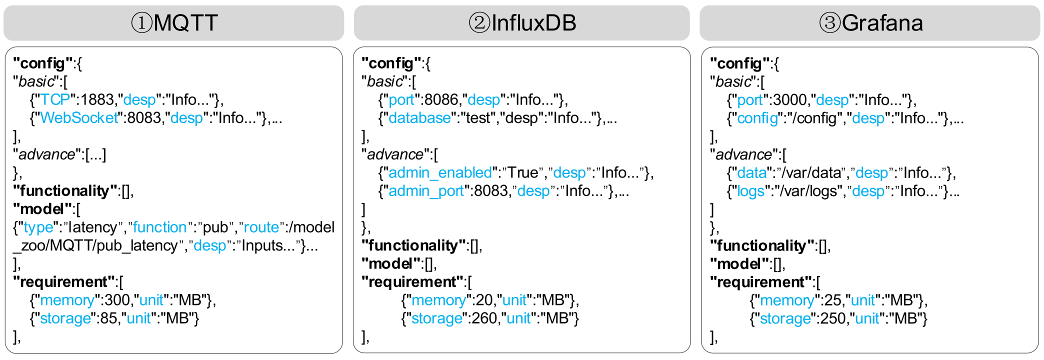
5 PERFORMANCE ESTIMATION SERVICE

After building an edge computing system, the developer may wonder what's the performance of this customized system. Given the performance estimation, developers are able to come up with intelligent scheduling policies like workload offloading, auto-scaling, and access control. Considering device and communication protocol heterogeneity, performance estimation of edge computing systems is a hard nut to crack. Existing industrial edge computing platforms only gives cost models for their edge services, which shed little lights on how to build or select hardware specifications and connection methodologies that can yield acceptable performance.

TinyEdge performance estimation service aims at giving users awareness of the resource consumption or key metrics like the latency, workload of the customized system. Towards this, TinyEdge includes module profiles (Section 5.1) and builds multi-dimensional models to describe key metrics of the customized system (Section 5.2).

5.1 Module Profile

TinyEdge module profile is the key information source for the performance models. A module profile contains category, customization information (functionality and configuration), resources requirement (memory, storage, and connection), and performance models for specific functions.


①MQTT
②InfluxDB
③Grafana

```
"config":{
"basic":[
   {"TCP":1883,"desp":"Info..."},
   {"WebSocket":8083,"desp":"Info..."},...
],
"advance":[...]
},
"functionality":[],
"model":[
{"type":"latency","function":"pub","route":/model
_zoo/MQTT/pub_latency","desp":"Inputs..."}...
],
"requirement":[
   {"memory":300,"unit":"MB"},
   {"storage":85,"unit":"MB"}
],
```

```
"config":{
"basic":[
   {"port":8086,"desp":"Info..."},
   {"database":"test","desp":"Info..."},...
],
"advance":[
   {"admin_enabled":"True","desp":"Info..."},
   {"admin_port":8083,"desp":"Info..."},...
]
},
"functionality":[],
"model":[],
"requirement":[
      {"memory":20,"unit":"MB"},
      {"storage":260,"unit":"MB"}
],
```

```
"config":{
"basic":[
   {"port":3000,"desp":"Info..."},
   {"config":"/config","desp":"Info..."},...
],
"advance":[
   {"data":"/var/data","desp":"Info..."},
   {"logs":"/var/logs","desp":"Info..."}...
]
},
"functionality":[],
"model":[],
"requirement":[
      {"memory":25,"unit":"MB"},
      {"storage":250,"unit":"MB"}
],
```

Fig. 6. An example of three module profiles for the IoT system, where **config** stores basic and advance configurations; **functionality** stores the choice of module functionality; **model** stores the route to the module performance model; **requirement** stores resources (including connection) requirement.

TinyEdge splits modules into three categories in accordance with their functionalities: (1) System modules that make up the essential part of an edge system, like device management, logging, authentication, database, and so on. (2) Processing modules that take charge of the computation part of an edge system, like data filter, video analyzing, object recognition, and so on. (3) Connecting modules that are responsible for accessing IoT devices or transmitting data to the cloud, like HTTP, MQTT, Bluetooth connector, and so on.

Customization information is extracted at the system-level customization stage. It records functionality and configuration of each user-selected module, serving as an input of TinyEdge configuration generator that maps high-level configuration into specific formats like Dockerfile or Kubernetes.

Unlike customization information, resources requirement and metrics models are given offline. We sample the average resource consumption and build metrics models for each module at different hardware specifications. While connection describes the hardware requirement that a module needs to communicate with others. For example, a Bluetooth connector requires an underlying hardware chipset to operate.

5.2 Performance Models

There are plenty of performance metrics that can describe different facets of a system like latency, workload, accuracy, energy, and so on. TinyEdge chooses to model latency and workload for the following two reasons: (1) they are both general performance metrics that can be inferred from a wide range of modules; (2) they can describe two essential edge computing features, which are low latency and device heterogeneity. Although accuracy is a critical metric, especially for machine learning modules, the results tend to have a little variance to the environment. It is usually module-specific factors that have the most prominent influence. As a result, it is hard to build a general model for accuracy. As for energy, TinyEdge focuses on edge nodes powerful enough to run containers, which are usually equipped with a power supply.

Existing solutions like [25] and [53] have not considered the number of access devices, which could potentially affect both latency and workload on the edge, especially in multi-device and multi-protocol scenarios; on the other hand, over-simplifying the problem causes inferior results.

Workload model. The main functionality of the workload model is to give a general form of workload under different types of hardware platforms (in terms of system architectures or CPU models) for distinct IoT applications. Due to the heterogeneity, TinyEdge uses black-box methods to build the workload model.

