TinyEdge: Enabling Rapid Edge System Customization for IoT Applications

Wenzhao Zhang, Yuxuan Zhang, Hongchang Fan, Yi Gao, Wei Dong, Jinfeng Wang
College of Computer Science, Zhejiang University, China.
Alibaba-Zhejiang University Joint Institute of Frontier Technologies.
{wz.zhang, yx.zhang, fanhc, gaoy, dongw, wangjinfeng}@zju.edu.cn

2020 IEEE/ACM Symposium on Edge Computing (SEC) | 978-1-7281-5943-0/20/$31.00 ©2020 IEEE | DOI: 10.1109/SEC50012.2020.00008

***Abstract*—Customizing and deploying an edge system is a time-consuming and complex task, considering the hardware heterogeneity, third-party software compatibility, diverse performance requirements, etc. In this paper, we present TinyEdge, a holistic system for the rapid customization of edge systems. The key idea of TinyEdge is to use a top-down approach for designing the software and estimating the performance of the customized edge systems under different hardware specifications. Developers select and configure modules to specify the critical logic of their interactions, without dealing with the specific hardware or software. Taking the configuration as input, TinyEdge automatically generates the deployment package and estimate the performance after sufficient profiling. TinyEdge provides a unified customization framework for modules to specify their 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 customization time and 67.79% lines of code on average compared with the state-of-the-art edge platforms; 2) TinyEdge builds compact modules and optimizes the latent circular dependency detection and message queuing efficiency; 3) TinyEdge performance estimation has low average absolute error in various settings.**

I. 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 [1], autonomous vehicle [2], and AR/VR [3], etc.

There exist several edge computing platforms both in academia and industry. For example, Paradrop [4] is a specific edge computing platform that provides computing and storage resources at wireless APs. EdgeX [5] 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) *Rapid customization of edge systems.* Different from edge applications, an edge system can be seen as the aggregation of applications. 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 computing. 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 specify the dataflow interactions among these services easily.

(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, the types of protocols connecting IoT devices and the 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, an edge-computing system to enable rapid development and deployment for data-intensive 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, etc.

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 [5] or serverless function [6] 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, provides a unified domain-specific language to reduce the effort of application coding.

Second, present cloud-based backend falls short in multi-dimensional consideration of performance modeling. State-of-the-art industrial edge platforms like Azure IoT Edge [6] 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 different machine learning algorithms to attain higher accuracy.

We evaluate TinyEdge using real-world edge systems. Results show that: 1) TinyEdge achieves rapid customization of

978-1-7281-5943-0/20/$31.00 ©2020 IEEE
DOI 10.1109/SEC50012.2020.00008

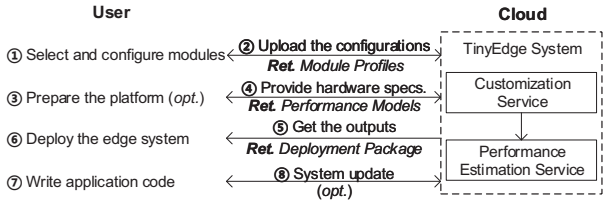

Fig. 1. Workflow overview of TinyEdge.

edge systems, reducing 44.15% of customization time and 67.79% lines of code on average compared with that of EdgeX and Azure IoT Edge; 2) TinyEdge builds precise and practical performance model, considers multi-dimensional information for different kinds of modules.

The contributions of this work are summarized as below:

- We present TinyEdge, a holistic system for customizing and deploying edge systems in a resource-aware manner. TinyEdge offers various highly customizable modules with little high-level configurations, automates detailed procedures, enabling flexible customization and rapid deployment of edge systems.
- We carefully consider different conditions in real-world edge systems to build precise workload and latency models for distinct types of modules.
- We design and implement a unified customization framework for third-party modules that covers module dependencies, interactions, and configuration sharing.
- We carefully evaluate TinyEdge using three concrete cases. Results show that TinyEdge achieves rapid customization of edge systems in terms of customization time and lines of code reduction; generates accurate performance estimation results in various metrics.

The rest of this paper is organized as follows: Section II illustrates the system overview of TinyEdge, including TinyEdge usage and design overview. Section III and IV explicate the TinyEdge customization and performance estimation service respectively. Section V presents the implementation details of TinyEdge system. Section VI shows the evaluation results. Section VII presents related works and Section VIII concludes this work.

II. TinyEdge Overview

In this section, we'll give an overview of TinyEdge in terms of system usage (Section II-A) and design (Section II-B).

A. 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 MQTT protocol, store the data in InfluxD and visualize them by Grafana. As shown in Fig. 1, a user needs to perform the following five steps:

- **Step. ① & ②** Select MQTT Connector, InfluxDB, and Grafana then provide necessary configurations (Fig. 4 and 5, detailed in Section III-A). The user will get a snapshot of module profiles (Fig. 9, detailed in Section IV-A) that contain the approximate minimal storage and memory consumption of the customized system.
- **Step. ③ & ④** Based on the resource consumption, the user can choose to prepare a 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.
- **Step. ⑤ & ⑥** Then the users will get the deployment package that consists of an OS-specific deployment tool (detailed in Section V) with other necessary supplementary configuration files for each module. Fig. 2 shows the deployment package directory architecture in this case. Necessary configuration files are stored under separate directories for each modules. *composer.py* in the root directory is a script that aggregates module configurations; while *deploy.sh* is the deployment tool to set up environment and the customized edge system.
- **Step. ⑦ & ⑧** After deploying the system, the user writes application code (Fig. 7, detailed in Section III-B) to specify the interaction logic, during which may trigger system update.

