CAPTAIN: A Testbed for Co-Simulation of Scalable Serverless Computing Environments for AIoT Enabled Predictive Maintenance in Industry 4.0

Muhammed Golec, Huaming Wu, *Senior Member, IEEE*, Ridvan Ozturac, Ajith Kumar Parlikad, *Member, IEEE*, Felix Cuadrado, Sukhpal Singh Gill, and Steve Uhlig

***Abstract*—The massive amounts of data generated by the Industrial Internet of Things (IIoT) require considerable processing power, which increases carbon emissions and energy usage, and we need sustainable solutions to enable flexible manufacturing. Serverless computing shows potential for meeting this requirement by scaling idle containers to zero energy-efficiency and cost, but this will lead to a cold start delay. Most solutions rely on idle containers, which necessitates dynamic request time forecasting and container execution monitoring. Furthermore, Artificial Intelligence of Things (AIoT) can provide autonomous and sustainable solutions by combining IIoT with artificial intelligence (AI) to solve this problem. Therefore, we develop a new testbed, CAPTAIN, to facilitate AI-based co-simulation of scalable and flexible serverless computing in IIoT environments. The AI module in the CAPTAIN framework employs random forest (RF) and light gradient-boosting machine (LightGBM) models to optimize cold start frequency and prevent cold starts based on their prediction results. The proxy module additionally monitors the client-server network and constantly updates the AI module training dataset via a message queue. Finally, we evaluated the proxy module's performance using a predictive maintenance-based real-world IIoT application and the AI module's performance in a realistic serverless environment using a Microsoft Azure dataset. The AI module of the CAPTAIN outperforms baselines in terms of cold start frequency, computational time with 0.5 ms, energy consumption with 1161.0 joules, and CO2 emissions with 32.25e-05 gCO$_2$. The CAPTAIN testbed provides a co-simulation of sustainable and scalable serverless computing environments for AIoT-enabled predictive maintenance in Industry 4.0.**

***Index Terms*—Artificial intelligence (AI), cloud computing, flexible manufacturing, Industrial Internet of Things (IIoT), predictive maintenance, serverless computing.**

Received 13 September 2024; revised 9 October 2024; accepted 25 October 2024. Date of publication 30 October 2024; date of current version 8 August 2025. The work of Muhammed Golec was supported by the Ministry of Education of the Turkish Republic for the funding. The work of Huaming Wu was supported in part by the National Natural Science Foundation of China under Grant 62071327, and in part by the Tianjin Science and Technology Planning Project under Grant 22ZYYYJC00020. The work of Felix Cuadrado was supported by HE ACES Project under Grant 101093126. *(Corresponding author: Huaming Wu.)*

Muhammed Golec is with the School of Electronic Engineering and Computer Science, Queen Mary University of London, E1 4NS London, U.K., and also with Abdullah Gul University, 38080 Kayseri, Türkiye (e-mail: m.golec@qmul.ac.uk).

Huaming Wu is with the Center for Applied Mathematics, Tianjin University, Tianjin 300072, China (e-mail: whming@tju.edu.cn).

Ridvan Ozturac is with the Engineering Team, Trendyol Group, 34485 Istanbul, Türkiye (e-mail: ridvan.ozturac@trendyol.com).

Ajith Kumar Parlikad is with the Institute for Manufacturing, Department of Engineering, University of Cambridge, CB3 0FS Cambridge, U.K. (e-mail: aknp2@cam.ac.uk).

Felix Cuadrado is with the School of Telecommunications Engineering Madrid, Universidad Politécnica de Madrid, 28032 Madrid, Spain (e-mail: felix.cuadrado@upm.es).

Sukhpal Singh Gill and Steve Uhlig are with the School of Electronic Engineering and Computer Science, Queen Mary University of London, E1 4NS London, U.K. (e-mail: s.s.gill@qmul.ac.uk; steve.uhlig@qmul.ac.uk).

Digital Object Identifier 10.1109/JIOT.2024.3488283

I. Introduction

The concept of Artificial Intelligence of Things (AIoT) emerged by integrating Industrial Internet of Things (IIoT) devices and artificial intelligence (AI) models [1]. Processing and interpreting the data collected from IIoT and sensors using AI models provide convenience to human life in various fields, such as military [2], civilian [3], and flexible manufacturing [4]. Studies in which early diagnosis is made using AI models of data collected from patients through IIoT and sensors are good examples of these applications [5]. Industry 4.0 uses cloud and edge-based systems with high-processing power and storage capabilities to process massive amounts of data, which increases electricity consumption and carbon emissions [6], [7]. Increasing environmental and energy crises have made eco-friendly technologies, such as green IIoT and cloud, which reduce electricity consumption and carbon emissions mandatory for Industry 4.0 [8].

A. AIoT Enabled Predictive Maintenance in Industry 4.0

Currently, predictive maintenance is one of the important challenges that must be solved to optimize asset management performance in Industry 4.0 [4], [9]. AIoT can provide autonomous and sustainable solutions to optimize predictive maintenance for anticipating future challenges [4]. Therefore, we need a scalable and sustainable computing environment to enable flexible manufacturing in Industry 4.0 [10]. Amazon's Lambda platform introduced serverless computing [11], [12], a new cloud computing paradigm [13], in 2014 as an environmentally friendly solution to this issue [4]. The word *"serverless"* is not because there are no servers in this paradigm, but to show that background management is entirely the cloud providers' responsibility [14], [15]. It has three main advantages [16]: 1) clients are completely isolated from server administration, so they can focus more on code development; 2) with the pay-as-you-go model it offers to customers, only a fee is charged for the space and processing power used; and 3) by offering dynamic scalability, it can automatically scale resources to meet fluctuating demand spikes [6]. Besides the

