Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

IoT-Pi: A Machine Learning-based Lightweight Framework for Cost-effective Distributed Computing using IoT

Tianchen Shao[1] | Deepraj Chowdhury[2] | Sukhpal Singh Gill*[1] | Rajkumar Buyya[3]

[1]School of Electronic Engineering and Computer Science, Queen Mary University of London, London, UK
[2]Department of Electronics and Communication Engineering, International Institute of Information Technology (IIIT), Naya Raipur, Chattisgarh, India
[3]Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia

Correspondence
*Sukhpal Singh Gill, School of Electronic Engineering and Computer Science,Queen Mary University of London, Mile End Road, Bethnal Green, London E14NS, UK
Email: s.s.gill@qmul.ac.uk

Summary

It is possible to develop intelligent and self-adaptive application on the edge nodes with rapid increase in computational capability of Internet of Things (IoT) devices. With the rapid growth of cloud technologies, the demand for hybrid architecture with cloud and IoT has also been boosted as well. To satisfy the critical and comprehensive requirements in the architecture evolution, we proposed a lightweight framework called IoT-Pi to provide a 3-phase (sample, learn, adapt) life cycle management of cloud resources with machine learning prediction working on IoT edge nodes using Raspberry Pi device. Compared to the traditional interference by human beings in the field of system administration, the accuracy rate of machine learning prediction in the proposed technique for some algorithms reached over 70%, which demonstrates the feasibility and effectiveness of running cloud resource management on an IoT devices such as Raspberry Pi.

KEYWORDS:
IoT, Machine Learning, Cloud Computing, Distributed Computing

1 | INTRODUCTION

Internet of Things (IoT) technologies are getting emerged with many micro-size chipset with powerful computational capacity [1]. These chipsets can be deployed to more critical environment that makes embed device more intelligent [1]. Now it is even supported to apply Raspberry Pi to more complex tasks requiring machine learning and more intelligent scenarios [1,2]. It is even verified that the device is capable of monitoring cloud resources [3]. The appearance of Raspberry Pi 4 is the integration of both powerful ARM chipset and embedded device. Massive computing nodes are working in parallel to provide powerful computational capacity, in comparison to traditional limited cluster or single node. It is also elastic to manage the lifecycle of all resources on the cloud. Many cloud service vendors provide dedicated Software Development Kit (SDK) to manage the cloud resources, which implement same purpose as Software Defined Network (SDN) [4] as well. However, cloud technologies have been dominated by a small group of companies with large volumes of computing resources. It is not affordable to run tasks based on Artificial Intelligence, when the business scenario is beyond the scale of the capacity owned by cloud service companies. However, the resources on the cloud are viable to be managed in real time [5]. Therefore, many independent developers and groups start to consume the cloud resources by applications in a more intelligent and economic approach. In this paper, we proposed a lightweight framework called IoT-Pi, which uses machine learning, IoT and cloud computing to solve this problem. The main contributions of this work are: 1) Verify the concept that IoT nodes undertake modern machine learning tasks, 2) Identify the idle state of cloud resource to manage the lifecycle adaptively, 3) Explore organizing cloud resources by machine learning techniques from IoT edge devices and 4) Minimize cost of resource usage when using cloud.

[0]**Abbreviations:** IoT, Internet of Things; AWS, Amazon Web Service; SVM, Support vector machine

TABLE 1 Comparison of IoT-Pi with existing works

Existing works	IoT(Raspberry Pi)	Cloud	Machine Learning
Pereira RI et al.[3]	✓	✓	
Saeed U et al.[6]	✓		✓
Cortez E et al.[7]		✓	✓
Bharadwaja V et al.[8]	✓		✓
IoT-Pi (this work)	✓	✓	✓