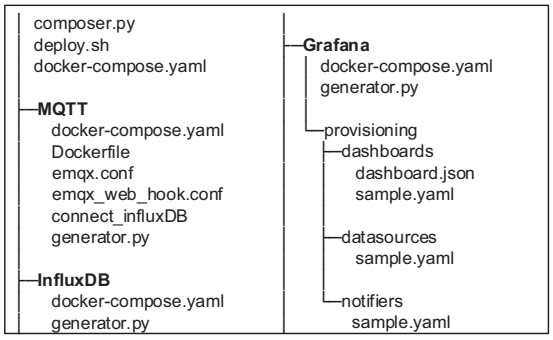

Fig. 2. Deployment package directory architecture of the IoT system.

We can see that TinyEdge workflow is clean and short; users can customize and deploy typical edge computing systems quickly. Performance model 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.

B. TinyEdge Design

Fig. 3 depicts the TinyEdge system architecture, including the customization service (Section III) and the recommendation service (Section IV). The module list serves as the input of TinyEdge and it's passed to both modules:

- 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

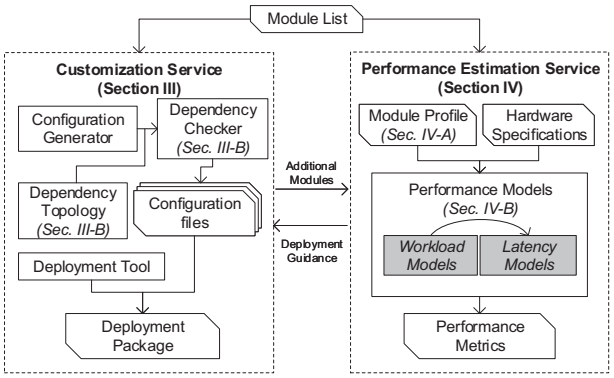

Fig. 3. The overall architecture of TinyEdge.

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 service. 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.

III. Customization Service

The main purpose of TinyEdge customization service is to provide a software and hardware agnostic, easy-to-use system that users can easily ensemble a scenario-specific edge system with much less effort without worrying about module dependencies and write clean and short application code to specify the key logic. In this section, we will explicate this service from two aspects, system-level customization (Section III-A), and application-level customization (Section III-B).

A. System-level Customization

In the system-level customization phase, the user first selects modules and provides necessary configurations for them. TinyEdge dependency checker will then check if the current user-selected modules satisfy module dependency, load necessary additional modules. To facilitate system-level rapid customization, TinyEdge leverages a number of techniques to provide easy configurations (reducing module configuring time) and carefully-checked module dependencies (avoiding module searching time).

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

Configuration reduction methods. After users selecting all the modules, the necessary configurations for each module are required to avoid module malfunction. Existing edge computing platforms are developer-oriented. They assume their users have certain level of developing expertise and left all the configurations of Docker-based modules for users, neglecting the steep learning curve of various edge modules and typical configuration format of Docker/Kubernetes. While there are plenty of configurations for each module, including module-specific and Docker-specific ones. Not all of the configurations are needed in most settings and some of them can be shared across modules. In order to reduce module configuring time, TinyEdge proposes the following two techniques.

(1) *Configuration classification.* For Docker-based modules, existing edge computing platforms require users not only learn module-specific configurations that have many items but also attain complex format of Docker-specific configurations. To enable rapid configuration, we pack up module configurations into 1) basic configurations that a module needs to operate normally and 2) advance ones that deal with more complex situations or performance requirement with default values. To handle sophisticated configuration format, 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 the advance ones are required only when necessary with much less effort without knowing the detail configuration format of Dockerfile/Kubernetes.

We revisit the example shown in Section II-A. Before TinyEdge configuration classification technique is applied, 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 Fig. 4, and repeat this procedure for all modules before he can finally deploy and run the customized system. While after the technique is applied, the user only need to follow the instruction of each TinyEdge modules to provide a unified form of configurations, which mostly are basic ones.

(2) *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 service, they either using RESTful API or serverless function to exchange parameters at runtime or configuring 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 II-A. The user

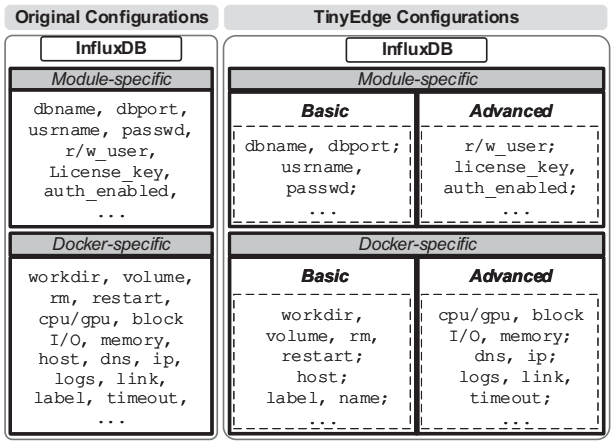

Fig. 4. Configuration classification example of InfluxDB.

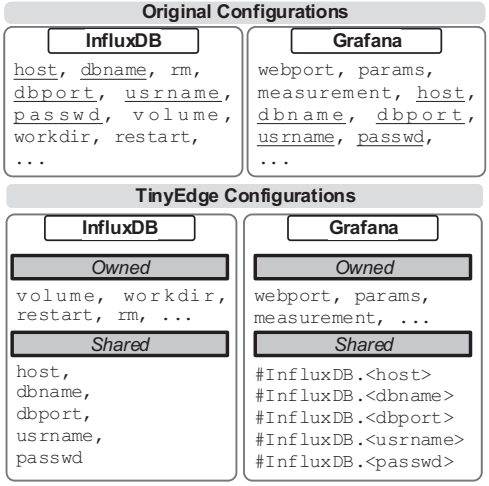

Fig. 5. Configuration sharing example of InfluxDB and Grafana.