TinyEdge takes the workload vector and hardware features as input and uses the load average that represents a fraction of CPU consumption as the model output for conformity. The workload vector is defined by the module developer (like frame sampling rate and resolution for image processing modules, the number of accessing end devices, and transmission data size for connecting modules);

While the hardware features mainly include hardware architectures (x86, ARM, and so on.) and the CPU specifications (the number of cores and threads, main frequency, maximum frequency, and so on), which will serve as inputs of a compensation function that map hardware features to the workload variation that is compared with the average workload getting under baseline hardware resources. Then we have

$$\omega_i = g_i(w_i) + c(H),$$

where ω_i represents the workload vector of module i, g_i is the workload model for module i that built upon some kinds of machine learning methods, similarly, $c(H)$ is the compensation model that takes different hardware specifications H as input and outputs the compensated value. Given the workload models, we can get a general form of workload W for application $\mathcal{A}$: $W = \sum_{i=1}^{i \in \mathcal{A}} \omega_i$.

Latency model. The main functionality of the latency model is to predict the end-to-end latency (from the time that a request is generated to the time that it is finished) of an IoT application. Different from workload, latency models can sometimes be well described in a more formal mathematical way, so we use a hybrid black- and white-box methods to build the latency model. More specifically, we use black-box methods to estimate the execution time while leverage white-box methods to characterize the waiting time.

The edge device is running an OS where requests of different modules are coming from time to time. Therefore, as the number of requests in the system increase, workload of the OS will gradually rise to the maximum level. Finally, some of the requests will have to queue till they can be processed. This scenario can be well analyzed by the queuing theory. Unlike previous works [52, 53] that using $M/M/1$ or $M/M/C$ queuing model, TinyEdge leverages a more realistic model which encode existing workload by using $M^{\alpha}/M^{\beta}/C$ queuing model [40].

Traditional $M^{\alpha}/M^{\beta}/C$ means the α requests arrive at a rate of λ, the executors can deal with β requests at a rate of μ, both arrival and execution time follow exponential distribution; there are C executors in the system. In order to integrate workload into queuing model, TinyEdge treats α as the workload requirement (in terms of load average) of module i, regards β as the available workload in the system. With the definition above, we can get the waiting time of module i in the system, i.e., t_i^{wait} as follows:

$$t_i^{wait}(w_i, \mu, \lambda) = f_i^{\mathbb{P}}(g_i(w_i)) + \frac{\rho^{W_{max}}}{\mu \cdot W_{max}! \cdot (1-\rho)^2} P_0,$$

where f_i is the execution latency model for module i that built upon some kinds of machine learning methods, W_{unit}, W_{max} is the unit and maximum level of workload, respectively. P_0 (the probability of no request in the system), and ρ (system intensity) are defined as follows:

$$P_0(\rho) = \left[\sum_{n=0}^{W_{max}-W_{unit}} \frac{(W_{max}\rho)^n}{n!} + \frac{(W_{max})^{W_{max}}}{W_{max}!} \left(\frac{\rho^{W_{max}}}{1-\rho} \right) \right]^{-1}.$$

$$\rho(\lambda, \mu, \alpha, \beta) = \frac{\alpha\lambda}{W_{max}\beta\mu}.$$

Given the queuing model, we build two different types of latency models for processing and connecting modules separately, considering the unique features of these two types of modules. We omit the latency model of system modules because they usually work as the coordinator between the above two types of modules, which tend to take up more time.

(1) *Latency model of processing modules.* For processing modules, the latency mainly consists of the execution time (estimated by a black-box method given the workload) plus the waiting time (estimated by the queuing model). Note that we have to assume the hardware resources like memory or storage are sufficient for a module in terms of building a performance model, otherwise, the model will be meaningless. We have:

$$\iota_i = f_i^{\mathbb{P}}(g_i(w_i)) + t_i^{wait}(w_i, \lambda, \mu).$$

(2) *Latency model of connecting modules.* For connecting modules, the latency is mainly composed by data transmission time, RTT, execution time, and the waiting time. We have:

$$\iota_i = \frac{D_i + D_o}{B} + RTT + f_i^{\mathbb{C}}(g_i(w_i)) + t_i^{wait}(w_i, \lambda, \mu),$$

where D_i, D_o represents the size of input and output data, B is the current bandwidth.

After building different latency models for processing and connecting modules, we can get a general form of latency for an IoT application $\mathcal{A}$: $L_i = \sum_{i=1}^{i \in \mathcal{A}} \iota_i$.

Function $f()$, $g()$, or $c()$ for latency, workload and compensation model is decided by a function selector. We implement several machine learning algorithms (like linear regression, SVM, and random forest) and will carry out a thorough test for models of each module under different hardware resources to select the best one to store.

Note that our current design assumes that a request is executed sequentially. There might be cases that it is run in a multi-threaded way. As such, we can first construct a request execution graph that aggregates modules run in parallel into one "stage". Then we use the latency model for each module to obtain the execution time of a stage by calculating the maximum value within that stage. Finally, the resultant overall execution time can be expressed as follows:

$$\sum_{i=1}^{|Stage|} \max_{1 \le j \le |Stage_i|} |latency_j|,$$

where j represents a specific module in $Stage_i$.

6 IMPLEMENTATION

In this section, we present some details of TinyEdge about how modules are selected, deployed, and interacted with each other on the edge device.

6.1 Module Selection

In order to decide which modules to include, we conduct an investigation of more than twenty edge-computing-related articles [5, 8, 14, 25, 27, 29, 42, 44–47, 54, 55, 63, 69–73, 76–78, 80] and find that the main functionalities of edge systems for IoT applications include: (1) connector for both IoT devices and cloud services, (2) data processing (e.g., stream analytics and machine learning), (3) data storage (e.g., SQLite, Redis, and InfluxDB), (4) security (e.g., authentication, encryption, and access control).

We implement the HTTP, Modbus, and Bluetooth connector, integrate EMQ [17], a popular open-source MQTT broker as our MQTT connector for (1); develop a simple data filter module and an object recognition module based on ResNet for (2); integrate MySQL for (3); implement a device authentication module for (4).

Moreover, edge systems are born to embrace a variety of heterogeneous IoT devices, the generating data is usually time serial. But most of the industrial edge platforms provide neither the device management module nor the timeseries database at the edge. So we implement a lightweight device management module and incorporate InfluxDB (timeseries database), MongoDB (semi-relational database), and Grafana (a visualization tool for timeseries data) to compensate for the issue.