TABLE I
COMPARISON OF OUR PREVIOUS WORKS WITH THE PROPOSED CAPTAIN FRAMEWORK

Works	AI Models	Environment	Dataset	Objective	IIoT	Co-Simulation
ATOM [18]	DDPG	Serverless Edge	Healthcare	Cold Start Prediction	No	No
MASTER [19]	XGBoost	Serverless Edge	Predictive Maintenance	Cold Start Prediction	Yes	No
CAPTAIN Testbed	RF + LGBM	Serverless	Predictive Maintenance & Microsft Azure	Cold Start Prediction & Prevention	Yes	Yes

advantages of serverless computing, there are still problems to be solved, cold start latency is one of the most important of these problems [16]. In serverless computing, each function is assigned to a container for execution. Assignment to the container is done in two ways [10]: 1) if the container is ready, the function is directly assigned to the container and executed and 2) if there is no ready container in the environment, a new container is created after the necessary libraries are loaded, the environment is set up, and the codes are loaded into the container. A particular resource is required for the execution and creation of containers. If there is no request to the serverless platform within a certain period, the containers are deleted, thus avoiding wasted resources. This feature of serverless computing is called zero scaling [17]. After the deletion of resources, a certain amount of time will be required to create a new container whenever a new request arrives. This required time causes a cold start. Likewise, if there are more requests than the container in the environment can handle, cold start latency will occur. It has been observed that factors, such as software language, CPU, and RAM, affect cold start latency [14]. So, there is a need to address these challenges if this promising paradigm can be utilized to enable flexible manufacturing in Industry 4.0 [4].

B. Our Previous Works

In our previous works, we developed two different frameworks named ATOM and MASTER, which predict cold start occurrence for the serverless edge computing environments [18], [19]. The ATOM framework uses the deep deterministic policy gradient (DDPG) to predict cold starts in the healthcare domain, while the MASTER framework employs the extreme gradient boosting (XGBoost) model to predict cold start latency. However, none of our previous works have been able to mitigate or prevent cold start latency, highlighting the need for new research to address this issue and enhance predictive maintenance performance for IIoT. Therefore, we developed a novel testbed, CAPTAIN, utilizing the random forest (RF) and LightGBM (LGBM) models, given their demonstrated effectiveness on complex patterned and large datasets like cold start [20]. Table I shows the main differences between our previous works [18], [19], and the proposed CAPTAIN testbed in terms of key parameters.

C. Motivation and Our Contributions

AIoT/IIoT applications for smart manufacturing that require high-processing power and energy consumption can use serverless computing, an environmentally friendly paradigm [21]. However, serverless releases idle resources (scales to zero) to prevent resource wasting, causing a cold start problem that leads to undesirable delays in time-sensitive AIoT scenarios, such as healthcare [17] and predictive maintenance-based real-world IIoT applications [9]. The literature [10] reported that there are solutions that cause resource waste, such as 1) keeping containers warm, where containers are kept warm for a certain period of time after the execution of functions is completed and 2) container pools, where containers are kept working and called when needed. The cold start problem in serverless computing should be solved because it has negative effects on 1) user experience; 2) cost, and 3) time-sensitive applications [22]. In most of the solutions offered (such as keeping the container warm), resources are wasted because the containers are kept running even in cases where a cold start does not occur. Although ATOM [18] and MASTER [19] can predict cold start latency, they did not solve the cold start problem, which requires solutions to reduce the frequency of cold starts and prevent cold start occurrence. AI-based time-series models can be used to fill this gap by allocating serverless resources in advance and predicting future demands by monitoring latency and request patterns for IIoT applications. Therefore, we propose the CAPTAIN, an AI-based serverless computing framework for IIoT applications that reduces the frequency of cold starts. CAPTAIN consists of the AI module to ping the server to prevent a cold start, and the Proxy module to update the dataset to train the AI module continuously. The Proxy Module constantly monitors the communication between the server and client to keep the dataset updated with a message queue (MQ) protocol. In the AI module, we used RF and LightGBM, which have higher prediction performance than other time-series models [23]. We selected these two models due to their utilization of the tree-based ensemble method, which offers several advantages [24]: 1) parallel processing, which increases the prediction performance in complex and large datasets, such as cold start and 2) being resistant to overfitting situations by creating a more general model by combining more than one learner. The main contributions of this work are as follows.

1) Present a new framework, CAPTAIN, for scalable and sustainable serverless computing in IIoT environments for flexible manufacturing.
2) Propose a time series model (LGBM and RF)-based approach for the optimization of cold start frequency by predicting the cold start latency in advance.
3) Utilize predictive maintenance-based IIoT application and Microsoft Azure datasets for performance evaluation.
4) Evaluate the performance of CAPTAIN using a Google cloud platform (GCP)-based realistic environment and compare it with three baselines (which used autoregressive integrated moving average (ARIMA) [25], seasonal auto-regressive integrated moving average (SARIMA) [26] and long short-term memory (LSTM) [27]).

TABLE II
COMPARISON OF CAPTAIN FRAMEWORK WITH EXISTING STUDIES

Studies	Platform	Method	Adaptability	External Resource Load	Online ML	AIoT	Environment	IIoT (Predictive Maintenance)	Co-Simulation
		Cold Start Objective: Latency							
[28]	AWS	Function Fusion	✗	✗	✗	✗	Serverless	✗	✗
[29]	OpenWhisk	Container Scheduling	✗	✗	✗	✗	Serverless	✗	✗
[30]	Knative	Pod Migration	✗	✗	✗	✗	Serverless	✗	✗
[31]	AWS	Application-level Optimization	✓	✗	✗	✗	Serverless	✗	✗
		Cold Start Objective: Frequency							
[14]	Kubeless	Q-Learning	✓	✗	✗	✗	Serverless	✗	✗
[27]	AWS Azure Openfaas Openwhisk	DNN, LSTM	✓	✗	✗	✗	Serverless	✗	✗
[26]	Kubeless	SARIMA	✓	✗	✗	✗	Serverless	✗	✗
[9]	FogWorkflowsim	GA	✗	✗	✗	✗	Fog	✓	✗
[4]	An industrial case	LSTM	✗	✗	✗	✗	✗	✓	✗
CAPTAIN Testbed	GCP	LGBM and RF	✓	✓	✓	✓	Serverless	✓	✓

Column 1: *Studies* considered in this research. **Column 2:** *Platforms* used to test the performance. **Column 3:** *Methods* used in the studies. **Column 4:** *Adaptability* indicates whether the frameworks or approaches are designed independently of the platform. **Column 5:** *External Resource Load* shows the methods, if any, used by the studies to prevent adding external resource loads to the system. **Column 6:** To solve the cold start problem, *Online ML* is enables adaptability to dynamic contexts and real-time learning.**Column 7:** Solve cold start problem in *AIoT* environments. **Column 8:** *Environment* indicates the implementation environment. **Column 9:** *Predictive Maintenance* indicates whether it is used in the studies. **Column 10:** *Co-Simulation* indicates a lightweight tested.