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 Fig. 5 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.

Fine-grained module dependency checking. As we know, exhaustively checking package dependency when installing a new one can be a tedious and time-consuming work. While TinyEdge aims to provide rapid edge system deployment, which sacrifices a certain level of portability to achieve more reusability and thus keep each module small. As a result, some TinyEdge modules may depend on other's functionality to operate. One way to resolve the problem is installing all dependent modules. However it requires devices to equip considerable computing resources, which may not suitable for many low-end edge devices.

There are a plethora of package management tools like APT [7] and YUM [8]. But they fall short in fine-grained module dependency checking, especially in rapidly changing edge computing scenarios, which requires highly customizable modules. In such cases, a change of module configuration may require dependency revision. While existing tools only check dependency when a new package is about to install.

For example, TinyEdge stream data connector can be configured to store streaming data locally or to a caching engine like Redis. When users change the configuration from storing data locally to a caching engine, its dependency change from none to a caching engine. To cope with this situation, TinyEdge designs a dependency checking mechanism and a customization format.

In order to resolve fine-grained module dependency dynamically, TinyEdge proposes a dependency checking mechanism, which contains checking policies and the checking algorithm.

TinyEdge maintains a predefined global dependency topology at run-time, when module configurations have changed, either when users first set up a system or change it afterwards, the dependency checker will search the dependency topology to ensure every dependent module are checked. If a module update causes incompatible dependency, the checker will raise a notification, let the user decide whether to abort the update or keep a replica of old version and enforce the update to a new container.

Towards this, TinyEdge includes a dependency checking policy for each module, which is basically a mapping between configuration field and dependency topology changes. For example, the `stored_data` field of the MQTT connector can be `local`, `Redis`, or `InfluxDB`. If this field changes from `local` to `Redis`, the dependency of the MQTT connector in the dependency topology will change from `None` to `Redis`.

The dependency checking algorithm should not only check and include/exclude every dependent module, but also detect circular dependency. One strawman method is using the hash set, which inserts every dependent module into a hash set, the circular dependency is detected when finding out the incoming module is already in the set. Fast-slow pointer algorithm is another commonly used method, the fast pointer takes one more step than the slow one, there is a circular dependency if the two pointers finally meet each other. In our scenario, however, edge system update happens from time to time, including version update of existing modules or addition of new modules, which requires faster circular 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.

The basic workflow of TinyEdge dependency checking algorithm is summarized in Algorithm 1: For every module m in the user-selected module list U, we recursively check whether or not every strongly related module of module m in the dependency topology D is included/excluded, during

```
1 %$deviceManagement%{
2   DROP TABLE IF EXISTS 'device';
3   CREATE TABLE 'device' (
4     'id' int(11) unsigned NOT NULL AUTO_INCREMENT,
5     'name' varchar(100) DEFAULT NULL,
6     'created' datetime DEFAULT NOW(),
7     'changed' datetime DEFAULT NULL,
8     'status' varchar(20) DEFAULT NULL,
9     %$deviceManagement~authentication%{
10    'password' varchar(100) DEFAULT NULL, %}
11    %$deviceManagement~broker%{'is_superuser'
          tinyint(1) DEFAULT 0, %}
12    PRIMARY KEY ('id')
13  )
14 %}
```

Fig. 6. An example of customization code.

```
1 from tinyedge.modules import mqtt, influxdb, grafana
2 from tinyedge.utils import Serverless
3
4 MQTT_Connector = mqtt("emq")
5 data = MQTT_Connector.get_data("device_id")
6 topic_id_1 = MQTT_Connector.Pub(data)
7 func = Serverless(language={"name":"python", "version":"
      3.6"}, package=[{"numpy":"1.14"}], path)
8 func.Sub(topic_id_1)
9 data = func.get_results(**args)
10 topic_id_2 = func.Pub(data)
11 Influx = influxdb("influxdb")
12 data = Influx.Sub(topic_id_2)
13 InfluxDB.insert(data)
14 Grafana = grafana("grafana")
15 data = Grafana.Sub(topic_id_1)
16 Grafana.visualize(data)
```

Fig. 7. Application code snippets of the IoT system.

which the hybrid circular detection method is imposed.

Algorithm 1 Dependency checking algorithm

Input: dependency topology D, user-selected module list U
Output: module list that includes all dependent modules L

function CHECK($D, m, \&L$)
  $L.insert(m)$
  while $D.index(m).next != NULL$ **do**
    $m = m.next$
    if $m.check != True$ **then**
      if $DETECT(m, L)$ **then**
        $CHECK(D, m, \&L)$
      else $RAISE\ 'circular\ detected'$
    else $continue$
  $m.check = True$
if $len(U) >= threshold$ **then**
  $DETECT = Fast\_Slow$
else $DETECT = Hash\_Set$
for all $m \in U$ **do**
  $CHECK(D, m, \&L, DETECT)$
return L

TinyEdge customization format. When the module dependency changes, the functionalities should change accordingly. Towards this, TinyEdge designs a customization format that can help to change module functionality during or after the system customization phase. Specifically, We use % as an escape character, %$ represents the conditions to enable the code block between %{ and %}. For module functionalities, conditions after %$ will have the form like "<module name>.<functionality name>". Fig. 6 is an example code block embedded in TinyEdge MySQL module. When the MySQL module is selected by the user, if the device management is also selected (line 1), then this code block will be loaded; otherwise, it will be deleted. In addition, if the authentication module is selected and its interactions with the device management are enabled, the MySQL will insert the password column in 'device' table (line 9).

