Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 3833644, 16 pages
https://doi.org/10.1155/2021/3833644

Research Article

Towards an Elastic Fog-Computing Framework for IoT Big Data Analytics Applications

Linh Manh Pham,[1,2,3] **Truong-Thang Nguyen**,[2] **and Tien-Quang Hoang**[4]

[1]*Graduate University of Science and Technology, Vietnam Academy of Science and Technology, 18 Hoang Quoc Viet, Cau Giay, Hanoi, Vietnam*
[2]*Institute of Information Technology, Vietnam Academy of Science and Technology, 18 Hoang Quoc Viet, Cau Giay, Hanoi, Vietnam*
[3]*VNU University of Engineering and Technology, 144 Xuan Thuy, Cau Giay, Hanoi, Vietnam*
[4]*Hanoi Pedagogical University 2, 32 Nguyen Van Linh, Xuan Hoa, Vinh Phuc, Vietnam*

Correspondence should be addressed to Linh Manh Pham; linhmp@vnu.edu.vn

Received 3 May 2021; Revised 8 June 2021; Accepted 28 July 2021; Published 15 August 2021

Academic Editor: Yong Zhang

IoT applications have been being moved to the cloud during the last decade in order to reduce operating costs and provide more scalable services to users. However, IoT latency-sensitive big data streaming systems (e.g., smart home application) is not suitable with the cloud and needs another model to fit in. Fog computing, aiming at bringing computation, communication, and storage resources from "cloud to ground" closest to smart end-devices, seems to be a complementary appropriate proposal for such type of application. Although there are various research efforts and solutions for deploying and conducting elasticity of IoT big data analytics applications on the cloud, similar work on fog computing is not many. This article firstly introduces AutoFog, a fog-computing framework, which provides holistic deployment and an elasticity solution for fog-based IoT big data analytics applications including a novel mechanism for elasticity provision. Secondly, the article also points out requirements that a framework of IoT big data analytics application on fog environment should support. Finally, through a realistic smart home use case, extensive experiments were conducted to validate typical aspects of our proposed framework.

1. Introduction

1.1. Deployment of IoT Big Data Analytics Applications in the Context of Fog Computing. The last decade has seen the emerging of cloud computing as a trendy technology and business model bringing to incremental value for cloud stakeholders. At the side of service consumers, this value comes from saving both deployment time and investment cost on IT infrastructure, offloading it to cloud service providers. Taking advantage of virtualization technology, hardware resources at data centers are shared by infrastructure service providers and sold to customers with a reasonable price. Platform or application services built on top of virtual resources now are delivered to the hands of cloud consumers with invoices billed by fine-grained time or resource unit like in the water or electric power industries.

However, Internet of Things (IoT) applications are not appropriate to be converted completely into services of the cloud computing model. For example, latency-sensitive IoT applications such as big data analytics (BDA for short) systems require prompt responses to outliers which need to be within several milliseconds or even microseconds in some special emergency cases. Moreover, these BDA systems can create petabytes of data that are not practical to stream back and forth between cloud data centers and end-devices. With this kind of application, it is better to keep some of their components staying at centralized clouds and move some of them down to the edge close to end-devices. The components at the edge should take care of operations requiring fast responses or reducing a huge amount of data which may consume much bandwidth if transferred over a wide area network. On the contrary, the components on the cloud is often responsible for compute-intensive long-running tasks on permanent data storage.

The fog-computing model, aiming at bringing computation, communication, and storage resources from "cloud to ground" closest to end-devices, seems to be an appropriately complementary proposal for such type of application. This model encourages developers to divide their applications into more fine-grained components deployed in the nodes throughout from the cloud, fog, to end-device strata as shown in Figure 1. The position of a component depends on specific tasks that this component has to be in charge of and whether its tasks are latency-sensitive or not.

Elasticity is a characteristic of cloud according to NIST [1]. Many cloud applications provide elasticity features to accommodate scalability and adapt to changes in demand. Unlike in the cloud, implementing elasticity for IoT applications in the fog is not an easy task, especially for BDA ones. An elasticity-supported platform for this kind of application must provide the ability to modelize all heterogeneous software and hardware components participating in these applications and distribute them throughout strata from the cloud, fog, to end-devices. The components of these modelized applications should be migrated horizontally between fog nodes or vertically between cloud and fog strata where there is an increase or decrease in the density of the end-devices. These components should be also duplicated where workloads from the end-devices increase. To avoid vendor lock-in issues, those components also should be moved flexibly among IoT service providers when needed. Application programming interfaces need to be provided so that intercommunication among components can transparently cross boundaries created by different communication protocols.

In short, we are lacking a holistic fog framework which allows IoT BDA service providers to deploy and conduct elasticity on their applications to adapt fluctuation of big data workload coming from various IoT end-devices.

1.2. Our Contributions. Before talking about our contributions, we discuss dedicated features that a fog elasticity framework should support in addition to the features inherited from the cloud.

First, a distributed application needs to be modelized before its automatic deployment. In cloud computing, a standard modeling domain-specific language such as TOSCA [2] or Camel [3] is enough to abstract both software and hardware components of application services. However, in cloud computing, it is often that only a small set of concepts need to be described such as "cloud provider," "virtual machine," or "hosting server." These sets of terms need to be extended to fulfill the demand of describing a large of new concepts of fog computing coming from widely heterogeneous components such as gateway, set-top box, base station, physical machine, and cloudlet.

Secondly, mobility is a conspicuous characteristic of end-devices in fog computing. The end-devices can move or are moved physically from this geographical area to another one such as vehicles and smartphones or eliminated at one place and replaced logically at another position such as short-lived wireless sensors. Along with these movements, some components located at the upper strata of a fog application such as fog or cloud nodes should be moved or migrated correspondingly. An elasticity controller for fog-based applications needs to provide modules to take the end-device mobility and its migrated workload into consideration.

In the third place, software or hardware components in fog do not always stay with the same provider. They can be distributed horizontally between fog providers or vertically across both fog and cloud providers. Moreover, one component can be found in the fog at this moment but can be migrated to the cloud at another time when some input conditions vary. Therefore, the cloud/fog federation to break the vendor lock-in issue is another concern for developers of fog elasticity tools.

Finally, fog computing's ecosystem is dominated by millions of chatty embedded devices with thousands kind of communication protocols. On the other hand, complex applications have many components which need to be interconnected to enhance automation and autonomy. These interconnections can be either fog-fog, fog-cloud, cloud-cloud, fog-devices, or cloud-devices. Providing mechanisms for interconnection between the components is one of the critical missions of a modern fog elasticity platform.

Supporting such extended requirements needs a holistic framework that can catch the required aspects of configuration to deliver a highly automated system for managing any IoT BDA application on the fog. In this paper, we present AutoFog, which supports the transformation of complex applications to fog-based ones as well as supports their large-scale automatic deployment and scaling. The transformation is smooth and less time-consuming, thanks to the reuse and extension of an existing domain-specific language (DSL) [4] and off-the-shelf components (COTS). Besides the extension of the DSL, another contribution is the introduction of a mechanism for automatic deployment and elasticity provisioning which automatically implements all the predefined component instances as well as monitors and conducts all the elasticity strategies in fog environment. The last one is a runtime system ensuring that the deployed fog application is properly globally configured while scaling the application model such as adding, removing, or migrating component instances including fog/cloud nodes. We also have implemented a prototype for the framework and conducted extensive experiments to evaluate AutoFog in deploying and scaling a real-world IoT BDA application on typical aspects that a fog-computing elasticity framework should resolve.

The rest of this article is organized as follows. Section 2 describes a real-world complex IoT BDA use case as a fog-based application that we use throughout this paper. Section 3 discusses important modules of our proposed framework. Section 4 presents our mechanism for dynamic deployment and elasticity provision of IoT applications. Section 5 reports some extensive experiments performed on a prototype of AutoFog serving as a proof of concept of our work. Finally, section 6 presents related work, and section 7 concludes the paper.

2. Use Case

In fog computing, components of an application are not only divided into tiers but also distributed to the three strata:

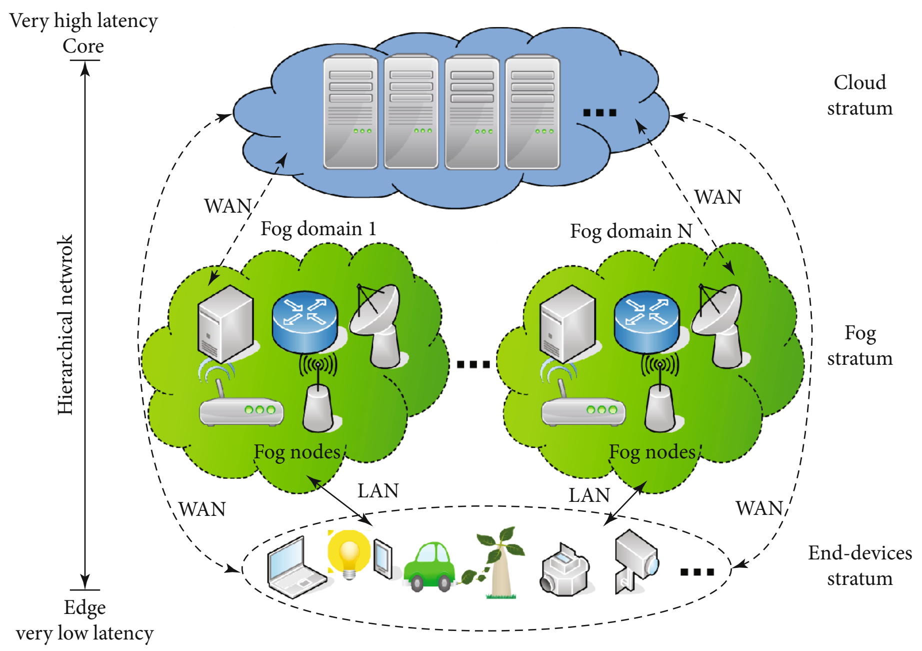


Figure 1: Three strata of fog-computing environment.

cloud, fog, and end-devices. We consider in this section a use case of a complex IoT BDA application which is divided into strata as shown in Figure 2. The components of the application can appear at positions marked by package icons. It is an energy management application for smart homes. In the application model, the smart home center manages and monitors the power consumption of multiple houses in a district.