Then we build a container registry, integrating all modules above. Note that users can select the module(s) both in the TinyEdge container registry and the Docker Hub. However, TinyEdge temporarily does not provide the customization option for modules from the Docker Hub; the recommendation service will not take those into account, either.

6.2 Optimization Techniques

Rapid circular dependency checking. As we know, exhaustively checking package dependency can be a tedious and time-consuming work. One strawman method is using a hash set that store dependent module, which exchange storage for speed. Another method is the fast-slow pointer algorithm, which sacrifice speed to save storage. In our scenario, edge system update rapidly so we require faster detection. Towards this, TinyEdge proposes a hybrid hash based fast-slow pointer method that caches dependent modules in the hash set at system set-up stage, during which the fast-slow pointer is used to detect circular dependency. While at system update stage, the hash set method is used when the number of update or addition modules is below a certain threshold.

Topic reduction in message queue. Always generate topics as the user defines may waste the system resources. For example, within the same application, multiple modules subscribe from different topics that are published by the same module, and we have to send the same data twice to different topics. TinyEdge proposes a rule-based topic generation technique to dynamically reduce the number of topics. Specifically: (1) Within the same application, only define one topic for multiple modules subscribe from the same publisher module. (2) Across different applications in the same system, dynamically merge topics that have the exact same subscriber module. As a result, the number of topics will be reduced and TinyEdge can leverage existing load balancing techniques to balance the traffic load of a topic.

Module compaction. Recall that each TinyEdge module is wrapped in a single Docker image. In order to reduce module pulling time, TinyEdge adopts a series of techniques to produce a much smaller modules with little impact on the functionalities. To keep a slim module: (1) TinyEdge borrows the best practice in the Docker community to use smaller base image and optimize the writing style of Dockerfile. (2) TinyEdge flattens a built image with tools like Docker-squash [36] or Compact [75] to run clean up commands in the container, squash multiple image layers into one, and finally generate a compact Docker image.

Optimized runtime. An edge computing platform runtime is responsible to interact with the cloud, enabling deployment and module communication within the edge, and so on. Current edge runtime used in Azure, KubeEdge, and EdgeX integrates too much functionalities like data caching, device management, and message routing. However, these functionalities can be redundant for specific scenarios. While TinyEdge aims at providing a flexible solution, which only offers two key functionalities, i.e., cloud-edge interaction and service management, keeping other functionalities in the module market. Users can pull other functionalities from the cloud when they need.

7 EVALUATION

In this section, we present the evaluation of TinyEdge. Section 7.1 presents the evaluation cases. Sections 7.2–7.5 shows the main results in terms of the system-, application-level customization, performance modeling, system optimization, and overhead.

7.1 Evaluation Cases

We use three real-world cases to evaluate different facets of TinyEdge. They are representative because they (1) cover the main aspects of edge computing dataflow, i.e., data collection,

Table 1. Deployment Component Comparison Between TinyEdge and State-of-the-Art Edge Platforms

Notations			
√	Have the out-of-the-box component	X	Do not have the component
⋇	Have similar component but not ready to use	-	No need to use the component in the case

Platform	TinyEdge			Azure IoT Edge			EdgeX			KubeEdge			Cloudlet			ParaDrop		
Component/Case	IoT	EI	GIoTTO	IoT	EI	GIoTTO	IoT	EI	GIoTTO	IoT	EI	GIoTTO	IoT	EI	GIoTTO	IoT	EI	GIoTTO
Timeseries Database	√	-	√	X	-	X	X	-	X	X	-	X	X	-	X	X	-	X
Traditional Database	-	-	√	-	-	√	-	-	X	-	-	√	-	-	X	-	-	X
Device Connector	√	√	√	√	√	√	√	√	√	√	√	√	X	X	X	⋇	√	⋇
Data Visualization	√	-	√	X	-	X	X	-	X	X	X	X	X	X	X	X	X	X
Intelligent Analysis	-	√	√	-	⋇	⋇	-	X	X	-	X	X	-	X	X	-	X	X
Device Management	-	-	√	-	-	X	-	-	⋇	-	-	√	-	-	X	-	-	X
Virtual Device	√	√	√	√	√	√	√	√	√	X	X	X	X	X	X	√	√	√

processing, visualization, and storage, which have already widely testified in the field and adopted in edge computing benchmarks [54]; (2) are comprehensive enough to support a system composed of device management, authentication, IoT device connector, in addition to above data-related functionalities.

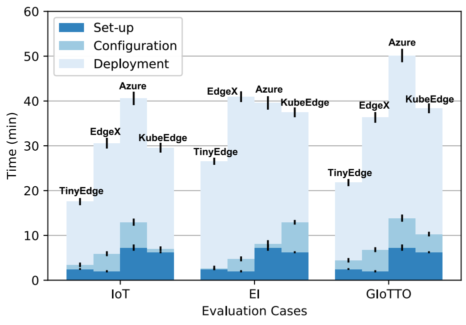
— **Data connection and visualization (IoT):** Reading data from a temperature and humidity sensor through different transmission protocols (including HTTP, MQTT, BLE, and Modbus), then storing the data in InfluxDB and visualizing them via Grafana.
— **Intelligent data processing (EI):** Receiving data input through HTTP, posing them to different edge intelligent applications (including object recognition, speech-to-text conversion, text-audio synchronization, and real-time face detection), then returning the results.
— **A hybrid-analysis system (GIoTTO):** It is part of the CMU GIoTTO project [18] that can receive sensor data via HTTP and MQTT protocol, store data in InfluxDB and visualize via Grafana, support virtual sensors' training and testing, store physical and virtual sensor information in MySQL.