D. Lightweight Testbed for Co-Simulation

We developed a coupled-simulation (co-simulation) framework known as CAPTAIN to build an experimental testbed that simulates scalable serverless computing environments effectively. We used a real-world IIoT application based on predictive maintenance and Microsoft Azure datasets to test in realistic serverless computing environments using GCP. This can be used by future researchers to design and test a wide range of new models with unpredictable configurations. The main features of the CAPTAIN testbed are as follows.

1) Prevents wasted resources on the server side and reduces resource waste by pinging the server to start containers only when necessary.
2) Minimizes the external latency and system resource load that may occur by using the GO Land software language and rabbit message queue (Rabbit MQ) protocols.
3) Continuously updates the training dataset using an online machine learning (ML)-based system to increase the time series prediction success rate.
4) Outperforms in terms of cold start frequency, computational time, energy consumption, and CO_2 emissions as compared to the baselines [25], [26], [27] for predictive maintenance-based real-world IIoT application.

The remainder of this article is organized as follows. Studies to solve the cold start problem in the literature are examined in Section II. The CAPTAIN Framework is presented in Section III. The experimental results and performance comparison are given in Section IV. Finally, Section V concludes this article.

II. Related Works

We categorize the reviewed studies on cold starts [22], which include: 1) approaches to minimize cold start latency; 2) strategies to reduce cold start frequency; and 3) existing literature on IIoT and predictive maintenance.

A. Studies to Reduce the Latency Caused by Cold Start

Agarwal et al. [14] conducted a study aiming to reduce cold start frequency by using Q-learning, a reinforcement learning technique. Environment states and reward systems are designed by monitoring per-instance CPU utilization. The Kubeless platform was used to test the workload. Kumari et al. [27] adopted a two-stage ML strategy to optimize the cold start frequency. First, idle container windows are predicted with the deep neural network (DNN) model. In the second stage, it follows a policy that activates preheated containers by predicting future server requests with the LSTM model. They compared the model to existing cold start reduction techniques on platforms, such as AWS Lambda, Microsoft Azure, OpenFaaS, and Openwhisk. Jegannathan et al. [26] used the SARIMA time series model to reduce the cold start frequency. This study aims to reduce the container preparation time that may cause a cold start by estimating the incoming request time. The authors deployed their proposed system to a predicted-based autoscaler (PBA) and compared its performance with the default horizontal pod autoscaler (HPA) in Kubernetes. According to the result, the system integrated into PBA has 18% higher performance. Lee et al. [28] tried to reduce cold start latency in functions running in parallel with their proposed technique. Using the Function Fusion technique, the two functions are combined, thus eliminating the cold start delay for the second function. Experiments have shown that the response time is reduced by 28% to 86%.

B. Studies to Reduce the Frequency of Cold Start Occurrence

Wu et al. [29] proposed a container scheduling strategy to recommend terminating and rebuilding containers according to the distribution of requests. Authors used OpenWhisk to test it, and results showed that it reduces cold start by as much as 85%. Lin and Glikson [30] reduced container preparation latency by using a pod migration-based technique in their lead system. Instead of creating a new container for requests to serverless, prewarmed containers are checked. Moreover, if there is an idle warmed container, it is allocated to execute the incoming function to the server. This way, a new container is not allocated for each function, and the cold start latency time is reduced. The results show that the container preparation time is shortened by 85%. Speeding up container startup time also means reducing cold start latency. For this reason, Liu et al. [31] determined the application optimization strategy in their proposed approach called FaaSLight. This strategy ensures that the basic part of the code in the application is loaded into the container. In this way, container preparation time and latency are reduced without the need to load the entire code.

C. Studies on Predictive Maintenance in Industry 4.0

Teoh et al. [9] presented a framework based on IoT and Fog-Computing for effective resource management in Industry 4.0. In this framework, a real failure affecting the production line is predicted using a genetic algorithm (GA) and thus allows the business owner to intervene in the production line. In another study, Sang et al. [4] proposed a predictive maintenance model, PMMI 4.0, in flexible manufacturing. PMMI 4.0 makes predictions using an LSTM-based time series model.

D. Critical Analysis

Table II shows a comparison of existing studies with the CAPTAIN framework. Previous works have been deployed on the open-source platform, and performance tests have been conducted. CAPTAIN, on the other hand, was deployed on GCP, a commercial platform, using a real-world IIoT application with predictive maintenance [19], [32]. CAPTAIN framework is adaptable because it is designed independently of the platform and can be easily deployed on any open-source platform. The Industry 4.0 workload has been created to test the CAPTAIN, but it can be applied to any scenario and workload that uses serverless architecture, such as e-commerce. Since it uses Rabit MQ, an open-source MQ protocol, it does not impose an external resource load and costs on the system. Further, server–client communication is always monitored to improve AI module performance by updating the dataset (Online ML). None of the studies reviewed target AIoT applications. The CAPTAIN framework is deployed in a serverless computing environment, facilitating the effective use of resources with anywhere access and auto-scalability for flexible manufacturing. Finally, we used predictive maintenance dataset to measure the performance of the CAPTAIN framework using co-simulation of-based sustainable and scalable serverless computing environments.

III. Captain Framework

In this section, we discuss the CAPTAIN framework along with the implementation details of its components.

A. System Model

Fig. 1 shows the system model for the CAPTAIN Framework, which is created by combining two different main modules: 1) proxy module, where client-server communication is monitored and 2) AI module, where time series-based forecasts are generated to reduce cold start frequency. We discuss these two modules in detail in the subsequent sections.