B. Application-level Customization

In the application-level customization phase, the user first writes application code using TinyEdge Domain Specific Language (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 learning 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 for customization modules that wraps all functions the modules own. Like *get_data()* of MQTT_Connector and *visualize()* of Grafana in Fig. 7.

② **Message routing.** Within the 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 data and creating topics in TinyEdge message queue engine. While the *Sub()* function is responsible for subscribing topics from TinyEdge 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 configuration and 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 share the merit of ease-of-programming, the serverless function still shows better potentials as it 1) has comprehensive infrastructures both in industrial and open-source community; 2) thoroughly decouples functional and service code modules; 3) has little requirement of module backend, which makes it easier to implement; 4) do not need to operate all the time.

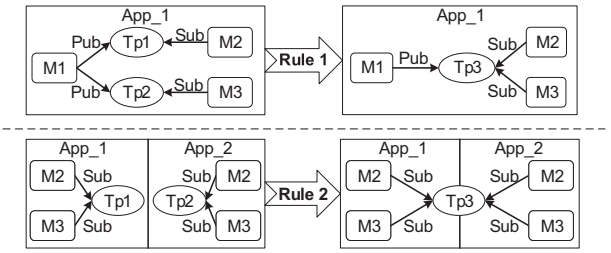

Fig. 8. TinyEdge topic generation mechanism, where M represents Module, Tp represents Topic, App represents Application.

Dynamic topic generation. Although TinyEdge DSL requires defining message routing in a point-to-point manner, i.e., users need to explicitly define the data flow by specifying the publisher and the subscriber, TinyEdge won't actually generate topics accordingly in the following two circumstances:

1) Within the same application, multiple modules subscribe from different topics that are published by the same module.

2) Across different applications in the same system, a same set of modules subscribe from different topics.

For circumstance 1), sending the same data twice to different topics is apparently a waste of resources. While circumstance 2) may also result in sub-optimal resource utilization for similar reasons.

Existing message queue engines like Apache Kafka don't take dynamic topic generation into account as they don't have the global view of the topic semantic information. In order to make better use of edge platform resources, we propose and implement two rules (as shown in Fig. 8) in state-of-the-art message queue engines to cope with the two circumstances mentioned above:

Rule 1: Within the same application, only define one topic for multiple modules subscribe from the same publisher module.

Rule 2: Across different applications in the same system, dynamically merge topics that have the exact same subscriber module.

Reversely, when the two circumstances are no longer satisfied, TinyEdge will split up the merged topics.

However, dynamically generate topics at runtime have the following two problems: 1) Potential message loss. Whenever we try to merge or split topics, the messages may keep coming through, which may cause message loss. 2) None-trivial dynamic topic switching. Topic publishing or subscribing is usually hard-coded in the applications. If we want to switch topics at runtime, we may need to modify the application code, recompile, and restart the whole application, which is definitely too heavy-weight.

For problem 1), TinyEdge will first generate the new topic for both circumstances and migrate all the incoming messages to the new topic but keep the old ones. When the buffer of an old topic is empty, TinyEdge will then destroy the old topic and transfer the feed to the new one.

For problem 2), existing edge computing platforms like Azure IoT Edge leverage module twins, allowing users to update some module configurations at runtime. The problem is twofold. Firstly, Azure runtime have to monitor configuration updates for interested modules, which is resource unfriendly. Secondly, modules have to tightly bind with Azure runtime and Azure cloud services, which has the "locked-in" issue. Other solutions like Spring Boot can help to inject the changed configurations into memory, which, unfortunately, is programming-language-specific and not general enough. While TinyEdge uses MQTT configuration files to enable runtime reload in a general and decoupled way; then we borrow the idea of IO multiplexing techniques that is widely used in caching engine like Redis, aggregating the monitoring tasks of multiple modules into a single process with a queue to reduce the resource consumption.

As a result, the number of topics will be reduced and TinyEdge can leverage load balancing techniques that most message queue engines will provide to balance the traffic load of a topic.

IV. Performance Estimation Service

After building an edge computing system, the developer may wonder what's the performance of this customized system. Given that, 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 under edge computing 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 IV-A) and builds multi-dimensional models to describe key metrics of the customized system (Section IV-B).

A. Module Profile

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

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, etc. 2) Processing modules that take charge of the computation part of an edge system, like data filter, video analyzing, object recognition, etc. 3) Connecting modules that are responsible for accessing IoT devices or transmitting data to the cloud, like HTTP, MQTT, Bluetooth connector, etc.

Customization information is extracted at the system-level customization stage. It records functionality and configuration