7.2 Customization Related Results

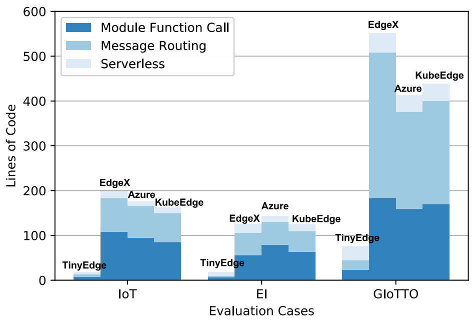
Above all, we qualitatively compare the deployment component of the aforementioned evaluation cases using TinyEdge, EdgeX, Azure IoT Edge, KubeEdge (which our previous work [79] did not consider), Cloudlet, and ParaDrop (from literature). Table 1 summaries the modules information of each case with respect to each platform, where ✓ means the platform has the module with the exact same capabilities; ⋇ means only a similar module is available and extra configuration is needed; - means the module is not required in this case; × means the platform does not have the module yet. For fairness, we use built-in modules that are marked with ✓ and ⋇ in baseline platforms; and replace modules marked × with TinyEdge modules.

Note that we compare TinyEdge's customization results only to EdgeX, Azure IoT Edge, and KubeEdge because they represent state-of-the-art industrial edge computing platforms that are relatively mature, support third-party modules integration, barely have hardware dependencies, and more importantly, use Docker as the backbone.

System-level customization. We abstract the end-to-end system customization into three stages, environment set-up (container and edge system/application project creation), configuration for modules, and deployment (container image building and pulling) three stages. Each stage was carried out by volunteers with different experiences in edge development multiple times followed by a step-by-step manual and the averaged results are given. Figure 7(a) shows the system-level customization time comparison of the four platforms. We only present the overall result rather than the breakdown of each module for the sake of clarity.



(a) System-level customization: Time comparison.



(b) Application-level customization: LoC comparison.

Fig. 7. Customization service-related results.

We can see that: (1) With the help of configuration partitioning and sharing, TinyEdge module configuration time is obviously shorter than the baseline; (2) assisted by a more concise workflow, OS-specific deployment tool, and smaller module size, TinyEdge has a faster deployment process.

Application-level customization. Similar to system-level customization, we abstract application-level customization into three stages, module function call (including necessary connection set-up, functional operations, and so on), message routing (defining interactions between modules both on edge and cloud), and serverless code (other supplementary logic that is beyond the capability of the selected modules). Figure 7(b) shows the lines of code comparison of four edge platforms.

We can see that the lines of code are greatly reduced with TinyEdge. The main reason is that TinyEdge wraps up module function call and message routing into a much simpler form, while users still need to write codes to accomplish the same functionality when using other platforms. For example, message routing in Azure IoT Edge requires users to configure each routing destination that has about 20 lines of extra code for each module. Moreover, the baseline platforms require users to follow different processes and use various methods to call distinct modules' functionality, which leads to a much steeper learning curve than TinyEdge.

In summary, TinyEdge can achieve rapid customization at both systems- and application-level. There are, however, situations that our current system cannot well support. For example, our system cannot be directly used in MEC with requirements to configure many parameters in the communication stack. Regarding this issue, developers can still benefit from our system. Following our system framework, developers can provide necessary modules in MEC by wrapping essential functions and the corresponding runtime into docker images and exposing necessary APIs, configurations, and performance models. Our system can then provide customization and performance estimation services by integrating their modules into our framework.

7.3 Performance Estimation Related Results

In this subsection, we carry out experiments to evaluate the performance of TinyEdge workload and latency models.

Preliminaries. Before the evaluation results, we first present the basic setup of the experiments.

(1) *Baseline models.* Considering the limited resources available on edge devices, and it's quite hard to get thousands of training samples under different hardware specifications, we only include traditional machine learning models rather than deep neural networks that are popular in recent years. At the current stage, we compare TinyEdge with three traditional algorithms: linear regression, SVM, and random forest.

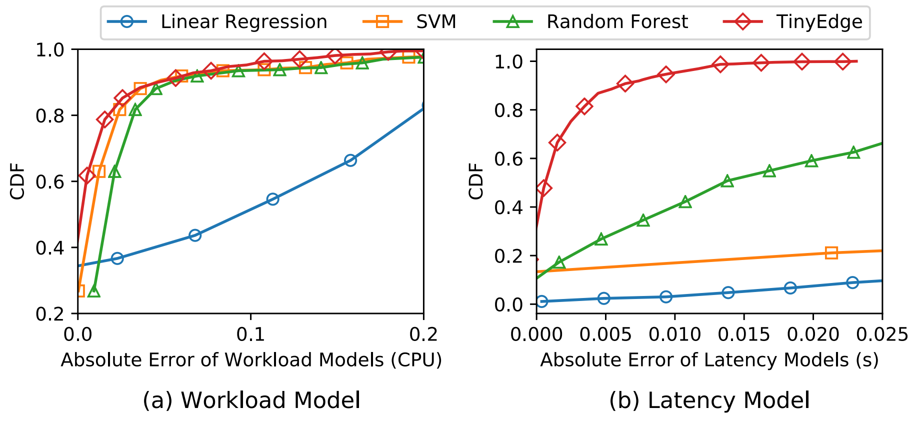


Fig. 8. Performance modeling methods comparison.

(2) *Data collection.* To get enough sampling data under different hardware specifications, we use a CPU frequency scaling tool called `cpupower` [67] to alter CPU frequency, use a network emulator NetEm [49] to adjust network speed, and write a script to generate different workloads. We've also set various resource limitation strategies like `cpuQuota`, and `memoryLimit` in Docker to simulate different resource conditions. We sample data under each condition several times and get the average to reduce variance. Finally, we've obtained around 1.5 k for training and 0.5 k for testing in each case.

Performance model comparison. We build models for processing and connecting modules of all three cases. The overall mean absolute error rate for the workload and latency model is 0.83% and 15.47% respectively.

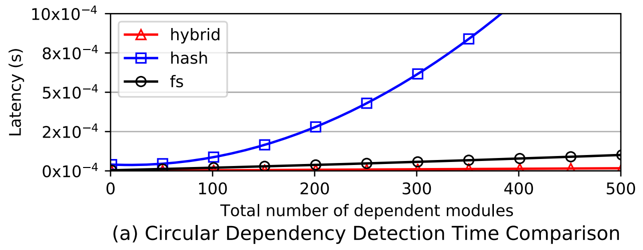
(1) *Workload models.* Figure 8(a) shows the comparison results of TinyEdge workload model and other baseline models. The results indicate that TinyEdge, SVM, and Random Forest perform well as the workload model (whose output is mainly CPU utilization) has a small variation under different settings and makes it more predictable. While the linear regression performs poorly because when the workload reaches the bottleneck, it's hard to get more CPU shares for the lack of resources. Note that for the workload model, TinyEdge uses a function selector to choose the best one as its backbone algorithm, so it will obviously perform better than the baseline.