Proxy Module: This module monitors the communication between the client and the server and records data, such as latency, the timestamp when the transaction took place, the host from which the message was received, and the target to which it was forwarded, using a MQ protocol. Fig. 2 shows the architecture of the proxy module. The proxy module uses an MQ protocol because of its advantages, such as high performance, reliability, and granular scalability [33]. Following is a summary of these concepts and the benefits they provide.

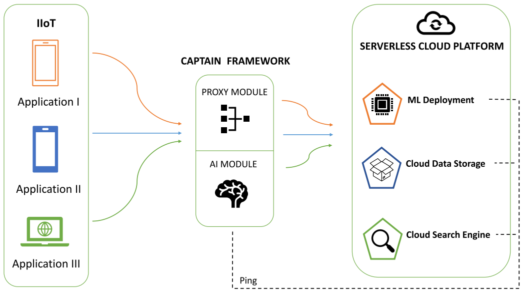


Fig. 1. System model.

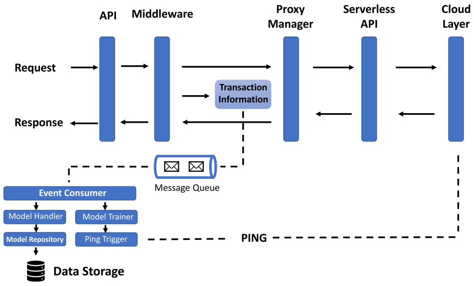


Fig. 2. Architecture of the proxy module.

1) *High Performance:* Incoming data can be added to the queue without waiting for the completion of the processes.
2) *Increased Reliability:* It reduces data loss to zero when different parts of the system are offline, reducing the errors that may occur.
3) *Granular Scalability:* The system can add requests to the queue at peak times without the risk of conflicts. Here, the risk of conflict means resource contention. Conflicts that occur when applications access the shared resource's disk storage, cache, and memory are known as resource contention [34]. Resource Contentions negatively affect Quality of service (QoS) parameters, such as *throughput*. With granular scalability, the amount of data in the queue can be scaled up and down.

Transaction information from MQ is saved in the database of the AI module with event-driven architecture. Using event-driven architecture aims to respond quickly enough in heavy data traffic situations, avoid system interruptions, and prevent malfunctions, such as communication delays. The data sampling time interval is the time difference between two requests. This can take from a few seconds to several hours. In the CAPTAIN framework, this time should be distributed between 5–20 min. Because in commercial platforms, such as AWS, Google, and Microsoft, containers that complete their function are kept warm for a certain period of time to reduce the occurrence of cold starts [35]. This is 15 min for GCP, so the server must not receive a request for at least 15 min [36] for a cold start to occur. The CAPTAIN framework constantly monitors server–client communication using online ML. Moreover, in this way, the AI module constantly updates its database. Our aim in doing this is to predict the cold start latency occurrence time as accurately as possible, thereby reducing

Algorithm 1 Event-Driven Architecture for Proxy Module

```
Input: Request
Output: Response
Variables:
$R \leftarrow$ Request
$R_k \leftarrow$ Routing Key
$T_h \leftarrow$ Target Host
$T_p \leftarrow$ Target Path
$D \leftarrow$ Duration
Begin
    $R$ = API.called()
    $R_k$ = KeyExtract(R.Routing)
    $T_h$ = GetTargetHost($R_k$)
    $T_p = T_h$ + R.Routing.Remove($R_k$)
    $R$.Path = $T_p$
    $R$ = proxy.call($R$)
    endTime = time.Now()
    async{
            event = TransactionInfoEvent(
            Timestamp: time.Now()
            Key: $R_k$
            Host: $T_h$
            Target: $T_p$
            Duration:D)
            rabbitPublisher.publish(event)}
    return Response
End
```

Algorithm 2 CAPTAIN Operating Mechanism

```
Input: $T_i$
Output: Ping
Variables:
$T_i \leftarrow$ Transaction Information
$Ping \leftarrow$ Ping
$P_m \leftarrow$ Proxy Module
$M_q \leftarrow$ Message Queue
$S_c \leftarrow$ Serverless Cloud
$DB_{AI} \leftarrow$ AI Database
$T_s \leftarrow$ Time Series Model
$F_c \leftarrow$ Forecast Value
$\tau \leftarrow$ Threshold
$t, l \leftarrow$ Time and Latency
Begin
    $P_m(\sum_0^i T) \longrightarrow M_q$
    $M_q(\sum_0^i T) \longrightarrow$ Update($DB_{AI}$)

    for $t, l$ in $DB_{AI}$:
        Train ($T_s$)
    $F_c$ = Forecast ($T_s$)
    if $F_c > \tau$:
        Send (Ping) $\rightarrow S_c$
End
```

the cold start frequency. The pseudo-code of the proxy module is given in Algorithm 1. First, the request routing information from the proxy and the key of the registered serverless API are extracted. In the next step, the serverless host address belonging to this key is obtained. The target path (target serverless address) is obtained with this information. On the incoming request, only the target path is changed, the request is sent, and the response is returned to the client. Finally, Timestamp, Key, Host, Target, and Duration information are sent asynchronously to Rabbit MQ as transaction info. Since there are no loops in Algorithm 1, the *time complexity* is $\mathcal{O}(1)$.

IIoT Application: We created an IIoT application using a predictive maintenance dataset as discussed in our previous work, MASTER [19]. We used the GCP Cloud Functions-based realistic serverless platform and the configuration settings for the instance created in GCP-cloud functions are as follows: Region: "Europe-west2," Trigger: "Http," Runtime: "Python 3.7" and Memory allocated: "256 MB." The AI model is deployed on a serverless instance to detect synchronous motor failures by analyzing six variables transmitted by sensors: "Type," "Air temperature," "Process temperature," "Rotational speed," "Torque" and "Tool wear." To create the workload for an IIoT application, a successively increasing number of concurrent requests are sent to the instance on the server. Here, concurrent requests represent the number of users and the dynamic scalability performance of the system is observed with the increasing number of users. All workloads, including six variants, are created in HTTP format using Apache JMeter running on the local computer and sent to the AI model, which detects synchronous motor failure in the production line and is deployed on the GCP instance.