There may have thousands of end-devices that consume electricity serving for human regular activities in the houses. Some mobile end-devices can be moved between the houses such as smartphones, laptops, and vehicles. This movement can cause temporarily peak demand in some discrete houses. In each household of a house, many IoT smart plugs are implemented to measure the energy consumption of end-devices. A smart plug is installed between the electrical smart device and the wall power outlet. A range of sensors is equipped in a smart plug to measure various values of associated power consumption. These raw data from thousands of smart plugs are sent to local agencies of the smart home center located closest to the corresponding houses for preprocessing or abnormal detection. Fog nodes in clusters, cloudlets, or private clouds are implemented in the local agencies to perform these operations. If an anomaly such as an unusual peak load or an outage is detected at this step, corresponding reactions are triggered from the application components distributed to the fog nodes. These reactions can be sending a simple notification to the administrator or adding more electric power from renewable energy sources to fulfill a peaking consumption demand. To ensure that these operations are fast and timely, the connection from the local agencies to the houses must be ensured by high-speed local transmission lines.

The preprocessed data are sent to more compute-intensive nodes in the cloud for further analysis to generate more valuable information such as energy consumption prediction during a period. The processed data and generated information can be stored permanently in cloud storage for long-running batch-processing tasks which may need to be conducted later. The IoT BDA application for energy management in smart homes is the real-world example used throughout this article.

3. AutoFog Architecture

As mentioned, fog computing adds an intermediate stratum between cloud and end-devices resulting in participation of more heterogeneous and fine-grained resources. In general, AutoFog architecture detailing in Figure 3 is composed of modules distributed into 4 layers: design, orchestration, elasticity, and infrastructure.

3.1. Design Layer. The design layer allows users to abstract and generalize complex distributed applications into application models using concepts defined by the framework. At heart of this layer is AutoFog DSL, a domain-specific language evolving from [4], which supports the description of hardware and software components of the application model and its fine-grained resources arranged hierarchically. In this language, the abstraction of a software component is called a *software type*. Similar software types can be generated from a software type template. These templates are stored in a software type catalog of the design layer. A software type is instantiated into *software instance* which is the running version of this type. Software instance inherits all the parameters and default values of its software type.

AutoFog DSL also proposes terms of *container type* and *container instance*. A container type represents a physical or virtual hardware component/device hosting software instances at runtime. It is worth mentioning that a small

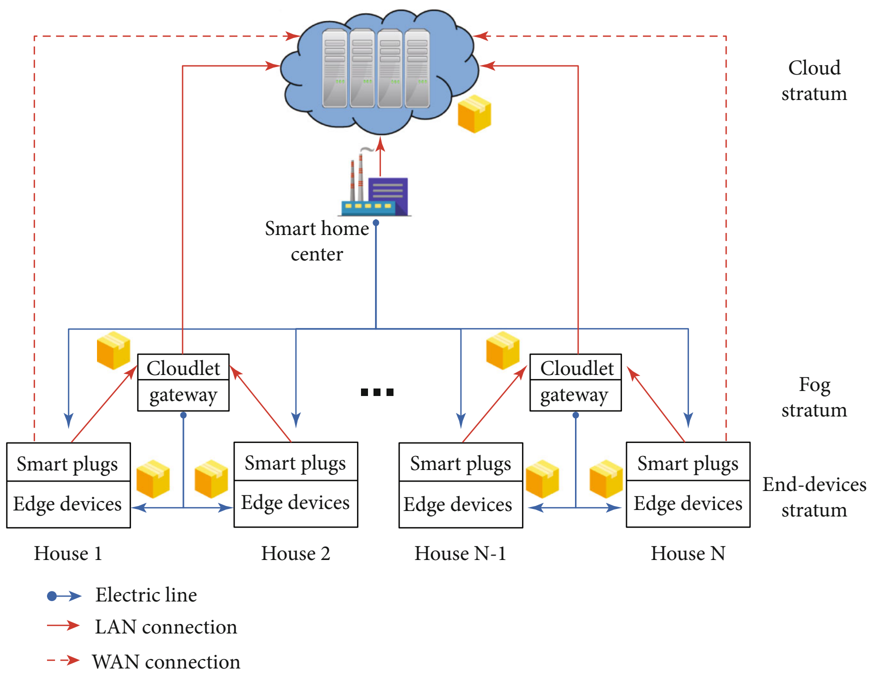


Figure 2: The IoT BDA use case for an energy management application.

end-device or a huge cloud data center can also be represented by a container type. Like software type templates, container type templates also can be stored in a container type catalog. Container instance is a container type in running state. The software instances instantiated from the same type can be distributed into multiple container instances of different container types. Moreover, software instances of different software types may collocate on the same container instance. Each software instance contains a reference to the container instance on which it is running.

Relations between software types (i.e., horizontal relationship) can be defined in the application model by series of "exported" and "imported" configuration variables. While an exported variable is a structure that a component exposes to remote components using it, an imported variable is a structure containing configuration information required by a component to initialize a service. Receiving the imported structure to boot is mandatory or not depending on specific software types.

Another kind of relation supported in AutoFog DSL is the parent-child relationship between a software type and its containers (i.e., vertical relationship). This relation in cloud applications is usually a simple map between the software components and their hosts (e.g., virtual machines). With fog-computing applications, it is often that a software component is deployed inside multiple levels of software and hardware containers. For example, a Tomcat war file is contained inside a Tomcat server, the Tomcat server, in turn, is packed in a Docker container, and a Docker container is hosted in a virtual machine of a cloudlet or a physical machine of a fog cluster. AutoFog DSL supports such a complex description to fulfill the gap when defining very heterogeneous resources of fog-computing applications.

In AutoFog, an application is a collection containing descriptions of container types, container instances, software types, software instances, and vertical/horizontal relations. An excerpt of the IoT BDA application model under AutoFog DSL is depicted in Figures 4 and 5. As shown in the figures, the means used for installation and configuration of the software instances in the container must be defined such as Bash, Chef, or Puppet. Corresponding to the selected technology, some scripting files defining operations on how to install and configure the software on the container may need to be provided along with the application model.

3.2. Orchestration Layer. Since a completed model of the fog-based application is sent to the orchestration layer, the model is parsed, and the life cycle of the application is managed and ensured by *Application Manager* (AM). It also checks the application's current state (not deployed, deploying, deployed and stopped, starting, deployed and started, stopping, etc.). Through this module, the running application can be updated by adding/removing types and instances to/from current application model. Application Manager has a global view of the application, all software components, and the links between them, but it does not intervene in the physical deployment of software components and containers. A copy of the current global view is sent to *Placement Manager* (PM) to compute a placement plan when the application is initialized or updated. Placement Manager supports various kinds of solvers such as constraint programming-based solvers, heuristics-based solvers, learning automata-based allocator, and metasolvers. The users need to select one of the supported solvers depending on what is more important to them, accuracy or performance. Another module in this layer is *Monitoring Manager* which is used to monitor states of container

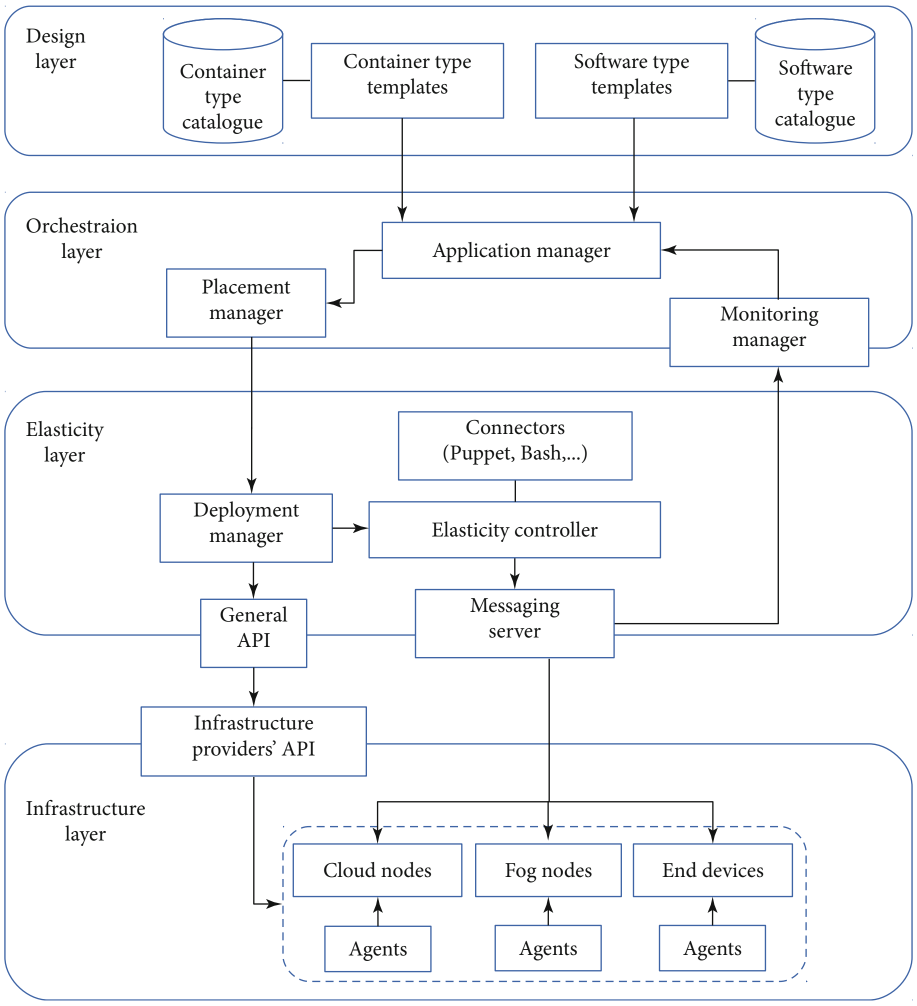


Figure 3: AutoFog architecture and interactions.

instances using heartbeats and notify the administrator that the container went down.