(2) *Latency models.* Figure 8(b) shows the comparison results of TinyEdge latency model and other baseline models. The results indicate that TinyEdge performs obviously better than other baseline models mainly because TinyEdge not only considers the deep-down mechanisms for both processing and connecting modules using a more realistic queuing model but also leverages a certain amount of sampled data to build a machine learning model to compensate the variation under different settings. While the baseline models perform not so well because (1) the correlations of input features and latency are too complex to model using simple models; (2) the sampled data may not be sufficient enough to fully unveil the potential of black-box methods.

7.4 System Optimization Related Results

In this section, we present the benefits of a series of optimization techniques TinyEdge adopts.

Dependency checking efficiency. As TinyEdge is still in the primary stage of development, the modules and their dependency are not sufficient enough to run a thorough experiment, so we carry out a simulation with 500 simple dependent modules (no module versioning included)



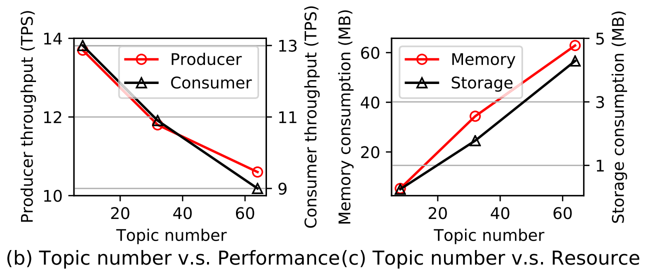


Fig. 9. Dependency and topic generation evaluation results.

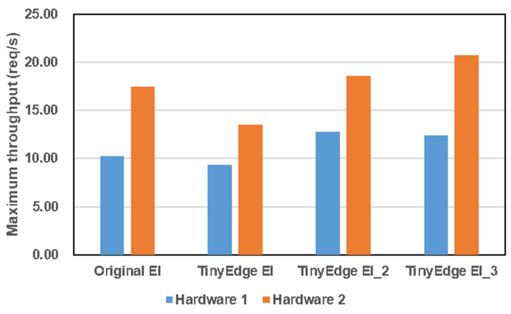
Table 2. Module Size Before and After Compaction

Case	Module	Original	Optimized	Reduction
IoT	(1) Virtual Device	930 MB	105 MB	**54.10%**
	(2) MQTT	85 MB	85 MB	
	(3) InfluxDB	260 MB	260 MB	
	(4) Grafana	250 MB	250 MB	
EI	(1) Virtual Device	Same as above		**56.87%**
	(5) Obj Recognition	1980 MB	1150 MB	
GIoTTO	(1) (2) (3) (4)	Same as above		**65.70%**
	(6) Device Management	935 MB	120 MB	
	(7) Authentication	935 MB	120 MB	
	(8) Virtual Sensor	1210 MB	390 MB	
	(9) MySQL	380 MB	380 MB	

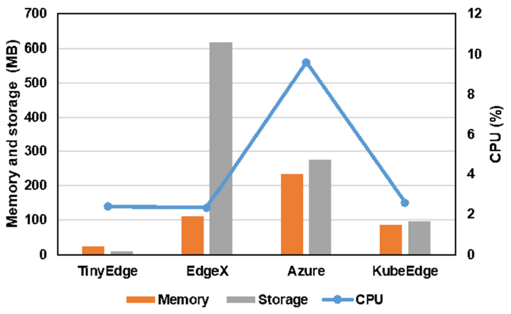
and apply three circular dependency detection algorithms. Figure 9(a) gives the comparison result, which shows that the TinyEdge hybrid approach performs better than the other two methods, considering the dynamics of edge systems update.

Module compaction. In order to evaluate the efficiency of module compaction techniques TinyEdge adopts, we build two sets of modules for the above three cases, and compared the ultimate module size before (marked as "Original" in Table 2) and after (marked as "Optimized" in Table 2) module compaction. Results show that the module compaction techniques can efficiently reduce the module size with an average of 58.89%.

Dynamic topic generation. For the lack of current cases, we also carry out simulations to justify the efficiency of TinyEdge dynamic topic generation technique. We manually generate 1–64 topics and measure the throughput of producer and consumer, as well as their resource consumption. We can see from Figure 9(b) and (c) that as the number of topics goes up, the throughput of both producer and consumer drop rapidly, and the memory and storage consumption gradually go up. Under current combinations of TinyEdge modules, we find out that the total number of topics can be reduced by around 20%, which, as a result, will improve the throughput of the message queue engine and reduce the resource consumption proportionally.



(a) Maximum throughput comparison.



(b) Runtime overhead comparison.

Fig. 10. System overhead comparison.

7.5 System Overhead

In this subsection, we will evaluate the overhead of TinyEdge in terms of backbone container engine and runtime.

The overhead of the Docker backbone of TinyEdge. A decoupled system may suffer performance degradation compared with a monolithic one. Previous works have proved that the performance of Docker is not only acceptable in comparison to native code but also better off than other alternatives like KVM, Xen, and LXC in general cases [19, 39, 57, 58, 62]. To further justify the overhead of Docker, we deploy EI customized by TinyEdge and the original EI on two different edge platforms: one equips a 2.9 GHz CPU with 4 cores and 8 threads (Hardware 1), and another equips a 3.4 GHz CPU with 6 cores and 12 threads (Hardware 2). We use the maximum throughput as a micro-benchmark to evaluate the overhead. Figure 10(a) shows the comparison.