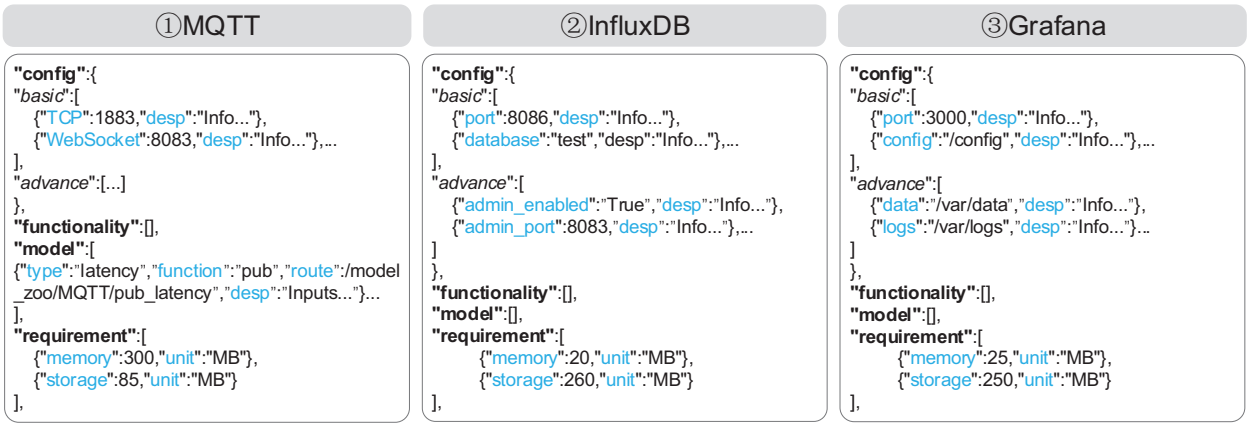

Fig. 9. An example of three module profiles for the IoT system, where **config** stores basic and advance configurations; **functionality** stores the choice of module functionality; **model** stores the route to the module performance model; **requirement** stores resources (including connection) requirement.

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. For example, a Bluetooth connector requires an underlying hardware chipset to operate.

B. Performance Models

There are plenty of performance metrics that can describe different faucets of a system like latency, workload, accuracy, energy, etc. 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, but the results tend to have little variance to the environment, if not, 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 power supply.

Existing solutions like [9] and [10] haven't consider the number of accessing 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-simplify the problem and cause inferior results.

Workload model. The main functionality of 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 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, etc.) and the CPU specifications (the number of cores and threads, main frequency, maximum frequency, etc.), 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 as input and outputs the compensate value.

Given the workload models, we can get a general form of workload for application $\mathcal{A}$:

$$W = \sum_{i=1}^{i \in \mathcal{A}} \omega_i.$$

Latency model. The main functionality of 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 model can sometimes be well described in a more formal mathematical way besides its heterogeneity, 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 [10], [11] 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 [12].

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. For example, assume the InfluxDB requires 0.05 CPU under a specific hardware resource with $C = 4$ CPUs, current available workload is 0.80 CPU, we will first transform the workload into integer by multiplying a workload unit $W_{unit} = 100$, then the corresponding queuing model will be $M^{5}/M^{80}/400$, where $W_{max} = 400$.

With the definition above, we can get the waiting time of module i t_i^{wait} in the system 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 P_0 (the probability of no request in the system) and ρ (system intensity) are defined as follows:

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

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

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 is 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 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.

V. 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.

Module selection. TinyEdge customization modules are used to customize edge systems. In order to decide which modules to include, we conduct a small investigation of 20+ edge computing related papers and 6 mainstream industrial edge computing platforms, and find that the main functionalities of edge systems for IoT applications basically include 1) connector for both IoT devices and cloud services, 2) data processing (like stream analytics and machine learning), 3) traditional database (like SQLite), 4) security.

We implement HTTP, Modbus, and Bluetooth connector, integrate EMQ [13], 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 device management module nor time-series database at the edge. So we implement a lightweight device management module and incorporate InfluxDB (time-series database), MongoDB (semi-relational database), and Grafana (a visualization tool for time-series data) to compensate for the issue.

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

Module compaction. Recall that each TinyEdge module is wrapped in a single Docker image. Lots of factors will affect the size of a module, sometimes a single-line-difference in Dockerfile will generate two modules that identical in

functionality but have hundreds of megabytes variation in module size. In order to reduce module pulling time, TinyEdge adopts two techniques to produce modules of a much smaller size with little impact on the functionalities.

(1) *Use smaller base images.* Base image is basically a operating system in the container, usually the larger a base image is, the more functionalities it can provide. Take python3.6 as an example, Python:3.6-alpine uses alpine as its base image, which takes up around 100 MB; while Python:3.6-buster uses Debian, which takes up almost 1000 MB. However, many functionalities can be redundant like package management and build tools. In order to reduce the module size, we choose smaller base images in TinyEdge.

(2) *Optimized Dockerfile.* When using Dockerfile to build a Docker image, each command in the file will create an image layer. Files from upper layers can not be deleted by lower layers that are created later, they can only be set to invisible. As a result, files should have been deleted are still physically existed. To keep a slim module: 1) TinyEdge borrows the best practice in the Docker community when writing a Dockerfile by grouping file-addition commands like `apt update/upgrade/install` and file-removal ones like `apt clean/autoremove`. 2) TinyEdge will flatten a built image with tools like Docker-squash [14] or Compact [15] to run clean up commands in the container, squash multiple image layers into one and finally generate an even more compact Docker image.

OS-specific deployment tools. The deployment tools are basically a set of scripts that designed for different OSes like Windows, MacOS, and all kinds of Linux distributions. It will take charge of checking necessary runtime environment of TinyEdge (including Docker, Docker-compose, and Kubernetes in multi-device settings), installing and/or upgrading the environment, building and/or running module(s) as configured, and finally activating them all at once. Taking the deployment package as input, the customized system will be deployed, up and running on the edge device with the help of the deployment tools.

TinyEdge runtime. TinyEdge runtime is a module running at the edge, whose responsibility is to parse and execute application code and interact with TinyEdge cloud, enabling the customized system to update. This module is mainly composed of a message queue engine and a serverless engine.

There is a wide range of open-source or commercial message queues and serverless engines. To embrace the diversity of engines and enhance system scalability, TinyEdge abstracts the main functionalities of both message queue and serverless engines, allowing users to implement connectors for different engines as their wish. In this paper, we choose to implement Apache Kafka [16] and OpenFaas [17] connector for TinyEdge message queue engine and serverless engine for their availability and popularity.