3.3. Elasticity Layer. The core components of this layer are *Deployment Manager* (DM) and *Elasticity Controller* (EC). They are modules coordinating the physical deployment of the application across containers. They must ensure all software instances are deployed with the correct configuration in hierarchical container instances. The DM instantiates containers through the provider's API, and the EC manages the deployment and scaling of the software types on the containers. Another mission of the DM is ensuring the federation between infrastructure providers. To do this, heterogeneous infrastructures from these providers must be abstracted. In the cloud, the DM must ensure that the software instances hosted in different containers belonging to various cloud providers work as in the same provider. To avoid vendor lock-in issues, some access lists may need to be added according to each provider's policy. In fog, this federation needs to be enforced not only among cloud providers but also between cloud and fog providers. Therefore, AutoFog provides a flexibly plug-in mechanism to add and abstract different cloud/fog providers. It provides a general AutoFog API with critical infrastructure primitives including the creation and deletion of software or container instances as well as minimal information about their states. These general requests will be translated and sent to specific cloud/fog providers' APIs, thanks to their corresponding infrastructure plug-ins. Thus AutoFog is completely independent of any fog/cloud infrastructure.

Right after a software instance is switched to running state, the EC maintains an *admin* topic on a *Messaging Server* to keep track of all components. The EC, therefore, plays an important role since it constitutes the entrance for both the initial configuration and the upcoming scale(s). Briefly, this module has the responsibility of elasticity control of the components it manages.

3.4. Infrastructure Layer. Installation and configuration of software at the Infrastructure layer are done by *AutoFog Agents*. An agent is a lightweight software installed in advance inside a container instance to manage the installation and elasticity of software instances of this container. Therefore, each agent only knows about the local components of its hosted container. In general, it is responsible for carrying out communication on

```
val iot_bda: application = { name="IOT_BDA",

  softwareTypeList= SOME
   (* Storm Cluster for Event Stream Processing *)
    [{name="Storm-Cluster",
      varList= SOME [{name="ip",value=""},
                     {name="port",value=""}],
      exportedStructList= NONE,
      importedStructList= SOME [{name="OpenHAB",
                                  channel= NONE,
                                  varName=["OpenHAB.ip","OpenHAB.port"],
                                  requiredToBoot=true}],
      hostingType="Cloudlet-Node", configurator="bash", children="Nimbus, Storm-Supervisor"},

    (* OpenHAB: Home Automation Bus *)
     {name="OpenHAB",
      varList= SOME [{name="ip",value=""},
                     {name="bindingChoice",value=""}],
      exportedStructList= SOME [{name="OpenHAB",
                                  channel= NONE,
                                  varName=["ip","bindingChoice"],
                                  requiredToBoot=true}],
      importedStructList= SOME [{name="SmartPlug",
                                  channel= NONE,
                                  varName=["SmartPlug.ip","SmartPlug.fogDomain"],
                                  requiredToBoot=true}],
      hostingType="Cloudlet-Node", configurator="puppet", children= NONE},

    (* Cassandra: Cloud Storage *)
     {name="Cassandra",
      varList= SOME [{name="ip",value=""},
                     {name="portCQL",value=""}],
      exportedStructList= SOME [{name="Cassandra",
                                  channel= NONE,
                                  varName=["ip","portCQL"],
                                  requiredToBoot=true}],
      importedStructList= SOME [{name="Storm-Supervisor",
                                  channel= NONE,
                                  varName=["Storm-Supervisor.ip","Storm-Supervisor.port"],
                                  requiredToBoot=true}],
      hostingType="VM-EC2", configurator="chef", children= NONE},
...

  containerTypeList= SOME
    (* EC2 VM - A Cloud Node *)
     [{name="VM-EC2",
      configurator= CLOUD_IAAS, vmTypeName="m4.xlarge", children="Cassandra, Storm-Cluster"},

    (* CLOUDLET NODE - A Fog Node *)
     {name="Cloudlet-Node",
      configurator= FOG_IAAS, fogTypeName="docker", children="OpenHAB, Storm-Cluster"}]
...
```

Figure 4: Types of IoT BDA application described by AutoFog DSL.

behalf of its container. A software instance is configured by the agent using the configuration connector specified in its corresponding software type. Additionally, the agents publish variables a software instance exports (i.e., exported vars) and variables this instance imports (i.e., imported vars) to corresponding topics in the messaging server. The agents communicate with each other and with the remaining AutoFog modules, thanks to communication channels in the messaging server.

A messaging protocol has also been implemented based on exchanging asynchronous messages and publishing/subscribing message topics which allows the upper layers to dynamically add/remove containers as well as software components to a running application. Using messaging services to exchange messages promotes interoperability between application components. All communication protocols used by fog hardware components need to be abstracted and converted to uniform messages supported by the message server. Currently, AutoFog implements RabbitMQ as its unique messaging service. Supports for other messaging brokers or services can be added to AutoFog as new plug-ins. In the following section, we describe how AutoFog manages its applications at runtime.

4. Dynamic Deployment and Elasticity Provision

In this section, we describe how AutoFog can be used to install and manage the IoT BDA application mentioned in

```
    softwareInstanceList= SOME                                                              1
     (* Nimbus instance *)                                                                  2
       [{name="storm-nimbus-1", softwareTypeName="Nimbus",                                  3
       varList= SOME [{name="ip", value=""},                                                4
                      {name="port", value="6627"}],                                         5
       exportedStructList= SOME [{name="Storm-Nimbus",                                      6
                                  channel= NONE,                                            7
                                  varName=["ip","port"],                                    8
                                  requiredToBoot=true}],                                    9
       importedStructList= SOME [{name="Storm-Supervisor",                                  10
                                  channel= NONE,                                            11
                                  varName=["Storm-Supervisor.ip","Storm-Supervisor.port"],  12
                                  requiredToBoot=true}],                                    13
       configurator= INHERITED, hostingInstanceName= NONE, state= NONE},                   14
                                                                                            15
      (* Cassandra instance *)                                                              16
       {name="cassandra-1", softwareTypeName="Cassandra",                                   17
       varList= SOME [{name="ip", value=""},                                                18
                      {name="portCQL", value="9042"}],                                      19
       exportedStructList= INHERITED,                                                       20
       importedStructList= INHERITED,                                                       21
       configurator= INHERITED, hostingInstanceName= NONE, state= NONE},                   22
                                                                                            23
     (* OpenHAB instance *)                                                                 24
       {name="openhab-1", softwareTypeName="OpenHAB",                                       25
       varList= SOME [{name="ip", value=""},                                                26
                      {name="bindingChoice", value="SmartPlugBinding"}],                    27
       exportedStructList= INHERITED,                                                       28
       importedStructList= INHERITED,                                                       29
       configurator= INHERITED, hostingInstanceName= NONE, state= NONE}],                  30
...                                                                                         31
                                                                                            32
   containerInstanceList= AUTO []                                                           33
                                                                                            34
   autoProtection= SOME                                                                     35
     [{defaultRules= {                                                                      36
       rule="iptables -A INPUT -i lo -j ACCEPT",                                            37
       rule="iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT",       38
       rule="iptables -A INPUT -p icmp -j ACCEPT",                                          39
       rule="iptables -A INPUT -p tcp --dport ssh -j ACCEPT",                               40
       rule="iptables -P FORWARD DROP",                                                     41
       rule="iptables -P INPUT DROP"}}],                                                    42
                                                                                            43
   autoRepair= NONE []                                                                      44
...                                                                                         45
```

Figure 5: Instances of IoT BDA application described by AutoFog DSL.

section 2. We depict in Figure 6 various steps required to deploy the IoT BDA application and conduct elasticity on the Fog using AutoFog. Below are details of these steps:

(1) The model of the fog application is sent to AutoFog to be deployed. Figures 4 and 5 represent the IoT BDA application model under AutoFog DSL. Figure 4 shows the different software and container types of the application. As depicted, we describe some software and container types such as Storm cluster [5] (Figure 4, lines 5-13), Cassandra [6] (Figure 4, lines 30-41), and OpenHAB [7] (Figure 4, lines 16-27). The model first is parsed by the AM module to generate a provider-independent model (PIM). The PIM then goes through the constraint-problem solvers of the PM module to generate a provider-specific model (PSM) describing which containers belonging to which infrastructure of platform providers will host the software components

(2) The PSM is sent to the DM module at the beginning of this step. The DM, through the local General API, contacts the infrastructure/platform API to ask for the instantiation of container instances. For example, in the case of the IoT BDA application, we initially deploy one Nimbus instance (Figure 5, lines 3-14), one Cassandra instance (Figure 5, lines 17-22), and one OpenHAB instance (Figure 5, lines 25-30). These instances are deployed on container types named “Cloudlet-node” (Docker [8]) on the fog or “VM-EC2” on the cloud. The users can either specify explicitly the name of a hosting container instance for a specific software instance or leave this task for the solvers. At the end of the step, the infrastructure/platform providers instantiate the container instances

(3) Each container instance has a message queue in the messaging server. The DM asks the EC to include software instance definition and corresponding scripting