AI Module: This module is trained with the dataset created using the proxy module and PINGs the server according to the AI-based time series model prediction result contained in it. In this way, containers are run before a request that could cause a cold start comes to the server, preventing a cold start. The main pseudo code for The CAPTAIN Framework is given in Algorithm 2. Transaction information (T_i) from the proxy module, which is constantly listening between the client and server, is continuously sent to Rabbit MQ, which is the MQ protocol. Rabbit MQ transmits the information it receives to the AI Database (DB_{AI}) asynchronously, and the data is constantly updated in this database. DB_{AI} is created using the online ML technique and trains the Time Series Model (T_s) continuously. Our goal in using online ML is to increase the T_s accuracy rate as much as possible. If the forecasts made using T_s are greater than the previously determined threshold (τ), it is predicted that the cold start will occur, and the server will be pinged. In this way, a cold start that may occur on the server is prevented. As can be seen from the pseudocode, the *time complexity* is $\mathcal{O}(n)$ because there is only a 'for loop'. And the for loop will continue to run until the number of elements in the dataset reaches n. As n increases, so does time complexity.

B. Implementation Details

The implementation details for the CAPTAIN framework, the proxy module, and the AI module are given below.

1) *Proxy Module:* For the proxy module, we utilize RabbitMQ due to its open-source nature, platform independence, and support for a wide range of programming languages [37]. The proxy module is implemented in the Go programming language [38], which helps minimize external latency between the server and the client.
2) *AI Module:* In the CAPTAIN framework, a single time-series model is employed for efficiency. Among the available models, RF demonstrated superior performance compared to LGBM, making it the chosen model for the AI module.

Fig. 3 illustrates the network diagram of the CAPTAIN framework and its implementation details.

1) *Cloud User:* The layer where IIoT and sensors are located;
2) *Switch:* A device that facilitates data transmission between different network devices.

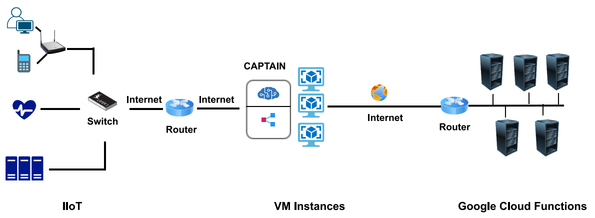


Fig. 3. Network diagram of the CAPTAIN framework.

3) *Routers:* Components responsible for routing data between devices across the Internet and within the network.
4) *Virtual Machine (VM) Instance:* The environment where the CAPTAIN framework is deployed.
5) *Cloud Function Instance:* Environments where the ML model is deployed.

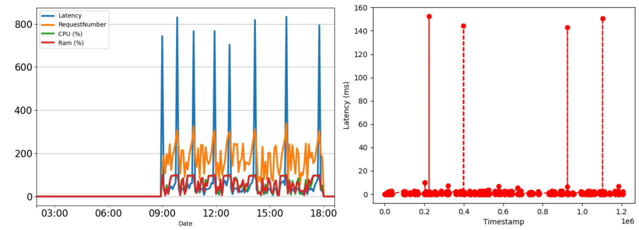
The proxy and AI modules of the CAPTAIN framework are deployed on VM instances using GCP's Compute Engine service. VM instances are utilized to accommodate varying numbers of cloud users (IIoT applications) accessing the CAPTAIN framework, with Compute Engine scaling VM instances up or down to manage fluctuating workloads. The ML model, designed for a serverless architecture, is deployed to Cloud Function instances.

C. Time Series Models

We have used AI-based time-series models for the cold start prediction in the proposed CAPTAIN framework (LightGBM and RF) and baselines (ARIMA, SARIMA, and LSTM). ARIMA model is used in time series analysis and forecasting [39], which is created based on the historical values found in the time series, and predictions are made. While creating the ARIMA model, p, d and q parameter values should be determined, respectively. p represents the order of time delays, d represents the number of times the data history values are subtracted to make the data stationary, and p represents the order of the moving-average model. The formulation of the ARIMA model is given as follows:

$$y'_t = c + \varphi_t y'_{t-1} + \cdots + \varphi_p y'_{t-p} + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q} + \varepsilon_t \qquad (1)$$

where auto-regressive model parameters are φ and moving average model parameters are θ. ε are error parameters. There are methods, such as drawing an ACF graph and auto ARIMA, to find the parameters (p, d, q) in the ARIMA model. SARIMA is a version of the ARIMA model with seasonal components to provide higher performance in seasonal time series than ARIMA [40]. Therefore, it provides higher performance in seasonal time series than ARIMA [40]. Contains $(p, d, q) * (P, D, Q, s)$ as parameters, where (p, d, q) are the parameters used for the nonseasonal part of the model, and (P, D, Q) are the parameters used for the seasonal part of the model and represent the same things. Unlike ARIMA, the s parameter is used for the seasonal length in the data. LSTM is a deep learning model used in sequential data structures, especially in time series [41]. This model is very successful in the time series analysis of sequential data using recurrent neural networks. LGBM [42] is a tree-based ML algorithm like XGBoost but it works faster than XGBoost and makes predictions by following the same formulation for time-series problems. RF [43] is an ensemble learning-based ML algorithm. Unlike XGBoost and LGBM, it trains each tree independently. For this reason, time-series data with features, such as temporal trends or seasonality, should be handled carefully.



(a) (b)

Fig. 4. Latency variation in Cold Start and Microsoft Azure datasets for performance evaluations. (a) Predictive maintenance dataset. (b) Microsoft Azure dataset.

D. Datasets

1) Predictive Maintenance Dataset: It is a synthetic, generic dataset derived from modeling a continuously running production line deployed in a real-world serverless environment.[1] It includes situations where the number of instances is triggered and scaled, which is a feature of great importance in serverless computing. In this dataset, it was reported that cold start occurred for three different situations [18]: 1) the first request to the instance; 2) the case of a new request to the instance that has not been running for more than 15 min (scale to zero feature); and 3) more than 300 requests to the instance. For the reason given in 3), the PING mechanism can be considered in the case where more instances of the same function request distribution. In other words, the PING mechanism works actively in cases where a new instance needs to be distributed due to large request rates. Fig. 4(a) shows the latency variation in the predictive maintenance dataset. The predictive maintenance dataset was created by examining a system that runs continuously six days a week and contains date, time, hour, day, latency, request, CPU, and RAM usage variables. These variables show, respectively, the date of arrival of the request to the server, the hour of arrival of the request to the server, the day of arrival of the request to the server, the amount of the request's latency between the client and the server, the number of simultaneous requests sent to the server, and the CPU and RAM usage rates of the instances.