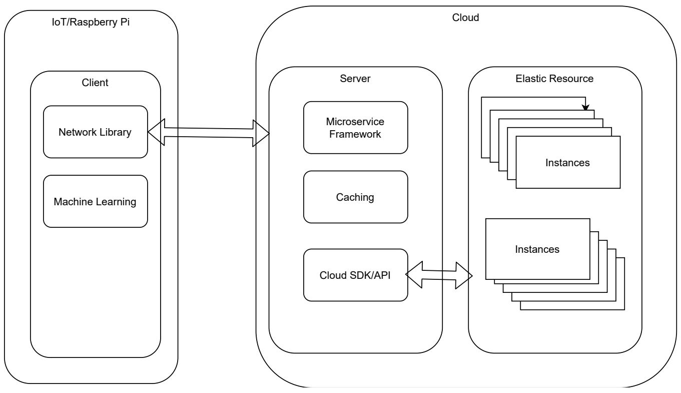

FIGURE 1 System architecture of IoT-Pi

2 | RELATED WORK

Pereira RI et al.[3] have designed a system called Renewable Energy Monitoring System that allows Raspberry Pi nodes to join the real-time remote monitoring on the cloud against decentralized photo-voltaic plants. The system manages to administer the resources by the update of remote firmware with their developed Converted Embed System with analog or digital signal. Saeed U et al.[6] performed an analysis and a comparison of drift fault detection at real-time level on a power system with limited computational capability (such as Raspberry Pi 3) with different machine learning techniques. Cortez E et al.[7] designed a product called Resource Central (RC) to predict resource usage on cloud-based on Virtual Machine (VM) workload data. The authors first extend characters of VM on Microsoft Azure, and then proposed the Resource Central (RC) system to collect information from VM. The system learns the VM behaviours and produces prediction to the resource managers. Moreover, the system show the connection between characters and VM lifetime. In another study Bharadwaja V[8], Artificial Neural Network (ANN) is implemented on Raspberry Pi for signal processing as an IoT application without considering cloud computing, which requires more computational capacity and increases its power consumption for edge devices.

Critical Analysis: Table 4 compares IoT-Pi with existing works. In this study[3], only IoT and Cloud computing are used without considering machine learning. In other related works[8, 6], requires more computational capacity and increases its power consumption for edge devices because they have not used cloud computing. In another work[7], machine learning and cloud computing is used but this work is not applicable for IoT applications. None of these studies have integrated IoT, cloud computing, and machine learning into a unified framework. Therefore, previous works cannot tackle the problem of adapting cloud resources by machine learning operating on the IoT devices. Because of this, a low-cost, intelligent distributed system is needed to manage cloud resources that incorporates machine learning and IoT technologies. Intelligent flexible management of IoT-Pi will make maximum use of cloud resources and business value to overcomes these issues.

3 | IOT-PI: SYSTEM ARCHITECTURE

Fig 1 shows the IoT-Pi system can be formed from 3 components by functionalities: client side, server side, elastic resource.

Client side contains main crucial components in the entire system that client nodes control the entire resource planning and life cycle (which consists of the definition of purpose, defining the hardware, defining the storage, defining the network and security, defining the management process, testing and analysis of the process), including a HTTP connection library and a machine learning framework. The HTTP connection library is mainly used to undertake data retrieval tasks that it connects to remote server to fetch the system information of the instances on the cloud. The retrieved data is near real time and can be stored in a local cache for further analysis and model training. The machine learning framework is sufficient to be light weighted so that it can be running on the critical environment such as Raspberry Pi nodes with few dependencies. In addition, the framework is integrated to be capable of running most machine learning tasks including data pre-processing, feature engineering, training and predicting as well. One of the preferred frameworks is Scikit-learn[9], which is lightweight and comprehensive.

Server side consists of a web microservice framework, a cache backend, cloud service SDK that access the cloud resource with management privileges and a SSH connection library. The web microservice framework provides the application running as a normal web application and easy to be managed in the circumstance of microservice orchestration. Additionally, the web microservice framework adopts a simple API design style following the principle of RESTful so that it can evade the side effect of multiple network connections or network congestion due to its idempotence. The cache backend contributes a provisional key-value storage mechanism to preserve the critical information collected from cloud clusters. If the backend is configured to connect a database, the cached information will be stored for a longer time. Furthermore, cloud service SDK from cloud service providers is integrated to the server side as well due to it is required to get instance information from cloud service providers such as getting active instances or public IP address. For the cloud instances hosted in a private environment such as OpenStack, the cloud service SDK may be changed to the corresponding application or library that interacts the cloud service API accordingly. Moreover, the purpose of a SSH connection library is to provide a connection via SSH to run commands remotely. It is particularly useful to get CPU or memory information by console commands. Apart from running commands, it is also a safe approach to perform heart beat test between server and cloud clusters.

Elastic resource can be implemented on a cloud service (such as Amazon Web Service and Google Cloud Platform) or self-managed bare metal clusters (such as OpenStack clusters). It is optional to set cache service as cache backend in the server side. Additionally, it is required for the cloud service provider that the managed instances on the cloud must be capable of scaling in or out with cloud service SDK or API. This is the key step of adapting resource by the system.