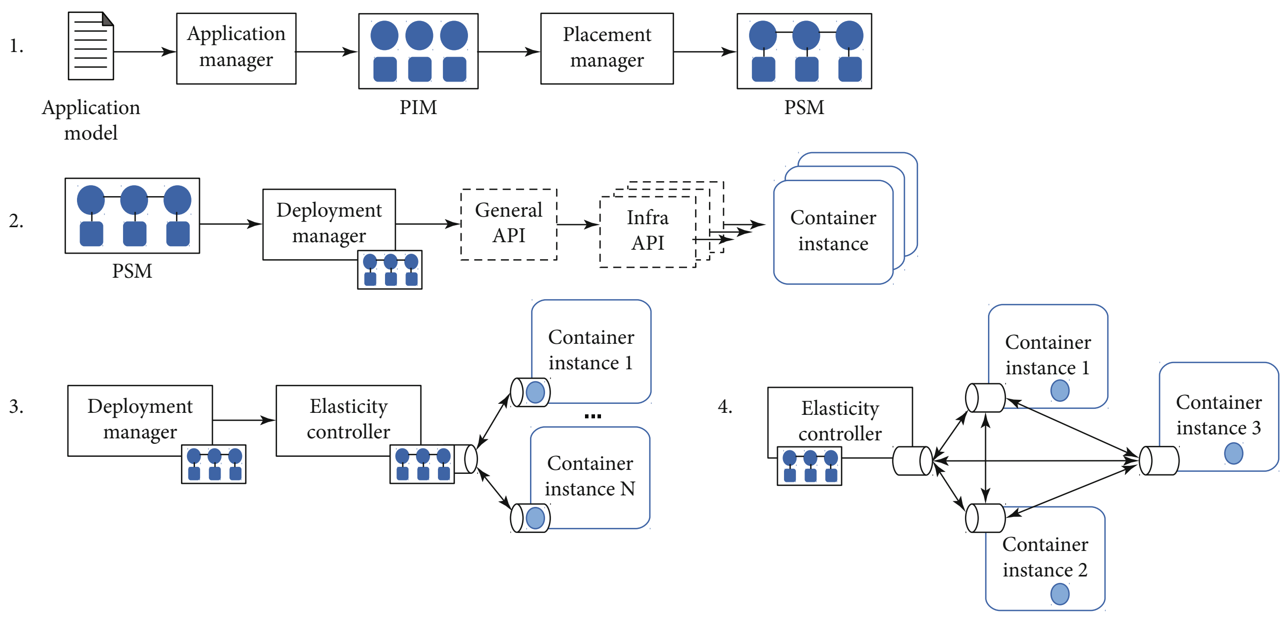


Figure 6: Steps of the AutoFog mechanism of dynamic deployment and elasticity provision.

files onto the message queues including the EC queue. When containers are started up and running, they receive information of instances that they are in charge of from the messaging server and start to install them

(4) All software and container instances of the application take part in the application-wide configuration after being installed. The imported and exported variables are exchanged, and when an event triggers elasticity actions, they are scaled out/in automatically. This ensures the correctness of the elasticity mechanism.

From now on, container instances are autonomous and independent from other modules of AutoFog. They can exchange information with each other, thanks to corresponding topics in the messaging server. This decentralized approach allows the application to work even when errors occur on the EC.

5. Evaluation

We implemented a prototype of AutoFog framework and conducted several experiments validating its functionality in terms of auto deployment and elasticity. The Smart Home BDA application in section 2 is chosen as the deployment's target of the framework.

5.1. Experiment Setup. The modules of AutoFog and software components of the smart home BDA application were developed to be packaged easily into Docker containers. These containers are orchestrated by the Kubernetes cluster [9] implemented in our homegrown infrastructure at the VNU University of Engineering and Technology (VNU-UET). Each Kubernetes pod is configured to contain only one container. The experimental implementation is depicted in Figure 7. In our implementation, fog nodes are ensured by Kubernetes and Cloud nodes are provisioned by OpenStack [10]. Because Kubernetes can provide both elasticity function and fog node, it works at both elasticity and infrastructure layers of AutoFog architecture. We have also created different Docker images which are all embedded an AutoFog agent beforehand:

(i) AutoFog image contains main modules of AutoFog such as AM, PM, DM, and EC. This image is used for AutoFog nodes at both Fog and Cloud strata

(ii) Storage image contains an instance of Cassandra, a NoSQL distributed database management system. It supports handling large amounts of data across many nodes with a highly available service. Its data model allows incremental modifications of rows. This image is used to instantiate the storage nodes where permanent data of the smart home BDA application are stored at the cloud stratum

(iii) Storm Master/worker images represent for two types of Storm nodes: Nimbus master node and Supervisor worker node at fog strata. The master node manages cluster of Storm Supervisor nodes where Storm topology is submitted to execute. Storm topology is composed of Spouts who pump data to the topology and Bolts who consume and process the data in parallel from Spouts

(iv) OpenHAB image includes an OpenHAB message binding which gathers measurements from smart plugs and forwards them to the Storm cluster. OpenHAB nodes created from this image working as edge gateways locate at the border between end-device and fog strata

(v) Message server image contains a RabbitMQ server to asynchronously handle message queuing telemetry

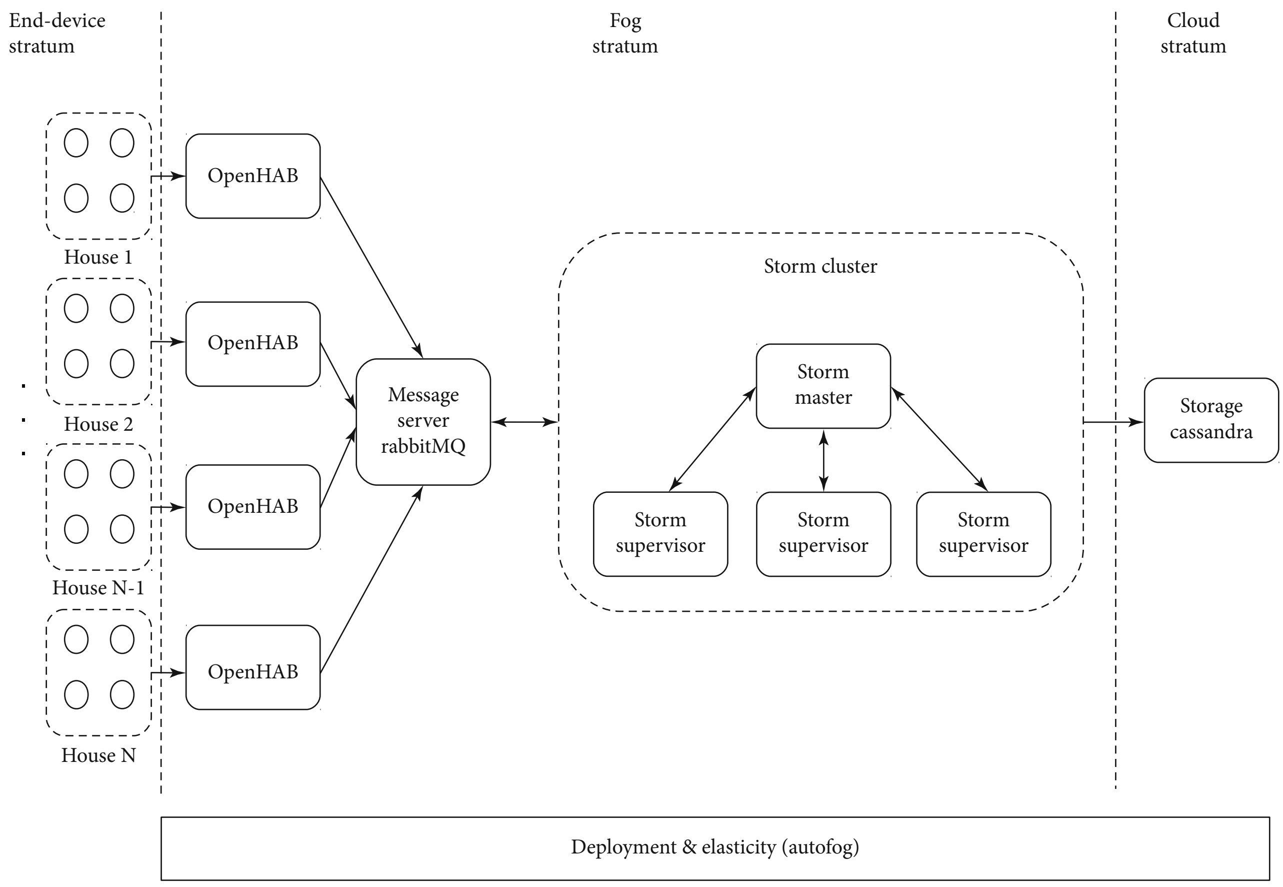


Figure 7: The experimental implementation of the smart home BDA application.

transport (MQTT) messages back and forth in the system. MQTT is a lightweight communication protocol broadly used for IoT applications [11]. This image is used for message server nodes at both fog and cloud strata.

In our experiments, the input data are synthesized from a practical data source provided by DEBS grand challenge 2014 [12]. The dataset contains over 4055 million of measurements for 2125 smart plugs deployed in multiple houses in Germany. The full data cover a period of one month in September 2013. We have developed an end-device image including a program which regenerates measurements retrieved from the DEBS dataset.

All the practical deployment times are calculated over 20 different runs to get the mean. A new container is needed for each software instance. The time in Table 1 covers the instantiation of the container, the initial configuration of the software until they reach states from which they can be started. For example, Storm Master (Nimbus) is the one whose instances take the longest time to deploy with ≈248 seconds on average.

The initial deployment contains one Cassandra storage node at the cloud stratum, one Storm Master node and one to two Storm Supervisor nodes working at fog stratum, two OpenHAB nodes working as edge gateways, and a maximum of 40 end-device nodes. Modules of AutoFog and its message server are grouped into one node called AutoFog node. The selection of these software components is just one of many specific combinations of IoT BDA applications. Other combinations can also be used in the experiments without losing the generality and validity of the AutoFog architecture.