Results show that performance degradation exists and it is closely related to computing power. When the computing power is relatively limited, monolithic and modular implementations have similar performance; as the computing power gets larger, the degradation becomes eminent. We further implement EI with 2 and 3 replicas. We can see that their maximum throughput outperforms that of the monolithic implementation. We think stringent resources limit the full potential of both monolithic and modular implementations; this phenomenon is more obvious in the monolithic setting. Although modular implementation suffers from interactions overhead, it is easier to extend and migrate.

The overhead of TinyEdge runtime. Lastly, we evaluate the minimal resource consumption of runtime among TinyEdge, EdgeX, Azure IoT Edge, and KubeEdge in terms of CPU, memory, and storage. We use the same edge device, carrying out the measurement separately several times, and give the average result in Figure 10(b). We can see that TinyEdge basically achieves the lowest runtime overhead in comparison to other baselines.

8 CONCLUSION

In this article, we present TinyEdge, a holistic framework to support rapid edge system customization for IoT applications. TinyEdge uses a top-down approach to software design. Users only need to select and configure modules of an edge system and specify critical interaction logic with TinyEdge DSL, without worrying about the underlying hardware. TinyEdge takes the configuration as input and automatically generates the deployment package as well as the performance models after sufficient profiling. We implement TinyEdge and evaluate its performance using benchmarks and three real-world case studies. Results show that TinyEdge achieves rapid customization of edge systems, reducing 44.15% of customization time and 67.79% of lines of code on average while giving accurate performance estimation.

REFERENCES

[1] Ahmed Al-Ansi, Abdullah M. Al-Ansi, Ammar Muthanna, Ibrahim A. Elgendy, and Andrey Koucheryavy. 2021. Survey on intelligence edge computing in 6G: Characteristics, challenges, potential use cases, and market drivers. *Future Internet* 13, 5 (2021), 118.

[2] Apache. 2019. Apache Edgent. Retrieved July 7, 2019 from http://edgent.apache.org/.

[3] Jose A. Ayala-Romero, Andres Garcia-Saavedra, Xavier Costa-Perez, and George Iosifidis. 2021. Bayesian online learning for energy-aware resource orchestration in virtualized rans. In *Proceedings of the IEEE INFOCOM*. 1–10.

[4] Baidu. 2019. Baidu IntelliEdge. Retrieved July 7, 2019 from https://cloud.baidu.com/product/bie.html.

[5] Ketan Bhardwaj, Ming-Wei Shih, Pragya Agarwal, Ada Gavrilovska, Taesoo Kim, and Karsten Schwan. 2016. Fast, scalable and secure onloading of edge functions using airbox. In *Proceedings of the IEEE/ACM SEC*. 14–27.

[6] Jianyu Cao, Wei Feng, Ning Ge, and Jianhua Lu. 2020. Delay characterization of mobile-edge computing for 6G time-sensitive services. *IEEE Internet of Things Journal* 8, 5 (2020), 3758–3773.

[7] Yongce Chen, Yubo Wu, Y. Thomas Hou, and Wenjing Lou. 2021. mCore: Achieving sub-millisecond scheduling for 5G MU-MIMO systems. In *Proceedings of the IEEE INFOCOM*. 1–10.

[8] Zicheng Chi, Yan Li, Xin Liu, Yao Yao, Yanchao Zhang, and Ting Zhu. 2019. Parallel inclusive communication for connecting heterogeneous IoT devices at the edge. In *Proceedings of the ACM SenSys*. 205–218.

[9] Alibaba Cloud. 2019. Link Edge. Retrieved July 7, 2019 from https://iot.aliyun.com/products/linkedge.

[10] CNCF. 2019. KubeEdge. Retrieved July 7, 2019 from https://kubeedge.io/en/.

[11] CNCF. 2022. K3s: Lightweight Kubernetes. Retrieved August 7, 2022 from https://k3s.io/.

[12] CNCF. 2022. WasmEdge: Bring the cloud-native and serverless application paradigms to edge computing. Retrieved August 7, 2022 from https://wasmedge.org/.

[13] Li Da Xu, Wu He, and Shancang Li. 2014. Internet of things in industries: A survey. *IEEE Transactions on Industrial Informatics* 10, 4 (2014), 2233–2243.

[14] Bradley Denby and Brandon Lucia. 2020. Orbital edge computing: Nanosatellite constellations as a new class of computer system. In *Proceedings of the ACM ASPLOS*. 939–954.

[15] Wei Dong, Borui Li, Gaoyang Guan, Zhihao Cheng, Jiadong Zhang, and Yi Gao. 2020. TinyLink: A holistic system for rapid development of IoT applications. *ACM Transactions on Sensor Networks (TOSN)* 17, 1 (2020), 1–29.

[16] Salvatore D'Oro, Leonardo Bonati, Michele Polese, and Tommaso Melodia. 2022. OrchestRAN: Network automation through orchestrated intelligence in the open RAN. In *Proceedings of the IEEE INFOCOM*. 1–10.

[17] Ltd. EMQ Technologies Co. 2019. EMQ: The Leader in Open Source MQTT Broker. Retrieved July 7, 2019 from https://www.emqx.io/.

[18] IoT Expedition. 2019. IoT Expedition: A large-scale deployment of Internet of Things that is extensible, privacy-sensitive, and end-user-programmable. Retrieved July 7, 2019 from https://iotexpedition.org/.

[19] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An updated performance comparison of virtual machines and linux containers. In *Proceedings of the IEEE ISPASS*. 171–172.

[20] Xenofon Foukas and Bozidar Radunovic. 2021. Concordia: Teaching the 5G vRAN to share compute. In *Proceedings of the ACM SIGCOMM*. 580–596.

[21] OpenJS Foundation and Node-RED contributors. 2022. Node-RED: Low-code programming for event-driven applications. Retrieved August 7, 2022 from https://nodered.org/.

[22] Silvery Fu and Sylvia Ratnasamy. 2021. dSpace: Composable abstractions for smart spaces. In *Proceedings of the ACM SOSP*. 295–310.

[23] Gines Garcia-Aviles, Andres Garcia-Saavedra, Marco Gramaglia, Xavier Costa-Perez, Pablo Serrano, and Albert Banchs. 2021. Nuberu: Reliable RAN virtualization in shared platforms. In *Proceedings of the ACM MobiCom*. 749–761.