2) Azure Dataset: It is a public dataset made available by serverless service providers.[2] This dataset, together with the predictive maintenance dataset, is utilised for evaluating baseline and CAPTAIN framework performance and for selecting the AI model to be employed in the AI module of the CAPTAIN. We chose this dataset because, unlike the

[1] https://github.com/MuhammedGolec/Cold-Start-Dataset-V2
[2] https://github.com/Azure/AzurePublicDataset

TABLE III
AI MODEL PARAMETERS

Work	Model	Parameters
SWILD [25]	ARIMA	$p = 1$, $d = 1$, $q = 0$, $P = 0$, $D = 1$, $Q = 0$
PBA [26]	SARIMA	$p = 0$, $d = 1$, $q = 0$, $P = 0$, $D = 1$, $Q = 1$
TLA [27]	LSTM	"scaler_type"=robust, "max_steps"=500, "encoder_n_layers"=2, "encoder_hidden_size"=128
CAPTAIN	LGBM	'learning_rate': 0.1, 'num_leaves': 31, 'max_depth': -1, 'min_child_samples': 20
	RF	'n_estimators': 100, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_features': 'auto'

predictive maintenance dataset, it has a wider range of possible request patterns and includes cases when the same function is deployed in parallel. Fig. 4(b) shows the latency variation in the Microsoft Azure dataset. In this dataset, which collects information about function invocations on Microsoft Azure for a 13-day period, there are four variables: 1) "app;" 2) "func;" 3) "end_timestamp;" and 4) "duration" [44]. These variables represent the ID of the application, the ID of the function of the applications, the end time of the function, and the total execution time of the function.

IV. Performance Evaluation

A. Experimental Setup

This section outlines the system and model configuration that were used to conduct these experiments, both locally and on a real-world serverless platform. The purpose of these experiments was to test the CAPTAIN framework's ability to predict cold starts, reduce their frequency, and compare it with basic methods.

1) Local System Environment: We conducted cold start prediction performance comparisons of AI models and baselines using the system configurations for the local PC: Intel Core i7-10750H CPU, 2.6 to 5.0 GHz clock speed, 16 GB of RAM, and Windows 10 Pro OS.

2) Serverless Environment Configuration: To test the proxy module of the CAPTAIN framework, the environment parameters of the serverless instance are as follows: *Platform:* Google Cloud Functions; *Region:* Europe-west2b; and *Runtime:* Python 3.11, *RAM:* 256 MB.

3) AI Model Parameters: AI models and their parameter values used in cold start prediction are shown in Table III. These values were obtained using various parameter optimization methods, aiming for the best cold start prediction performance.

4) Distribution Cost: The distribution cost of the CAPTAIN framework is considered from three aspects.

1) *PING Cost:* The AI module of the CAPTAIN framework predicts the time of the request to the server and makes the container on the server operational by starting it with the PING mechanism (HTTP request). Let's assume that the AI module's prediction accuracy is 3%, which means 30 wrong predictions for every 1000 requests. This means that containers are started unnecessarily 30 times. For this purpose, the following formulation can be used when calculating the PING cost (per 1000 requests)

$$\text{Cost}_{\text{PING}} : \text{Cost}_{\text{Container}} \times 30 \tag{2}$$

where $\text{Cost}_{\text{Container}}$ represents the cost of running the container for every 1000 requests. This error rate is at a level that can be ignored when calculating the cost in large serverless systems. In addition, it should be noted that the running fee of the container will vary depending on the cloud functions environment parameters used by the customer using the CAPTAIN system.

2) *CAPTAIN Proxy Module Cost:* We deployed the CAPTAIN framework on a VM instance on the GCP Compute Engine, which can scale this VM instance if needed. The number of VMs can be automatically increased or decreased to balance the workload. The cost of the proxy module will vary depending on the environment parameters of the VM instance used. In this article, the instance environment parameters where the proxy module is deployed are as follows: Region: "Europe-west2 (London)," Zone: "Europe-west2-a," Series: "T2D (Scale-out workloads)," vCPUs: "1 -60," Memory: "4 - 240 GB," Platform: "AMD EPYC Milan," and Monthly Cost: $41.77. Additionally, we mentioned that the Rabbit MQ protocol is used to record transaction information in the proxy module. Since Rabbit MQ is not supported free of charge by GCP, the Rabbit MQ server is also created by using another VM instance. The VM environment parameters used are e2-micro (2vCPU, 1core, 1GB memory) and the monthly cost is $7.11.

3) *CAPTAIN AI Module Cost:* We deploy the CAPTAIN AI module to a separate VM instance with different environment parameters than the proxy module. Because the situations in which the AI module and the proxy module need to be scaled may be different. To explain as an example, let's consider a scenario where the AI module should scale and the proxy module should not. In this case, it is sufficient to scale only the lower paid instance where the AI module is located, which will reduce the total cost spent on the CAPTAIN framework. The environment parameters for the VM used for the AI module are as follows: Region: "us-central1 (Iowa)," Zone: "us-central1-a," Series: "T2A (Scale-out workloads)," vCPUs: "1-48," Memory: "4 - 192 GB," Platform: "Ampere Altra Arm," and Monthly Cost: $29.10.

B. Evaluation Metrics

We have evaluated the performance of the CAPTAIN framework using time series performance metrics, such as mean squared error (MSE), root mean square error (RMSE), mean absolute error (MAE), and R^2 score as well as QoS metrics, such as throughput, cold start latency average response rate (ARR), and energy consumption and carbon emissions. Our previous works [17], [18], [19], [45] provide the details of time series performance and QoS metrics.

C. Workloads

In this article, various workloads were created to test the proxy and AI modules.