Table 1: Deployment time of five smart home BDA's instances over 20 runs.

Software types	Mean	99th percentile
Storm Master	248.05 ± 6.4	221.16 ± 9.4
Storm Supervisor	112.02 ± 16.1	125.53 ± 12.3
OpenHAB	145.43 ± 8.9	154.21 ± 9.7
RabbitMQ	112.71 ± 14.8	118.04 ± 11.2
Cassandra	103.88 ± 11.6	98.09 ± 8.5

5.2. Storm Topology. Storm is one of components of the smart home BDA application. Thus, Storm can be described by AutoFog DSL at the design layer and deployed and managed by submodules of both orchestration and elasticity layers. Storm topology to process the DEBS IoT data is shown in Figure 8. The topology is composed of 5 components as follows.

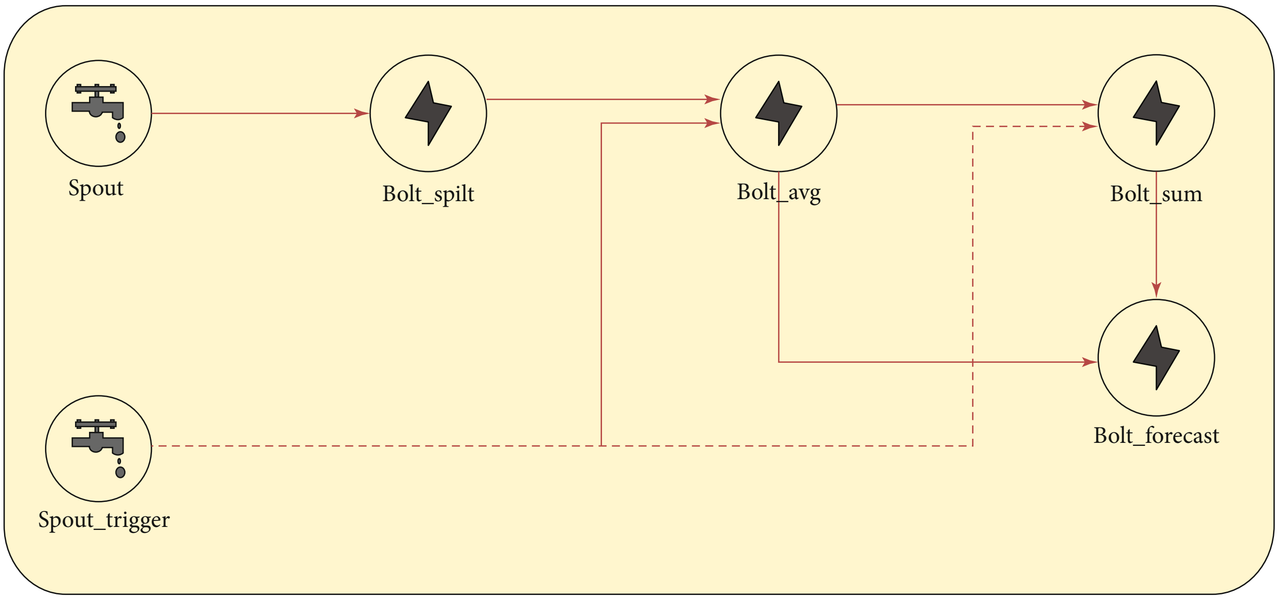


FIGURE 8: Storm topology of the smart home BDA application.

(i) Spout_data: it has the function to create MQTT clients that connect to the message broker, subscribe to predetermined topics, receive data from the broker, separate the data into meaningful fields, and then send them to the back Bolts for processing

(ii) Bolt_split: it has the function to read data sent from Spout_data, read the timestamp field, divide the data into time slices with predetermined window sizes according to the system's needs (1 minute, 5 minutes, to 120 minutes), and then send it to Bolt_avg for further processing

(iii) Bolt_avg: its function is to receive data from Bolt_split to calculate the average amount of electricity that the device uses in time slice with predetermined window sizes according to the needs of the system (1 minute, 5 minutes, to 120 minutes). The data after calculating will be saved to RAM memory and then sent to Bolt_sum for further processing. In addition, Bolt_avg stores the average data of the energy consumed by each device in the local database. The data stored on the database will be released to save memory to ensure the long-term operation of the system

(iv) Bolt_sum: it has the function to add the total energy used of all the equipment in the house to calculate the total amount of electricity consumed by that house in the time slice with predetermined window sizes according to the needs of the system. Similar to Bolt_avg, Bolt_sum stores the average data of the energy consumed by each house in a local database. The data stored on the database will also be released to save memory to ensure the long-term operation of the system

(v) Bolt_forecast: it has the function that uses data from previous time slice to predict energy usage value of next two time slices and then save it to database.

To vary IoT workload to the Storm topology, in the Bolt_forecast, we implement three prediction models making short-term electric load forecast of smart IoT devices. The utilization of these models causes differences in the amount of input tuples used in Storm's Bolts, especially in Bolt_avg, Bolt_sum, and Bolt_forecast. In the first model, time is divided into time slices, and the load average of any future time slice is predicted based on the average electric load of the previous ith time slices having the same timeframe of all preceding days. Assuming we predict the average load of the second time slice $P(ts_{i+2})$ from the current slide ts_i, the formula used is

$$P(ts_{i+2}) = \frac{\text{avgLoad}(ts_i) + \text{median}\left(\text{avgLoad}(ts_j)\right)}{2}. \quad (1)$$

In formula (1): $\text{avgLoad}(ts_i)$ is the average load of current time slice ts_i, $\text{avgLoad}(ts_j)$ is the average load of time slices ts_j—time slices of previous days have the same timeframe as slice ts_i—and $\text{median}(\text{avgLoad}(ts_j))$ is the median of all previous time slices ts_j.

With the second model, $\text{avgLoad}(ts_j)$ is the average load of all previous time slices in the same day up to the current time slice. For the third one, $\text{avgLoad}(ts_j)$ is average load of time slices of previous weeks having the same timeframe as slice ts_i on the same date.

5.3. Result. We publish messages from the smart home dataset to the Storm topology of the smart home IoT application. In Bolt_forecast, we change the prediction models, and results measured on Storm Nimbus Master node representing many experimental runs are shown in Figure 9. The time on the x-axis is the execution time for experimental system to publish all messages from the DEBS data files and process these messages through Bolts of the Storm topology. The

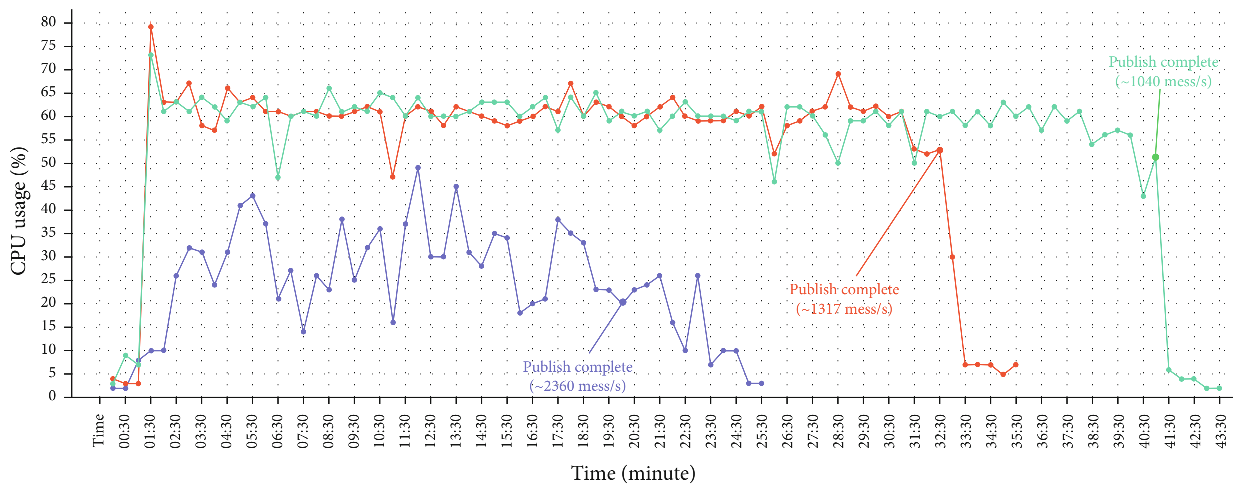


FIGURE 9: CPU usage over time of processing smart home dataset with three prediction models. The green, red, and blue lines represent the first, second, and third prediction models.

green, red, and blue lines represent the first, second, and third prediction models, respectively. The execution time of the blue one lasts about 20 minutes with the average CPU usage is quite different from time to time, averaging approximately 30%. The average throughput at the end of data processing is about 2306 messages per second. As for the green and red ones, the CPU usage is roughly the same, maintaining at 65-70%, and at a time spike can be as high as 75-80%. The time to publish and process data with the first model is about 41 minutes, greater than the second one which is about 32.5 minutes. The average throughput at the end of data processing of the first and second models is about 1040 and 1317 messages per second, respectively. The main reason for the difference in execution time is that the prediction models use different amounts of historical data leading to various computation times in the Bolts. The predicting results of all three prediction models are depicted in Figure 10.

We see in Figure 9 that both the first and second models exhibit the average CPU usages higher than 75%. Therefore, elasticity strategies can be applied to reduce the average CPU usage and at the same time shorten the execution time. To perform elasticity, two techniques can be implemented on Kubernetes: Vertical Pod Autoscaler and Horizontal Pod Autoscaler.