[24] Gaoyang Guan, Wei Dong, Yi Gao, Kaibo Fu, and Zhihao Cheng. 2017. Tinylink: A holistic system for rapid development of iot applications. In *Proceedings of the ACM MobiCom*. 383–395.

[25] Gaoyang Guan, Wei Dong, Jiadong Zhang, Yi Gao, Tao Gu, and Jiajun Bu. 2019. Queec: QoE-aware edge computing for complex IoT event processing under dynamic workloads. In *Proceedings of the TURC*. 1–5.

[26] Gaoyang Guan, Borui Li, Yi Gao, Yuxuan Zhang, Jiajun Bu, and Wei Dong. 2020. TinyLink 2.0: Integrating device, cloud, and client development for IoT applications. In *Proceedings of the ACM MobiCom*. 1–13.

[27] Peizhen Guo, Bo Hu, and Wenjun Hu. 2021. Mistify: Automating DNN model porting for on-device inference at the edge. In *Proceedings of the USENIX NSDI*. 705–719.

[28] Najmul Hassan, Kok-Lim Alvin Yau, and Celimuge Wu. 2019. Edge computing in 5G: A review. *IEEE Access* 7 (2019), 127276–127289.

[29] Chuang Hu, Wei Bao, Dan Wang, and Fengming Liu. 2019. Dynamic adaptive DNN surgery for inference acceleration on the edge. In *Proceedings of the IEEE INFOCOM*. 1423–1431.

[30] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. 2015. Mobile edge computing—A key technology towards 5G. *ETSI White Paper* 11, 11 (2015), 1–16.
[31] IFTTT. 2022. IFTTT. Retrieved August 7, 2022 from https://ifttt.com/.
[32] Home Assistant Inc. 2022. Home Assistant: Open source home automation that puts local control and privacy first. Retrieved August 7, 2022 from https://www.home-assistant.io/.
[33] Tuya Inc. 2022. Tuya IoT Platform. Retrieved August 7, 2022 from https://www.tuya.com/.
[34] Mariam Ishtiaq, Nasir Saeed, and Muhammad Asif Khan. 2021. Edge computing in IoT: A 6G perspective. arXiv:2111.08943. Retrieved August 7, 2022 from https://arxiv.org/abs/2111.08943.
[35] Zongmin Jiang, Yangming Guo, and Zhuqing Wang. 2021. Digital twin to improve the virtual-real integration of industrial IoT. *Journal of Industrial Information Integration* 22 (2021), 100196.
[36] jwilder. 2019. Docker-squash:Squash docker images to make them smaller. Retrieved July 7, 2019 from https://github.com/jwilder/docker-squash.
[37] Nasim Kazemifard and Vahid Shah-Mansouri. 2021. Minimum delay function placement and resource allocation for Open RAN (O-RAN) 5G networks. *Computer Networks* 188 (2021), 107809.
[38] Jin Ho Kim. 2021. 6G and Internet of Things: A survey. *Journal of Management Analytics* 8, 2 (2021), 316–332.
[39] Zhanibek Kozhirbayev and Richard O. Sinnott. 2017. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems* 68 (2017), 175–182.
[40] Jitendra Kumar and Vikas Shinde. 2018. Performance evaluation bulk arrival and bulk service with multi server using queue model. *International Journal of Research in Advent Technology* 6, 11 (2018), 3069–3076.
[41] Borui Li and Wei Dong. 2020. Automatic generation of IoT device platforms with AutoLink. *Internet of Things Journal* 8, 7 (2020), 5893–5903.
[42] Borui Li and Wei Dong. 2020. EdgeProg: Edge-centric programming for IoT applications. In *Proceedings of the IEEE ICDCS*. 212–222.
[43] Shancang Li, Li Da Xu, and Shanshan Zhao. 2018. 5G Internet of Things: A survey. *Journal of Industrial Information Integration* 10 (2018), 1–9.
[44] Yongbo Li, Yurong Chen, Tian Lan, and Guru Venkataramani. 2017. Mobiqor: Pushing the envelope of mobile edge computing via quality-of-result optimization. In *Proceedings of the IEEE ICDCS*. 1261–1270.
[45] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge assisted real-time object detection for mobile augmented reality. In *Proceedings of the ACM MobiCom*.
[46] Liangkai Liu, Xingzhou Zhang, Mu Qiao, and Weisong Shi. 2018. Safeshareride: Edge-based attack detection in ridesharing services. In *Proceedings of the IEEE/ACM SEC*. 17–29.
[47] Peng Liu, Dale Willis, and Suman Banerjee. 2016. Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *Proceedings of the IEEE/ACM SEC*. 1–13.
[48] Lauri Lovén, Teemu Leppänen, Ella Peltonen, Juha Partala, Erkki Harjula, Pawani Porambage, Mika Ylianttila, and Jukka Riekki. 2019. EdgeAI: A vision for distributed, edge-native artificial intelligence in future 6G networks. *The 1st 6G Wireless Summit* (2019), 1–2.
[49] Canonical Ltd. 2019. NetEm 4.15.18. Retrieved from http://manpages.ubuntu.com/manpages/bionic/man8/tc-netem.8.html.
[50] Yang Lu. 2020. Security in 6G: The prospects and the relevant technologies. *Journal of Industrial Integration and Management* 5, 3 (2020), 271–289.
[51] Yang Lu and Xue Ning. 2020. A vision of 6G–5G's successor. *Journal of Management Analytics* 7, 3 (2020), 301–320.
[52] Xiao Ma, Ao Zhou, Shan Zhang, and Shangguang Wang. 2020. Cooperative Service Caching and Workload Scheduling in Mobile Edge Computing. arXiv:2002.01358. Retrieved from https://arxiv.org/abs/2002.01358.
[53] Sumit Maheshwari, Dipankar Raychaudhuri, Ivan Seskar, and Francesco Bronzino. 2018. Scalability and performance evaluation of edge cloud systems for latency constrained applications. In *Proceedings of the IEEE/ACM SEC*. 286–299.
[54] Jonathan McChesney, Nan Wang, Ashish Tanwer, Eyal de Lara, and Blesson Varghese. 2019. DeFog: Fog computing benchmarks. In *Proceedings of the ACM/IEEE SEC*. 47–58.
[55] Matthias Meyer, Timo Farei-Campagna, Akos Pasztor, Reto Da Forno, Tonio Gsell, Jérome Faillettaz, Andreas Vieli, Samuel Weber, Jan Beutel, and Lothar Thiele. 2019. Event-triggered natural hazard monitoring with convolutional neural networks on the edge. In *Proceedings of the ACM/IEEE IPSN*. 73–84.
[56] Microsoft. 2019. Azure IoT Edge. Retrieved July 7, 2019 from https://azure.microsoft.com/en-us/services/iot-edge/.
[57] Roberto Morabito. 2016. A performance evaluation of container technologies on Internet of Things devices. In *Proceedings of the IEEE INFOCOM WKSHPS*. 999–1000.
[58] Roberto Morabito, Jimmy Kjällman, and Miika Komu. 2015. Hypervisors vs. lightweight virtualization: A performance comparison. In *Proceedings of the IEEE ICCE*. 386–393.
[59] Amazon Web Services Inc. or its affiliates. 2019. AWS IoT Greengrass. Retrieved July 7, 2019 from https://aws.amazon.com/greengrass/.