4 | IMPLEMENTATION AND PERFORMANCE EVALUATION

For demonstrating the effectiveness of the prospective design, a case study on application in real world is presented in the section: minimise idle instance on peer-py [1] facilitating the cloud resources from Amazon Web Services (AWS). The experiment is carried out on a Raspberry Pi 4 Model B with configuration of processor as ARM Cortex A72 having a LPDDR4 RAM of 8GB.It is having a 32 bit Operating System(Debian Buster Based). The dataset, is used for experimentation is hierarchical and it is accessible from the hosted public code repository: https://github.com/t1anchen/peer-py/tree/master/data. In real production environment, the dataset is directly sampled from cloud instances via server side. The dataset is generally formatted as numeric number. Live Compare activities populate hierarchy datasets, which record the findings as a hierarchy of objects. The outcomes of retrieval, comparison, and filtering processes are stored in hierarchy datasets. The main fields in the dataset includes (1) timestamp: elapsed seconds from UNIX epoch, (2) ins_id: instance id on the cloud as global unique identifier for instances (3) cpu_ut: cpu utilization percentage (4) is_idle: 0 or 1. The raw data may only include timestamp, ins_id and cpu_ut.

Approach: In the peer-py, the code of client side is working on a Raspberry Pi node. It can be with server side running the same node. However, in the circumstance of real production, the server should be deployed to an individual server or a server isolated from elastic resource side and client side. The main body of server is based on a web microservice framework *Flask*[10]. The framework is adequately simple that it is viable to be extended in microservice and integrated to a complex system. The cache is co-working with the *Flask*, which provides a short-term, temporary data persistence on the server side. To simulate the usage on the cloud, *stress*[2] has been introduced to simulate a configurable amount of CPU, memory or I/O stress on a POSIX operating system. The configuration of stress has a CPU with 4 I/O , running on the remote cloud instances. It has a random timeout with maximum limit of 10 seconds and a extra sleep of 1 second The running tool results in a full load CPU usage within timeout, for example, 100% CPU usage after randomly 4 seconds timeout. To reproduce the scenario of managing the

[1]peer-py https://github.com/t1anchen/peer-py
[2]https://packages.debian.org/buster/stress

TABLE 2 Parameters for machine learning training

Model	Support Vector Machine (SVM) Classifier		kNN Classifier	Logistic Regression		MLPClassifier	
Parameters	kernel	C	n_neighbor	penalty	solver	activation	alpha
Values	poly	1.0	6	L2	liblinear	relu	0.005

entire lifecycle management against cloud instances, it is allowed for peer-py CI test firstly to create instances with the number of creating instances specified by command line arguments. Secondly, after provisioning the instances completes, it is accessible to connect to the remote instances to install the *stress* and run. Afterwards, the test performs all phases in the proposed technique to operate the instances. If the tool is set to Dry Run mode, the chosen instances will not be terminated, but a log entry will display instead. Finally, the tool will persist the data to the local disk for further study.

For client section, it is designed primarily to be running on a Raspberry Pi node, which is also responsible for running machine learning prediction and controlling. However, it can be flexible with server side running on the same node according to the detail machine environment and business restraints. For server section, it is recommended to deploy on a server, which forms a typical client-server architecture. The main body of server is based on a web microservice framework *Flask*[10]. The framework is adequately simple that it is viable to be extended in microservice and integrated to a more complex system in the distributed environment. The cache is co-working with the *Flask*, which provides a short-term, temporary data persistence on the server side. For elastic resource, to simulate the usage on the cloud, *stress*[3] has been introduced to simulate a configurable amount of CPU, memory or I/O stress on a POSIX operating system. The configuration of stress has a CPU with 4 I/O , running on the remote cloud instances. It has a random timeout with maximum limit of 10 seconds and a extra sleep of 1 second The running tool results in a full load CPU usage within timeout, for example, 100% CPU usage after randomly 4 seconds timeout. To reproduce the scenario of managing the entire lifecycle management against cloud instances, it is allowed for peer-py CI test firstly to create instances with the number of creating instances specified by command line arguments. Secondly, after provisioning the instances completes, it is accessible to connect to the remote instances to install the *stress* and run. Afterwards, the test performs all phases in the proposed technique to operate the instances. If the tool is set to Dry Run mode, the chosen instances will not be terminated, but a log entry will display instead. Finally, the tool will persist the data to the local disk for further study.