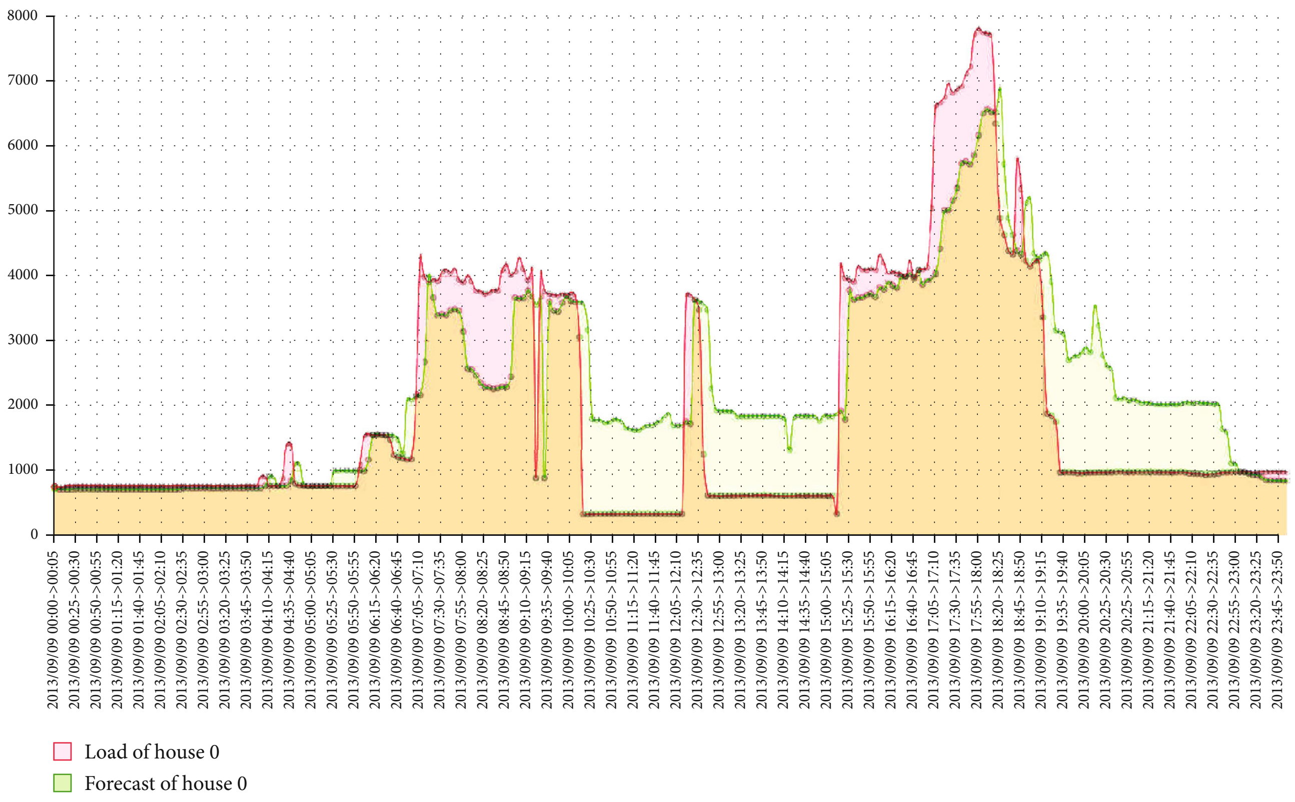
Horizontal Pod Autoscaler (HPA) is a technique to automatically increase or decrease the number of Kubernetes pods by collecting and evaluating CPU usage metrics from the Kubernetes Metrics Server. The number of pods will be in the range min and max which are set when generating HPA. The HPA is implemented as a Kubernetes API resource and as a controller. Every 15 seconds, the controller periodically checks and adjusts the number of pods so that the observed average CPU usage matches the value specified by the user. HPA calculates the number of pods based on a formula where Ceil() is the rounding function:

$$\#\text{RequiredPods} = \text{Ceil}\left(\#\text{CurrentPods} \times \frac{\text{PresentValue}}{\text{ExpectedValue}}\right). \quad (2)$$

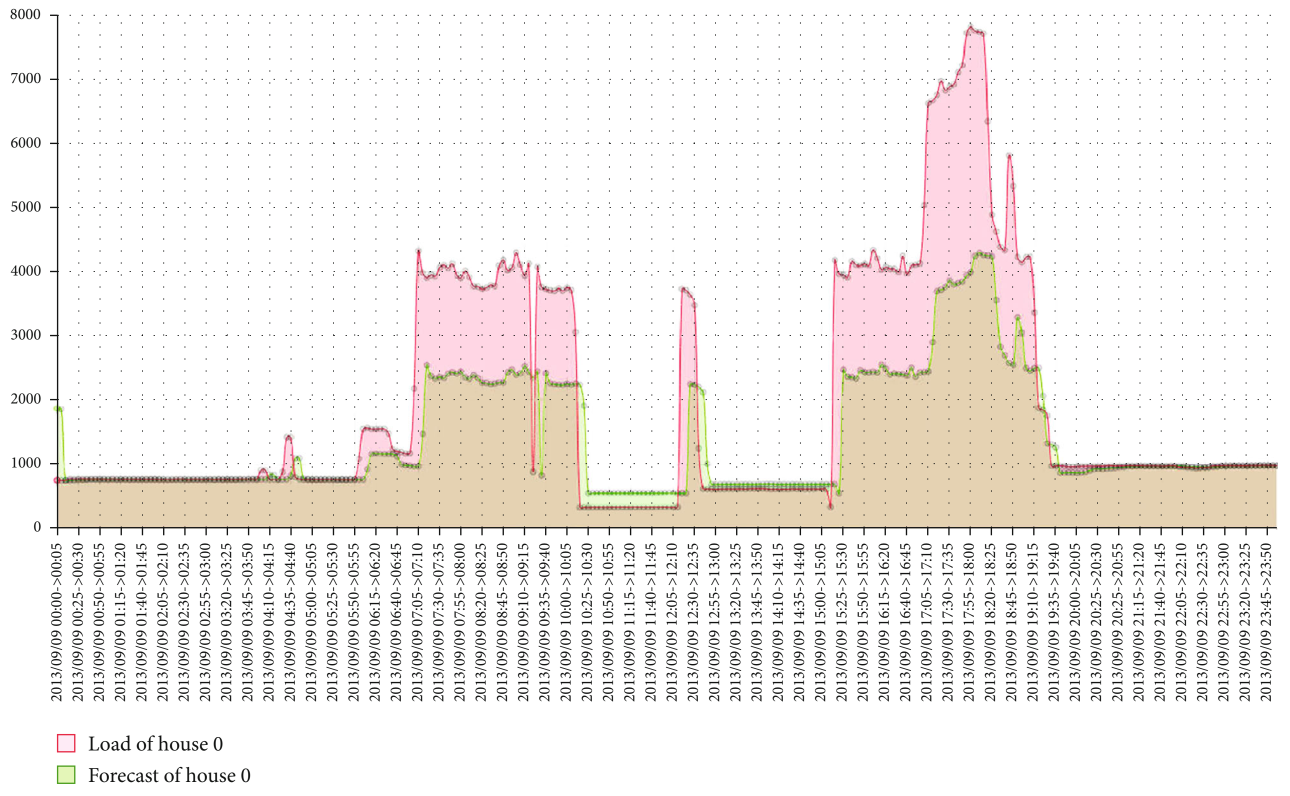
Vertical Pod Autoscaler (VPA) is a technique that automatically increases and decreases resources such as CPU and memory for pods depending on the needs of the pods. Technically, VPA does not dynamically change resources for existing pods; instead, it checks the managed pods to see if the resources are set correctly and, if incorrectly, removes them so that the controller can create other pods with updated configurations.

5.3.1. Horizontal Elasticity. Without loss of generality, we conduct HPA with the first model only. A HPA object for the deployment and the pod "StormWorker" are built with the following limits: When the average CPU usage greater than 75% will trigger an auto scale up increasing number of StormWorker pods, it will do a scale down to decrease the number of pods. When the used RAM memory over 3 GB (75%) will perform auto scale up, it will do a scale down. The minimum and maximum numbers of pods are 1 and 5, respectively. Elasticity results are shown in Figure 11. The horizontal scaling mechanism responds very quickly and works quite smoothly to changes in pod's CPU resource usage even when resource usage spikes during very small amount of time. This mechanism does not cause downtime of the Storm workers during use. We see that the average CPU usage and execution time are reduced to 40% and 36.5 minutes in the case of using elasticity comparing to 75% and 41 minutes in the case of not using elasticity.

5.3.2. Vertical Elasticity. By default, for stability, the VPA will not perform an automatic update of pod resources if the number of pod copies is less than 2. So two pods are created with each pod configuration as following: the minimum resource is 0.5 CPU core and 1 GB RAM; the maximum resource is 1 CPU core and 2 GB RAM. Next, a VPA object is created for pods with *updateMode=Auto*. After the pods are created, the IoT load is injected to the two pods with input messages from the DEBS dataset. The obtained results are depicted in Figure 12. Each line with a specific color is a representation of a pod containing the running container of