Predictive Maintenance Dataset Trace: First, an IIoT application is deployed to the GCP Cloud Functions instance using the environment parameters [19]. To collect traces, varying numbers of simultaneous HTTP requests are generated using Apache JMeter. The range of requests [0, 800] is chosen to encompass scenarios that trigger a cold start, including 1) the arrival of the first request to the server; 2) a new request reaching a container that has been idle for more than 15 min (scaled to zero); and 3) a situation where more than 300 simultaneous requests require the initiation of a new container. These traces are collected continuously from 08:00 to 18:00 over a period of 6 days. The proxy module in the CAPTAIN framework is then tested using this workload.

Microsoft Azure Dataset Trace: The dataset used for cold start prediction experiments in the CAPTAIN framework comes from Azure function calls recorded between 15 July 2019 and 28 July 2019 [25]. It shows that over half of the applications have just one function, and most requests come from applications with up to three functions. There's little correlation between the number of functions and request volume. Half of the functions run in less than 1 s, and 90% of functions have a maximum execution time of 60 s, including cold start delays. This suggests that cold start latency significantly affects function execution time, highlighting the importance of addressing cold starts in serverless systems. Additionally, serverless functions are typically shorter than those in traditional cloud-based workloads [25], so serverless providers need to deploy containers faster.

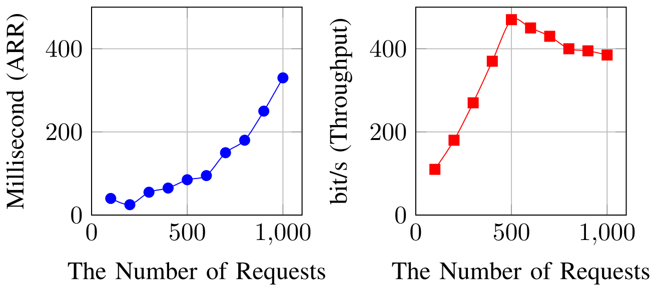
D. Baseline Models

The baselines used to compare the performance of the CAPTAIN framework are as follows:

1) Serverless in the Wild (SWILD) [25]: This resource management policy, known as the Hybrid Histogram Policy, integrates both keep-alive and prewarming techniques to minimize the occurrence of cold starts. It applies a customized keep-alive policy tailored to each workload and uses the ARIMA time-series model to predict optimal prewarming times for containers based on request invocation patterns. As a result, this policy reduces energy consumption compared to the static keep-alive policies commonly employed by cloud service providers.

2) Prediction-Based Autoscaler [26]: PBA utilizes a SARIMA-based time series model to estimate incoming request times. Based on these predictions, it adjusts the number of containerseither increasing or decreasing their countto minimize cold start latency and ensure more efficient resource allocation.

3) Two-Layer Adaptive Approach [27]: This approach employs a two-layer structure to manage container lifecycle and minimize cold starts. In the first layer, an actor-critic algorithm determines the duration for which containers should remain active, thereby keeping them warm for a specified period after function execution to prevent cold starts. In the second layer, the LSTM model predicts function invocation times, and containers are prewarmed based on these predictions to further reduce cold start latency.



(a) (b)

Fig. 5. Performance measurements in terms of latency and throughput while deploying the predictive maintenance dataset. (a) ARR. (b) Throughput.

E. Experimental Results

1) Predictive Maintenance Dataset-Based Experiments: This section discusses the experimental results based on the predictive maintenance dataset.

Serverless Platform Performance: We used the GCP Cloud Functions-based realistic serverless platform for experiments. Serverless computing can dynamically scale resources as needed. We measured the scalability performance using GCP Cloud Functions' QoS parameters, such as throughput and latency, and observed the cold start latency occurring on the instance. Fig. 5 shows the ARR and throughput values that the system displays to an increasing number of concurrent requests. For a description of these two QoS metrics, please see Section IV-B. Fig. 5(a) shows the ARR value of the serverless platform against the increasing number of users. It is normal for the ARR to increase as the concurrent request increases. However, upon careful inspection, the ARR for 100 concurrent requests is higher than the ARR for 200 concurrent requests. This is because of the cold start originating from the serverless paradigm. The amount of cold start latency can be affected by the amount of RAM memory and the software language used. This latency can be a problem for time-sensitive AIoT applications [17]. Fig. 5(b) shows the throughput value of the serverless platform in response to the increasing number of users. As the number of users increases, the throughput will also increase. The highest throughput value was achieved when 500 simultaneous requests were sent, and after that point, the throughput gradually decreased. This is due to resource contention on the server, such as shared memory and storage [46].

Proxy Module Performance: We test the CAPTAIN framework proxy module performance from two different aspects. i) We test whether there is an external delay caused by the proxy module positioned between the client and the server. Doing this is important because external latency from the proxy module can compromise the consistency of this article's destination. ii) We test to what extent the proxy module correctly monitors a communication network. To do this, we will compare by recreating the dataset previously created in another study [19]. It has been mentioned before that the proxy module in the CAPTAIN framework is placed between the client-server and monitors the transaction information. The most crucial point to be considered while creating the proxy

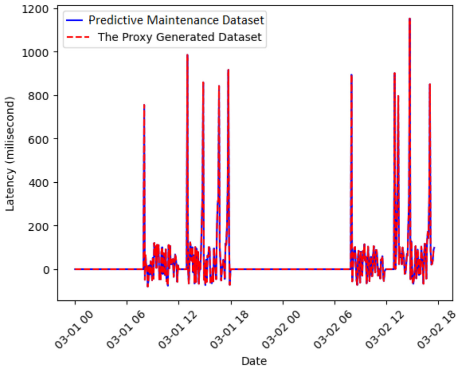


Fig. 6. Latency variation in the dataset produced by the proxy module.