[60] E. B. Priyanka, C. Maheswari, S. Thangavel, and M. Ponni Bala. 2020. Integrating IoT with LQR-PID controller for online surveillance and control of flow and pressure in fluid transportation system. *Journal of Industrial Information Integration* 17 (2020), 100127.

[61] EdgeX Foundry Project. 2019. EdgeX. Retrieved July 7, 2019 from https://www.edgexfoundry.org/.

[62] Moritz Raho, Alexander Spyridakis, Michele Paolino, and Daniel Raho. 2015. KVM, xen and docker: A performance analysis for ARM based NFV and cloud computing. In *Proceedings of the IEEE AIEEE*. 1–8.

[63] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. 2016. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *Proceedings of the IEEE/ACM SEC*. 168–178.

[64] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing* 8, 4 (2009), 14–23.

[65] Chenguang Shen, Rayman Preet Singh, Amar Phanishayee, Aman Kansal, and Ratul Mahajan. 2016. Beam: Ending monolithic applications for connected devices. In *Proceedings of the USENIX ATC*. 143–157.

[66] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. 2017. On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration. *IEEE Communications Surveys and Tutorials* 19, 3 (2017), 1657–1681.

[67] Ubuntu Kernel Team. 2019. cpupower 4.15.18. Retrieved July 7, 2019 from http://manpages.ubuntu.com/manpages/bionic/man1/cpupower-set.1.html.

[68] Tuyen X. Tran, Abolfazl Hajisami, Parul Pandey, and Dario Pompili. 2017. Collaborative mobile edge computing in 5G networks: New paradigms, scenarios, and challenges. *IEEE Communications Magazine* 55, 4 (2017), 54–61.

[69] Fangxin Wang, Feng Wang, Jiangchuan Liu, Ryan Shea, and Lifeng Sun. 2020. Intelligent video caching at network edge: A multi-agent deep reinforcement learning approach. In *Proceedings of the IEEE INFOCOM*. 2499–2508.

[70] Lingdong Wang, Liyao Xiang, Jiayu Xu, Jiaju Chen, Xing Zhao, Dixi Yao, Xinbing Wang, and Baochun Li. 2020. Context-aware deep model compression for edge cloud computing. In *Proceedings of the IEEE ICDCS*. 787–797.

[71] Shibo Wang, Shusen Yang, and Cong Zhao. 2020. SurveilEdge: Real-time video query based on collaborative cloud-edge deep learning. In *Proceedings of the IEEE INFOCOM*. 2519–2528.

[72] Xiufeng Xie and Kyu-Han Kim. 2019. Source compression with bounded dnn perception loss for iot edge computer vision. In *Proceedings of the ACM MobiCom*. 1–16.

[73] Shuochao Yao, Jinyang Li, Dongxin Liu, Tianshi Wang, Shengzhong Liu, Huajie Shao, and Tarek Abdelzaher. 2020. Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency. In *Proceedings of the ACM SenSys*. 476–488.

[74] Yulan Yuan, Lei Jiao, Konglin Zhu, Xiaojun Lin, and Lin Zhang. 2022. AI in 5G: The case of online distributed transfer learning over edge networks. In *Proceedings of the IEEE INFOCOM*. 1–10.

[75] ZangPing. 2019. Compact: A Docker image compression tool. Retrieved July 7, 2019 from https://github.com/wct-devops/compact.

[76] Daniel Zhang, Nathan Vance, Yang Zhang, Md Tahmid Rashid, and Dong Wang. 2019. Edgebatch: Towards ai-empowered optimal task batching in intelligent edge systems. In *Proceedings of the IEEE RTSS*. 366–379.

[77] Qingyang Zhang, Quan Zhang, Weisong Shi, and Hong Zhong. 2018. Distributed collaborative execution on the edges and its application to amber alerts. *IEEE Internet of Things Journal* 5, 5 (2018), 3580–3593.

[78] Quan Zhang, Xiaohong Zhang, Qingyang Zhang, Weisong Shi, and Hong Zhong. 2016. Firework: Big data sharing and processing in collaborative edge environment. In *Proceedings of the IEEE HotWeb*. 20–25.

[79] W. Zhang, Y. Zhang, H. Fan, Y. Gao, W. Dong, and J. Wang. 2020. TinyEdge: Enabling rapid edge system customization for IoT applications. In *Proceedings of the IEEE/ACM SEC*. 1–13.

[80] Zhihe Zhao, Kai Wang, Neiwen Ling, and Guoliang Xing. 2021. EdgeML: An AutoML framework for real-time deep learning on the edge. In *Proceedings of the ACM IoTDi*. 133–144.

Received 3 April 2021; revised 25 April 2022; accepted 29 August 2022