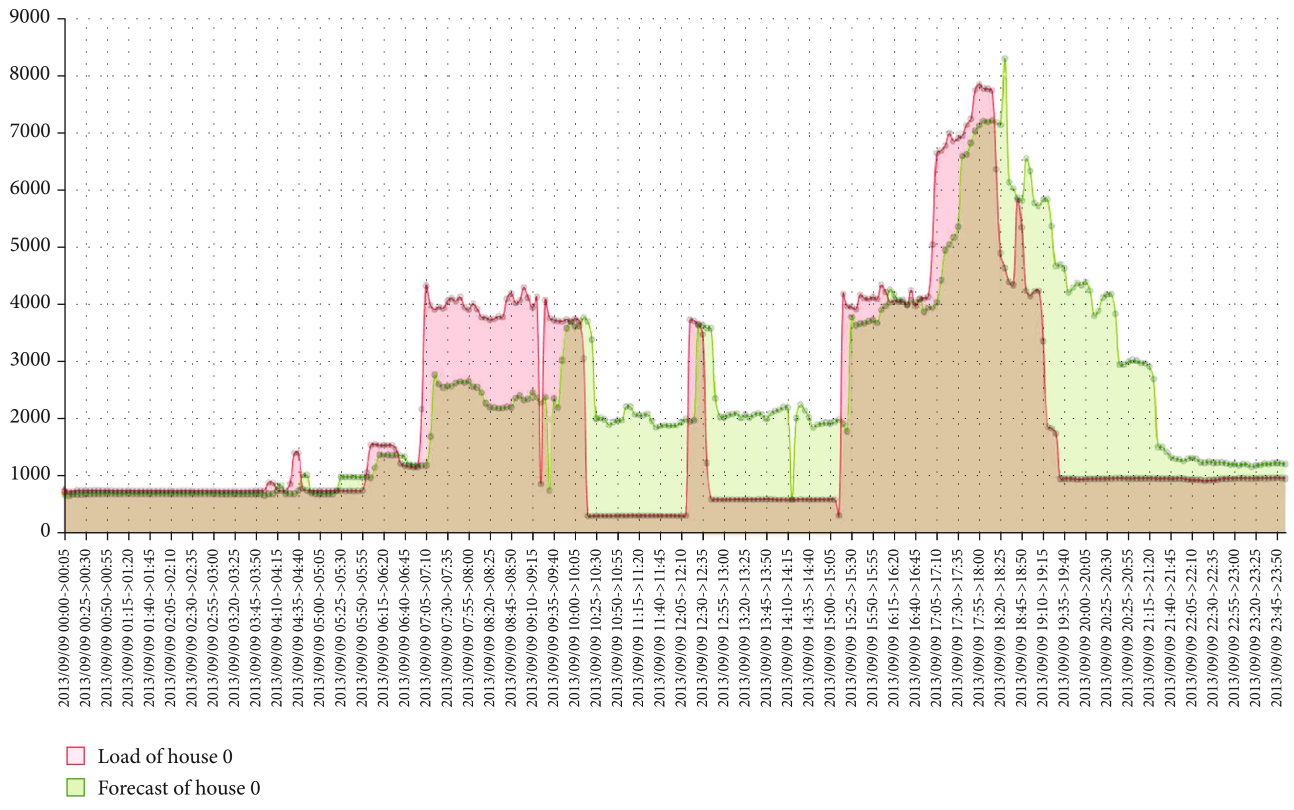


(a)



(b)

Figure 10: Continued.



(c)

Figure 10: Results of all three prediction models. (a)The first model. (b)The second model. (c)The third model.

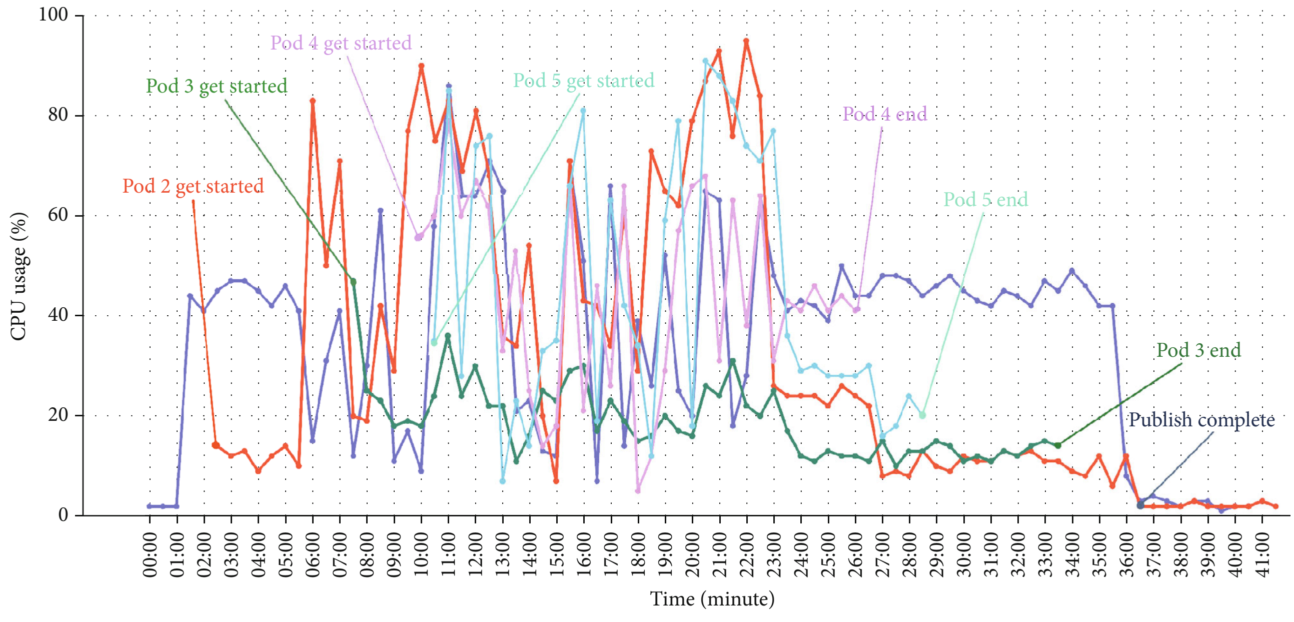


Figure 11: CPU resources are automatically scaled horizontally; 5 lines represent 5 StormWorker pods.

Storm worker. After vertical elasticity actions, two pods (orange, blue) with minimum configuration are replaced by two new pods with maximum configuration (green, red), respectively. It is obvious that the average CPU usage is reduced significantly.

At the moment, Kubernetes did not have a mechanism for updating resources directly on a running pod; replicating pod is the only way. Therefore, VPA can probably bring unexpected downtime for the Storm worker. When VPA creates new pods and causes downtime in about 30 seconds,

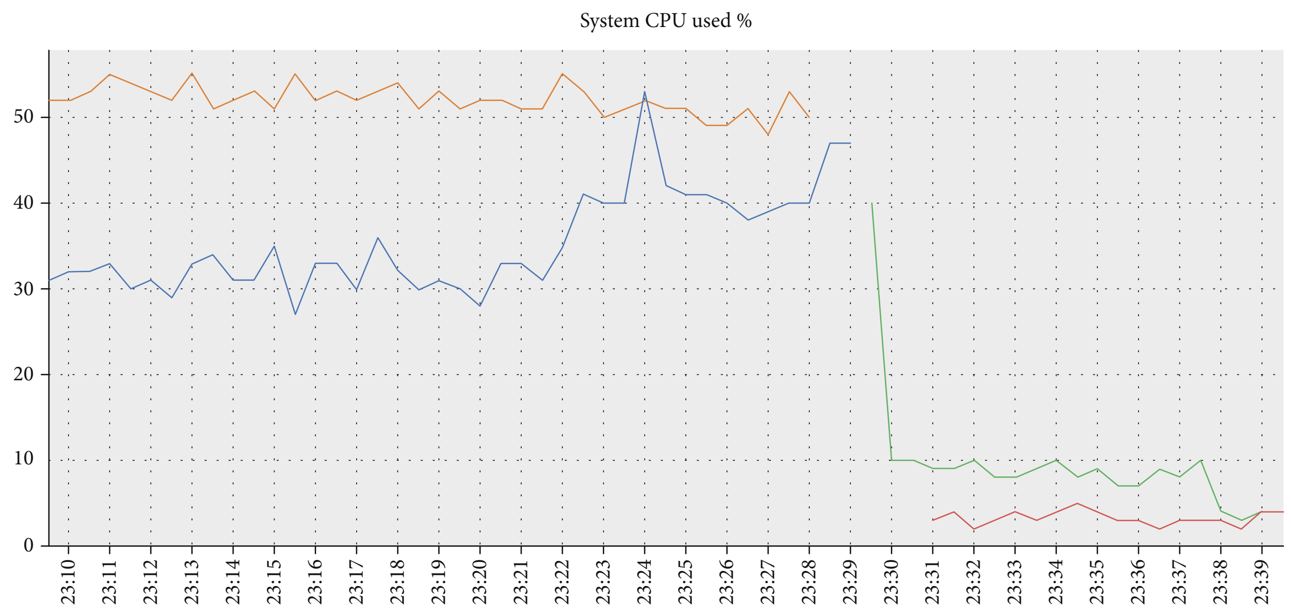


FIGURE 12: Real-time average CPU usage when using VPA.

there is a certain amount of messages lost during that time that depends on the speed of publishing. After that, the succeeding connections could be functioning normal.

6. Related Work

6.1. General Frameworks for Fog-Based IoT BDA Applications. For almost a decade since the introduction of fog computing, many frameworks have been being proposed to support fog-based IoT BDA applications. Almost all frameworks own one [13–18] or multiple fog orchestrators (FO) [19–21] operating at the orchestration layer. With the former, FO must have holistic view of fog resources and connect to all fog nodes in the framework. Multiple FO can resolve the scalability issue of the single one but might incur some overhead from communication between these FO.

Chen et al. propose a FA2ST (fog-as-a-service technology) fog framework supporting any kind of IoT application [14]. On-demand discovery of fog service is provided to figure out if a connected fog node's resource is currently available when an IoT request comes. In another research, an IoV-fog infrastructure is defined to provide supports to overworked RSUs of UAVs [16]. Such a RSU can trigger a deployment of UAV, and data is migrated to this UAV to decrease response latency and increase the IoV computation. Storm, a stream processing platform, is extended by Cardellini et al. to enable a distributed IoT resource scheduler which is latency aware [13]. Fog nodes in this extension have knowledge of resource availability of each other and thus ensures QoS of IoT service distribution. Donassolo et al. propose FITOR, a Fog-IoT ORchestrator which monitors the fog infrastructure and keeps track of every fog resources anytime [15]. It helps to deploy the IoT data to fog nodes automatically. Foggy framework introduced by Yigitoglu et al. allows the deployment of IoT task requests to an appropriate fog node having available resources and satisfying several QoS requirements such as priority, latency, and privacy [18]. In the same vein, Foggy FOC uses MQTT protocol to monitor all fog resources [17]. To increase future deployment, it has a mechanism to store historical IoT workloads and requirements.