VI. Evaluation

In this section, we present the evaluation of TinyEdge. Section VI-A presents the evaluation cases. Section VI-B - VI-C shows the main results in terms of system-, application-level customization, and performance modeling.

A. Evaluation Cases

We use three real-world cases to evaluate different facets of TinyEdge. They are representative because they 1) cover main aspects of edge computing data flow, i.e., data collection, processing, visualization, and storage, which have already widely testified in the field and adopted in edge computing benchmark works like [18]; 2) are comprehensive enough to support a holistic 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 finally 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 [19] 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.

B. Customization Related Results

Above all, we compare the customization results of the aforementioned evaluation cases using TinyEdge, EdgeX, and Azure IoT Edge. We choose latter two as the baseline 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, both use Docker as the backbone. 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 [20], [21], [22], [23], [24].

TABLE I summaries the modules information of each case with respect to each platform, where ✓ means the platform has the module that has the exact same capability; while ⁕ means only a similar module is available and extra configuration is needed; - means the module isn't required in this case; × means the platform doesn't have the module yet. For fairness, we use built-in modules that marked with ✓ and ⁕ in EdgeX and Azure IoT Edge, and follow the set up flow of their own; while those marked with × are replaced by TinyEdge modules and integrated in a third-party manner.

System-level customization. For system-level customization, we carry out experiments to justify the efficiency of

TABLE I
DEPLOYMENT MODULES COMPARISON BETWEEN TINYEDGE AND STATE-OF-THE-ART EDGE PLATFORMS

Platform	TinyEdge			Azure IoT Edge			EdgeX		
Component/Use Case	IoT	EI	GIoTTO	IoT	EI	GIoTTO	IoT	EI	GIoTTO
Time Series Database	√	-	√	X	-	X	X	-	X
Traditional Database	-	-	√	-	-	√	-	-	X
Device Connector	√	√	√	√	√	√	√	√	√
Data Visulization	√	-	√	X	-	X	X	-	X
Intelligent Analysis	-	√	√	-	*	*	-	X	X
Device Management	-	-	√	-	-	X	-	-	*
Virtual Device	√	√	√	√	√	√	√	√	√

Notations	
√	Have the out-of-the-box component
X	Do not have the component
-	No need to use in the case
*	Have similar component but needs extra effort to use

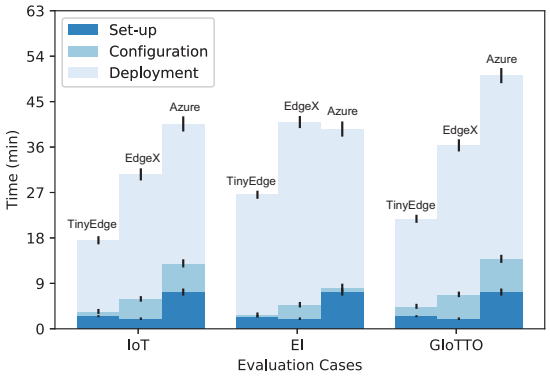

Fig. 10. System-level customization time comparison among TinyEdge, EdgeX, and Azure IoT Edge.

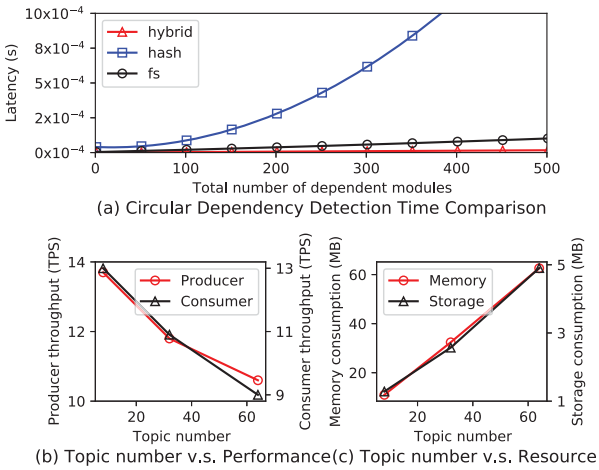

Fig. 11. Dependency and Topic Generation Evaluation Results.

TinyEdge configuration reduction methods, fine-grained module dependency checking and the module compaction methods we apply.

(1) *End-to-end customization time comparison.* 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 experience in edge development multiple times followed by a step-by-step manual and the averaged results are given. Fig. 10 shows the system-level customization time comparison of three platforms. We only present the overall result rather than the break down of each modules for the sake of clarity.

We can see clearly that: 1) With the help of configuration partitioning and sharing, TinyEdge module configuration time is obviously shorter than the baseline; 2) assisted by more concise workflow, OS-specific deployment tool, and smaller module size, TinyEdge can achieve faster deployment process. Note that TinyEdge set-up time is slightly longer than that of EdgeX because TinyEdge requires users to login and create an edge system project; though Azure IoT Edge share the same process as TinyEdge, it has more fields to fill in.

(2) *Dependency checking efficiency comparison.* As TinyEdge still under 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) and apply three circular dependency detection algorithms. Fig. 11 (a) gives the comparison result, which shows that TinyEdge hybrid approach performs better than other two methods, considering the dynamic changing of edge system update.

(3) *Module compaction comparison.* 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 II) and after (marked as "Optimized" in TABLE II) module compaction. Results show that the module compaction techniques can efficiently reduce the module size with an average of 58.89%.

Application-level customization. For application-level customization, we carry out experiments to justify the efficiency of TinyEdge DSL and dynamic topic generation.