Local Parameters: For establishing the metrics for evaluating performance of the model, a set of parameters will be introduced when sampling data on the server side. The parameter $t_{\text{cpu\_ut}}$ can be computed from the difference of two samples. When the server raise a SSH connection to the target cloud instance, it retrieves CPU utilization time t_{total} and t_{idle}. Thus, the effective working time may be calculated from $t_{\text{total}} - t_{\text{idle}}$. Equation 1 has provided the method of calculation.

$$t_{\text{cpu\_ut}} = \frac{t_{\text{total2}} - t_{\text{idle2}} - (t_{\text{total1}} - t_{\text{idle1}})}{t_{\text{total2}} - t_{\text{total1}}} \tag{1}$$

where cpu_ut is CPU utilization percentage, t_{total} is total CPU utilisation time and t_{idle} is time when CPU is idle. To simulate the entire life cycle management, many parameters in peer-py can be customised to achieve this. `create` is the parameter that specify the instance creation behaviour. If it set to 0, there is no additional instance to be created. In the future development, the creation behaviour and scale-in strategy will be implemented to replace parts of functionalities of this feature. If it larger than 0, the number specified of instances will be created. After creating instances, there are 90 seconds to wait for provisioning and additional 90 seconds to allow external operation to install, for this experiment we have kept the value as two. *stress* and start to simulating workload. Subsequently, the parameter the parameter `-sample-interval-nanoseconds` denotes the duration time between two command line execution when sampling. In our experiment, we have used as 0.2 milliseconds `-training-rounds` is the number that requires the client to repeat the numbers specified of rounds when sampling data on the training dataset. We have done our experiment with 400 training rounds. Similarly, the ~~`-predicting-rounds`~~`-predicting-rounds` makes the application repeat the sampling data behaviour but on testing dataset. The ~~predicting~~predicting rounds for our experiment are 100. The selection of the parameters, particularly the machine learning model settings for training on the IoT devices according to previous studies[6,11] to optimise the model training. Table 2 shows the parameters for machine learning training.

Cloud: The AWS Elastic Compute Cloud (EC2) service is adopted to provide the cloud resource and management service concerning the virtual cloud instances. The SDK of AWS EC2 is prefixed as *boto*[4] and it provides a comprehensive API service

[3]https://packages.debian.org/buster/stress
[3]The value of rounds depends on the dataset size.
[4]The old version is boto2, and the latest version is boto3 in Python

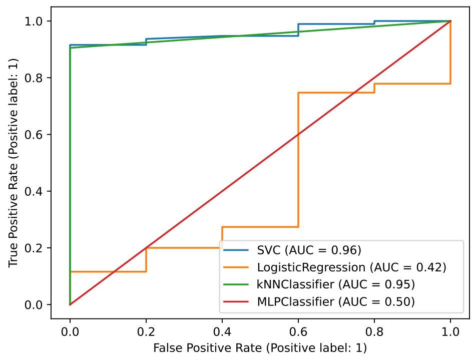

FIGURE 2 ROC curves for different models

TABLE 3 Comparison between KNeighborClassification and SVC

ML Algos	Accuracy	Precision	Recall	F1 Score	AUC
KNN	0.79	1.0	0.72	0.83	0.86
SVC	0.74	0.74	1.0	0.85	0.85

to manage the cloud resources hosted by AWS, particularly the lifecycle management of instances. It also provides monitoring features by combining the monitoring product AWS CloudWatch with the built-in monitoring of EC2 instance, which could provide a more integrated and complex monitoring service than peer-py. However, peer-py is designed to provide a general implementation for cloud resource management, and it is free to be adapted to many alternative cloud services apart from AWS. SDK version used are boto3.1.17.105 and botocore 1.20.105. The specification of the cloud instance used for the experiment is having a Instance type t2 micro,with a single CPU. The Amazon Machine Image (AMI) is ami-0747bdcabd34c712a. The Operation system is Ubuntu server 18.04 with 64 bit architecture, having a general purpose (SSD) elastic book store (EBS) of volume 8 GB. The cloud instance has no multiple availability zone, no spot and recovered instance and no load balancer enabled.
Execution Time: The execution time of completing the entire lifecycle management. In this table, **TSR** means *Training Sample Rounds* and **PSR** means *Prediction Sample Rounds*. Meanwhile, the unit of **Elapsed Time** is second. It is seen that with TSR value 40 and PSR 10, the elapsed time is 265.42 seconds, similarly for TSR as 80 and PSR as 20, the elapsed time is 328.64 seconds, finally for TSR value as 400 and PSR as 100 the elaspsed time is 871.368381 seconds.