To increase security and reliability, Fogbus [20], a scalable fog framework, partitions fog nodes into various roles including computing, gateway, repository, and broker nodes. A defective fog node can be restored by repository nodes and taken over by other fog nodes. A blockchain solution is applied to validate dependability of IoT data sources. Fog nodes are clustered into colonies in research of Skarlat et al. [19]. In their fog architecture, FO of each colony keeps all fog available resource information. IoT requests firstly are allocated to fog nodes in a colony. If the colony does not have enough resources, the FO will find another colony to fulfill the tasks through transferring the requests using REST API. It also can propagate the requests to the cloud stratum if appropriate. Data migration between fog nodes and RSUs in an IoV-fog application is considered by Zhang et al. [21]. Multiple fog nodes in a region are grouped into a cluster and managed by a coordinator (FO). If a vehicle moves to a new region, an IoT module may be handed over to another fog cluster to avoid interrupted IoT processing.

Although these frameworks are aimed at satisfying deploying and provisioning fog resources using one or multiple FO, they do not take elasticity feature into account as in our research.

6.2. Elasticity Frameworks for Fog-Based IoT BDA Applications. Although a large number of frameworks are proposed for fog-based IoT BDA applications, not many studies consider elasticity for this kind of application. Mobile fog [22] proposes a scaling mechanism where overloaded workloads are resolved by fog nodes created dynamically. It also properly distributes

IoT data to these new fog nodes. Moreover, its API data migration is suitable for ambulant IoT devices like smart phones, cameras, and vehicles. To enable elasticity for IoT data stream processing applications using container, Wu et al. modify Kubernetes HPA to adapt at runtime the deployment of containerized BDA applications to the estimated load arrival rate [23]. In a similar way, Netto et al. scale Docker containers in Kubernetes using a state machine approach [24]. Adaptive AI services run on IoT gateways and fostered on the cloud are enabled by Elastic-IoT-Fog (EiF), a flexible fog-computing framework [25]. EiF virtualizes an IoT service layer platform and orchestrates various fog nodes. The feasibility of elasticity feature in EiF is depicted via an example of intelligent traffic flow management and monitoring, in which network slicing units and respective resource elasticity are dynamic provisioned. Zanni et al. present and report the evaluation of a system consisting of virtual services in a combined fog, cloud, and IoT environments with various device settings [26]. By using geometric monitoring, the paper proposes an original solution to dynamically scale and provision the resources for the fog-computing layer. Elasticity is expressed in aspect of moving and redeploying more mobile components to the fog nodes closest to the targeted end-devices. Wang et al. design a three-tier edge computing system architecture to dynamically route data to proper edge servers and elastically adjust their computing capacity for the real-time urban surveillance applications [27]. Moreover, the paper also introduces schemes of workload balance and resource redistribution in emergency situations. The EU ELASTIC project is aimed at developing a software architecture for extreme-scale BDA in fog-computing ecosystems [28]. With the architecture, ELASTIC supports elasticity across the fog compute strata while fulfilling communication, real-time, energy, and secure properties.

The above-mentioned studies and solutions bring elasticity feature for the resources of IoT BDA applications on fog-computing environment but do not mention the automatic deployment of these applications based on the description of given software/hardware components and deployment plans as the function provided by AutoFog.

7. Conclusion

We have presented AutoFog, a framework with a four-layer architecture, which supports transformation of IoT BDA applications to elastic fog-based ones and automatic deployment of these applications on fog environment. A mechanism of elasticity provision is integrated into the framework to enable adaptation to changes of workload from IoT smart devices. The transformation is more smooth and less time-consuming through the reuse and extension of an existing domain-specific language and off-the-shelf components. The validating experiments with the practical smart home use case were conducted with Kubernetes for fog nodes and OpenStack for cloud nodes. The results show that the implementation of AutoFog framework accompanied by our proposed elasticity mechanism is more flexible and faster when there was fluctuations in managed resources.

Data Availability

The CSV data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

Acknowledgments

This research is funded by Graduate University of Science and Technology under grant number GUST.STS.TĐ2019-TT02.

References

[1] P. Mell and T. Grance, "The NIST definition of cloud computing (draft)," *NIST Special Publication*, vol. 800, no. 2011, p. 145, 2011.

[2] A. Brogi, J. Soldani, and P. W. Wang, "TOSCA in a nutshell: promises and perspectives," in *In Service-Oriented and Cloud Computing*, M. Villari, W. Zimmermann, and K.-K. Lau, Eds., pp. 171–186, Springer, Berlin Heidelberg, Berlin, Heidelberg, 2014.

[3] A. Sintef, "Cloud application modelling and execution language (CAMEL) and the PaaSageWorkflow," *In European Conference on Service-Oriented and Cloud Computing*, vol. 567, pp. , 2015437–439, 2015.

[4] L. M. Pham and T. Pham, "Autonomic fine-grained migration and replication of component-based applications across multi-clouds," *In 2015 2nd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS)*, pp. , 20155–10, 2015.

[5] A. Storm, https://storm.apache.org/releases/2.2.0/index.html. Accessed:2021-04-19.

[6] A. Cassandra, https://cassandra.apache.org/doc/latest/ .Accessed:2021-04-19.

[7] OpenHAB, https://www.openhab.org/docs/. Accessed:2021-04-19.

[8] Docker container, https://docs.docker.com/.Accessed:2021-04-19.

[9] Kubernetes website, https://kubernetes.io/.Accessed:2021-04-19.

[10] OpenStack, "Open source cloud computing infrastructure," https://www.openstack.org/.Accessed:2021-04-19.

[11] MQTT, https://mqtt.org/mqtt-specification/.Accessed:2021-04-19.

[12] DEBS, "Grand challenge: smart homes," 2014, https://debs .org/grand-challenges/2014/.Accessed:2021-04-19.

[13] V. Cardellini, V. Grassi, F. L. Presti, and M. Nardelli, "On QoS-aware scheduling of data stream applications over fog computing infrastructures," *2015 IEEE Symposium on Computers and Communication (ISCC)*, pp. 271–276, 2015.

[14] N. Chen, Y. Yang, T. Zhang, M. Zhou, X. Luo, and J. K. Zao, "Fog as a service technology," *IEEE Communications Magazine*, vol. 56, no. 11, pp. 95–101, 2018.

[15] B. Donassolo, I. Fajjari, A. Legrand, and P. Mertikopoulos, "Fog based framework for IoT service provisioning," *2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC).*, pp. 1–6, 2019.

[16] N. Madan, A. W. Malik, A. U. Rahman, and S. D. Ravana, "On-demand resource provisioning for vehicular networks using flying fog," *Vehicular Communications*, vol. 25, no. 2020, p. 100252, 2020.

[17] D. Soni and A. Makwana, "A survey on MQTT: a protocol of internet of things(IOT)," *In International Conference on Telecommunication, Power Analysis and Computing Techniques (ICTPACT)*, 2017.

[18] E. Yigitoglu, M. Mohamed, L. Liu, and H. Ludwig, "Foggy: a framework for continuous automated IoT application deployment in fog computing," *In 2017 IEEE International Conference on AI Mobile Services (AIMS)*, pp. , 201738–45, 2017.

[19] O. Skarlat, S. Schulte, M. Borkowski, and P. Leitner, "Resource provisioning for IoT services in the fog," *In 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. , 201632–39, 2016.

[20] S. Tuli, R. Mahmud, S. Tuli, and R. Buyya, "FogBus: a blockchain-based lightweight framework for edge and fog computing," *Journal of Systems and Software*, vol. 154, no. 2019, pp. 22–36, 2019.

[21] W. Zhang, Z. Zhang, and H. Chao, "Cooperative fog computing for dealing with big data in the internet of vehicles: architecture and hierarchical resource management," *IEEE Communications Magazine*, vol. 55, no. 12, pp. 60–67, 2017.

[22] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: a programming model for large-scale applications on the Internet of Things. In Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing (Hong Kong, China) (MCC'13)," *Association for Computing Machinery*, 2013, pp. 15–20, New York, NY, USA, 2013.

[23] Y. Wu, R. Rao, P. Hong, and J. Ma, "FAS: a flow aware scaling mechanism for stream processing platform service based on LMS," *In Proceedings of the 2017 International Conference on Management Engineering, Software Engineering and Service Sciences (Wuhan, China) (ICMSS'17). Association for Computing Machinery*, 2017, pp. 280–284, New York, NY, USA, 2017.

[24] H. V. Netto, A. F. Luiz, M. Correia, L. de Oliveira Rech, and C. P. Oliveira, "Koordinator: a service approach for replicating Docker containers in Kubernetes," *In 2018 IEEE Symposium on Computers and Communications (ISCC). 00058–00063*, 2018.

[25] J. An, W. Li, F. L. Gall et al., "EiF: toward an elastic IoT fog framework for AI services," *IEEE Communications Magazine*, vol. 57, no. 5, pp. 28–33, 2019.

[26] A. Zanni, F. Stefan, U. Jennehag, and P. Bellavista, "Elastic provisioning of Internet of Things services using fog computing: an experience report," *2018 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pp. , 201817–22, 2018.

[27] J. Wang, J. Pan, and F. Esposito, "Elastic urban video surveillance system using edge computing. In Proceedings of the Workshop on Smart Internet of Things (San Jose, California) (SmartIoT'17)," , New York, NY, USA, Association for Computing Machinery, 2017.

[28] EU ELASTIC website, https://elastic-project.eu/about/objectives.Accessed:2021-04-19.