(1) *Lines of code comparison.* Similar with system-level customization, we abstract application-level customization into three stages, module function call (including necessary connection set-up, functional operations, etc.), 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). Fig. 12 shows the lines of code comparison of the three platforms.

We can see clearly that the lines of code is greatly reduced

TABLE II
MODULE SIZE BEFORE & AFTER COMPACTION COMPARISON.

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	

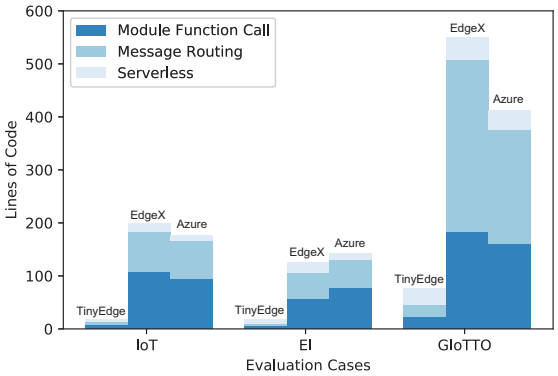

Fig. 12. Application-level customization code comparison among TinyEdge, EdgeX, and Azure IoT Edge.

when using TinyEdge. The main reason is that TinyEdge sacrifices certain level of flexibility and wraps up module 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 EdgeX and Azure IoT Edge. For example, message routing in Azure IoT Edge requires users configure each routing destination that has about 20 lines of extra code for each module; similarly, message routing in EdgeX requires to register a export client with about 25 lines of extra code for each module. Moreover, the baseline platforms require users follow different processes and use various methods to call distinct modules' functionality, which leads to much steeper learning curve than TinyEdge.

(2) *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 & consumer, as well as the resource consumption.

We can see from Fig. 11 (b) & (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 around 20%, which, as a result, will improve the throughput of the message queue engine and reduce the resource consumption proportionally.

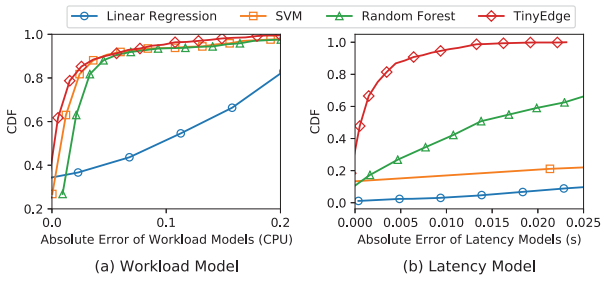

Fig. 13. Performance Modeling Comparison Between TinyEdge and Conventional ML Methods.

C. 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 set up 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 current stage, we compare TinyEdge with three traditional algorithms: linear regression, SVM, and random forest.

(2) *Data collection.* To get enough sampling data under different hardware specifications, we use a CPU frequency scaling tool called cpupower [25] to alter CPU frequency, use a network emulator NetEm [26] to adjust network speed, and write a script to generate different workload. We've also set various resource limitation strategies like `cpuQuota`, `cpuPeriod`, and `memoryLimit` in Docker to simulate different resource conditions. The data sampling under each conditions were carried out several times and get the average to reduce variance. Finally, we've obtained around 1.5k for training and 0.5k for testing in each case.

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

(1) *Workload models.* Fig. 13 (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 workload model, which mainly because the output of workload model, i.e., the CPU utilization, has a small variation under different settings and makes it more predictable. While the linear regression performs poorly because as the workload reach the bottleneck, it's hard to get more CPU shares for the lack of resources. Note that for workload model, TinyEdge uses a function selector to choose a best one as its backbone algorithm, so it will definitely perform better than the baseline.

(2) *Latency models.* Fig. 13 (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, which mainly because TinyEdge not only considers the deep down mechanisms for both processing and connecting modules using a more realistic queuing model, but also leverage certain amount of sampled data to build a machine learning model to compensate the variation under different settings. While the base line models performs not so well because 1) the correlations of input features and latency is 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.

VII. Related Work

We classify the related work into VII-A edge computing platforms and VII-B performance modeling.

A. Edge Computing Platforms

Due to the prominence of edge computing, a number of edge computing platforms with different purposes have emerged, targeting at stream analysis [27], [28], data sharing [29], security [30], and most importantly, deploying systems by integrating cloud-edge-end resources [31], [4], [6], [5]. In this section, we focus on the last category. The comparison between our work and the existing works will also be discussed.

Cloudlet. Cloudlet [31] is devised to instantiate customized service software on edge devices rapidly; also simplify the challenge of meeting the peak bandwidth demand of multiple users interactively generating and receiving media. Cloudlet uses VM overlay as a building block of edge computing system, which results in low extensibility as it's OS-dependent, users need to create different overlays for the same system on alternative OSes.

Paradrop. Paradrop [4] is another representative edge computing platform in academia. It's a specific 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. However, Paradrop doesn't provide resource and performance models, and its instance tool is supported on limited hardware or VM.

EdgeX. EdgeX [5] is an open-source project supported by Linux Foundry, whose 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 [5]. When customizing a new system, EdgeX requires to deploy all its core modules, which incurs high redundancy. Limited modules are supported in EdgeX, a great deal of manual work is needed to integrate new ones.

Azure IoT Edge. Azure IoT Edge [6] is the symbol of industrial edge computing platforms. As an industrial platform with a powerful cloud backbone, Azure IoT Edge can basically satisfy all we need to deploy an edge system, but its the high integration with the cloud that makes it hard to use, users have to get familiar with a wide range of cloud, edge services.

Unlike the above works, TinyEdge not only leverages the container-based design to achieve high extensibility but also a customization framework to specify the dependency, interaction and high-level configuration, as well as the profiling based system performance modeling. These can help to achieve more improvements in customization time, application logic expressiveness, and give coarse-grained guidance for users.