Accuracy: The Receiver Operating Characteristic (ROC) curve is one of the performance metrics against machine learning models. The corresponding ROC of the prediction including different models has been presented in the Fig. 2. From the chart it is reckoned that K-Nearest Neighbor (kNN) Classification and Support Vector Classifier (SVC) own the better performance around ROC, with Area Under Curve (AUC) of ROC is 0.95 and 0.96 individually. When AUC scores approximate to 1, it means the model has better performance. Additionally, the mean accuracy scores of these two models working on the instances with simulated workload are both around 95%, respectively. The mean training accuracy of kNN and SVC are 0.6675 and 0.7475 respectively while the testing accuracy is 0.7800 for both the algorithms for idle instance where as for busy instance, training accuracy of kNN and SVC is 0.9250 and 0.9375 respectively, and testing accuracy for both the algorithm is 0.9499 Table 3 shows the details of the two outperformed models. Table 4 shows a comparison of IoT-Pi relative to the baseline using CPU usage and F1-score. It can be seen that IoT-Pi gives enhanced performance when measured by CPU usage and F1 score, while the accuracy is nearly the same.

5 | CONCLUSIONS AND FUTURE DIRECTIONS

In this work, an IoT-Pi framework has been proposed to make IoT devices as main working platform in real world to optimize the cost of cluster usage. IoT-Pi framework consists of three phases: client side, server side and elastic resource. Further, four

TABLE 4 Comparison of IoT-Pi with baseline approach

Existing works	**CPU Usage** (%)	**F1-Score** (%)
Cortez E et al.[7]	74	76
IoT-Pi (this work)	65	85

different machine learning models (SVM Classifier, Logistic Regression, kNN Classifier and, MLP-Classifier) are implemented and results show that these models are able to predict the idle state of instance with over 70% accuracy. High availability is one of the most critical topics in massive high-performance computation and cloud computing. In IoT-Pi,the implementation has not involved high availability considerations. However, it is inevitable to respond the challenge from network congestion and instability in the future. The application is only working in serial order so that it cannot access the resources concurrently. Improving concurrency in the future is meaningful so that it can accelerate the speed of accessing resource.

DATA AVAILABILITY STATEMENT

IoT-Pi is released as an open source software. The data and code that support the findings of this study are available in GitHub at https://github.com/iamssgill/IoT-Pi

References

1. Sajjad M, Nasir M, Muhammad K, et al. Raspberry Pi assisted face recognition framework for enhanced law-enforcement services in smart cities. *Future Generation Computer Systems* 2020; 108: 995–1007.
2. Canedo J, Skjellum A. Using machine learning to secure IoT systems. In: IEEE. ; 2016: 219–222.
3. Pereira RI, Dupont IM, Carvalho PC, Jucá SC. IoT embedded linux system based on Raspberry Pi applied to real-time cloud monitoring of a decentralized photovoltaic plant. *Measurement* 2018; 114: 286-297.
4. Kreutz Dea. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE* 2015; 103(1): 14-76.
5. He S, Guo L, Guo Y. Real Time Elastic Cloud Management for Limited Resources. In: ; 2011: 622-629.
6. Saeed U, Ullah Jan S, Lee YD, Koo I. Machine Learning-based Real-Time Sensor Drift Fault Detection using Raspberry Pi. In: ; 2020: 1-7.
7. Cortez E, Bonde A, Muzio A, Russinovich M, Fontoura M, Bianchini R. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In: SOSP '17. Association for Computing Machinery; 2017: 153–167.
8. Bharadwaja V, Ananmy R, Nikhil S, Vineetha KV, Shah J, Kurup DG. Implementation of Artificial Neural Network on Raspberry Pi for Signal Processing Applications. In: ; 2018: 1488-1491.
9. Pedregosa F, Varoquaux G, Gramfort A, et al. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 2011; 12: 2825–2830.
10. Grinberg M. *Flask web development: developing web applications with python.* " O'Reilly Media, Inc." . 2018.
11. Alsouda Y, Pllana S, Kurti A. IoT-Based Urban Noise Identification Using Machine Learning: Performance of SVM, KNN, Bagging, and Random Forest. In: COINS '19. Association for Computing Machinery; 2019; New York, NY, USA: 62–67.