B. 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 [32] 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 [9]: a QoE-aware edge computing system, where they use regression technique to model the execution time and workload of specific edge computing applications like speech and face recognition. Maheshwari et al [10] 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 white and black solution to combine the merits of both methods and integrates the resource constraint into the model, making the estimation results more reliable.

VIII. Conclusion

In this paper, we present TinyEdge, a holistic rapid customization edge system for data-intensive IoT applications. TinyEdge uses a top-down approach for software design. Users only need to select and configure modules of an edge system, specify critical interaction logic with TinyEdge interfaces, without worrying about the underlying hardware. TinyEdge takes the configuration as input, 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% lines of code on average while giving accurate performance estimation in various settings.

Acknowledgment

This work is supported by the National Key R&D Program of China under Grant No. 2019YFB1600700 and National Science Foundation of China under Grant No. 61872437.

REFERENCES

[1] Q. Zhang, Z. Quan, S. Weisong, and Z. Hong, "Distributed collaborative execution on the edges and its application to amber alerts," *IEEE Internet of Things Journal*, vol. PP, no. 99, pp. 1–1, 2018.

[2] L. Liu, X. Zhang, M. Qiao, and W. Shi, "Safeshareride: Edge-based attack detection in ridesharing services," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 17–29.

[3] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *MobiCom*. ACM, 2019.

[4] P. Liu, D. Willis, and S. Banerjee, "Paradrop: Enabling lightweight multi-tenancy at the networks extreme edge," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2016, pp. 1–13.

[5] (2019) Edgex: Community forged the open platform for the iot edge. EdgeX Foundry. [Online]. Available: https://www.edgexfoundry.org/

[6] (2019) Azure iot edge: Cloud intelligence deployed locally on iot edge devices. Azure IoT. [Online]. Available: https://azure.microsoft.com/en-us/services/iot-edge/

[7] (2019) Debian policy manual v4.5.0.0. [Online]. Available: https://www.debian.org/doc/debian-policy/ch-relationships

[8] (2020) Yum package manager. [Online]. Available: http://yum.baseurl.org/

[9] G. Guan, W. Dong, J. Zhang, Y. Gao, T. Gu, and J. Bu, "Queec: Qoe-aware edge computing for complex iot event processing under dynamic workloads," in *Proceedings of the ACM Turing Celebration Conference-China*, 2019, pp. 1–5.

[10] S. Maheshwari, D. Raychaudhuri, I. Seskar, and F. Bronzino, "Scalability and performance evaluation of edge cloud systems for latency constrained applications," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 286–299.

[11] X. Ma, A. Zhou, S. Zhang, and S. Wang, "Cooperative service caching and workload scheduling in mobile edge computing," 2020.

[12] J. Kumar and V. Shinde, "Performance evaluation bulk arrival and bulk service with multi server using queue model," 11 2018.

[13] (2019) Emq: The leader in open source mqtt broker. EMQ Technologies Co., Ltd. [Online]. Available: https://www.emqx.io/

[14] jwilder. (2019) Docker-squash:squash docker images to make them smaller. [Online]. Available: https://github.com/jwilder/docker-squash

[15] ZangPing. (2019) Compact: A docker image compression tool. [Online]. Available: https://github.com/wct-devops/compact

[16] (2019) Apache kafka: A distributed streaming platform. Apache. [Online]. Available: http://kafka.apache.org/

[17] (2019) Openfaas: Serverless functions, made simple. OpenFaas. [Online]. Available: https://www.openfaas.com/

[18] J. McChesney, N. Wang, A. Tanwer, E. de Lara, and B. Varghese, "Defog: fog computing benchmarks," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 47–58.

[19] (2019) Iot expedition: A large-scale deployment of internet of things that is extensible, privacy-sensitive, and end-user-programmable. IoT Expedition. [Online]. Available: https://iotexpedition.org/

[20] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the cloud," *Future Generation Computer Systems*, vol. 68, pp. 175–182, 2017.

[21] R. Morabito, "A performance evaluation of container technologies on internet of things devices," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2016, pp. 999–1000.

[22] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2015, pp. 171–172.

[23] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *2015 IEEE International Conference on Cloud Engineering*. IEEE, 2015, pp. 386–393.

[24] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing," in *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*. IEEE, 2015, pp. 1–8.

[25] (2019) cpupower 4.15.18. Ubuntu Kernel Team. [Online]. Available: http://manpages.ubuntu.com/manpages/bionic/man1/cpupower-set.1.html

[26] (2019) Netem 4.15.18. Ubuntu Kernel Team. [Online]. Available: http://manpages.ubuntu.com/manpages/bionic/man8/tc-netem.8.html

[27] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov, "Spanedge: Towards unifying stream processing over central and near-the-edge data centers," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2016, pp. 168–178.

[28] (2019) Apache edgent: A community for accelerating analytics at the edge. Apache. [Online]. Available: http://edgent.apache.org/

[29] Q. Zhang, X. Zhang, Q. Zhang, W. Shi, and H. Zhong, "Firework: Big data sharing and processing in collaborative edge environment," in *2016 Fourth IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. IEEE, 2016, pp. 20–25.

[30] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan, "Fast, scalable and secure onloading of edge functions using airbox," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2016, pp. 14–27.

[31] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, no. 4, pp. 14–23, 2009.

[32] Y. Li, Y. Chen, T. Lan, and G. Venkataramani, "Mobiqor: Pushing the envelope of mobile edge computing via quality-of-result optimization," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 1261–1270.