TABLE IV
External Latency Caused by the Proxy Module

Request Queue	Client-Proxy	Proxy-Server
1	843 sec	840 sec
2	22 mSec	18 mSec
3	19 mSec	16 mSec
4	26 mSec	24 mSec
5	29 mSec	27 mSec
6	30 mSec	27 mSec
7	25 mSec	23 mSec
8	28 mSec	25 mSec
9	33 mSec	30 mSec
10	26 mSec	24 mSec

module is the external latency that may arise from the proxy. In order to create as little latency as possible in the framework, the proxy module has been created using the Go language. The external latency is calculated with POSTMAN [47]. Table IV shows the latency times between client-proxy and proxy-server. According to these results, the latency between client and server is twenty-four milliseconds on average (excluding the first request originating from a cold start), while external latency originating from the proxy module is only 3 ms. This amount guarantees that the latency time that may occur due to the proxy is negligible.

Using the CAPTAIN framework, we reproduce the predictive maintenance dataset in [18]. Our purpose in doing this is to determine how accurately the proxy module monitors client-server communication and creates a dataset. First, a GCP instance was created using all environment parameters. The workload has been created as described in Section III-D using JMeter. The resulting dataset and the original dataset are compared in Fig. 6, which shows the proxy module obtained values very close to the values in the original predictive maintenance. The reasons why there are small differences between the two datasets are as follows.

1) *Network Effect:* The request sent to the Server is executed in a different data center in the cloud provider.
2) *Environmental Factors:* The request sent to the Server is executed in different physical resources. But in general, this difference is negligible and the results prove that the proxy module is working correctly.

2) Microsoft Azure Dataset-Based Experiments: We evaluate the CAPTAIN and the baselines in terms of computational time, energy consumption, and CO_2 emissions to draw attention to environmental sustainability and carbon footprint awareness. Fig. 7 shows the computation time, energy, and CO_2 amounts for CAPTAIN and other frameworks, respectively. While the model with the lowest values for all three metrics is LGBM and RF, the model with the highest values is LSTM. The reason for this is that LSTM contains multiple layers and many neural networks, and it takes more input values than ML models. For this reason, it will cause longer computation time and subsequently higher energy consumption and CO_2 emissions. The results show that CAPTAIN has a higher latency prediction rate and less energy and CO_2 emissions compared to other baselines.

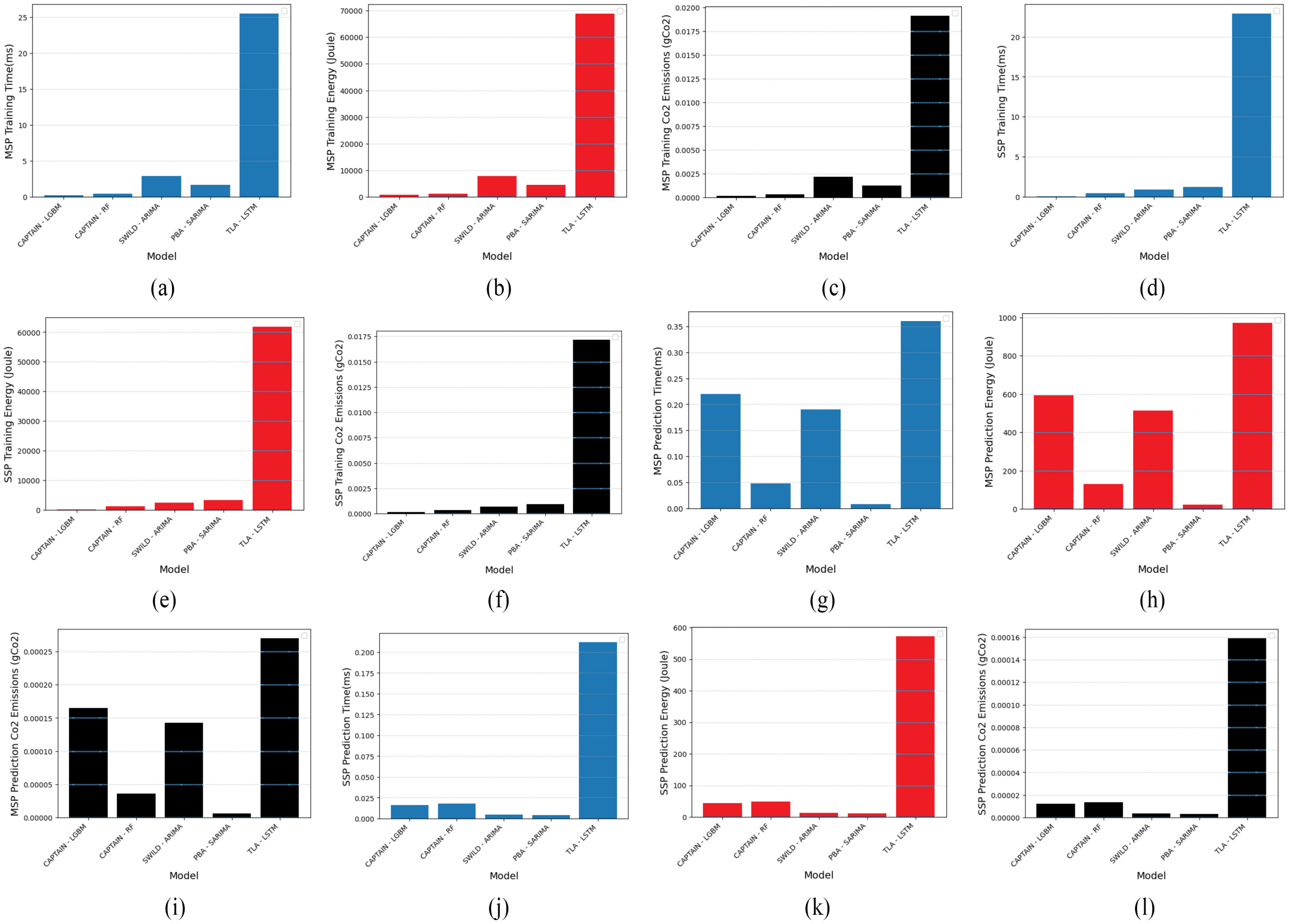
TABLE V
Performance Comparison of CAPTAIN With Baselines Regarding Cold Start MSP and SSP Using the Azure Dataset

MSP PREDICTION					
Work	**Model**	**MAE**	**MAPE**	**RMSE**	**MSE**
SWILD [25]	ARIMA	0.640	1.77	1.051	1.108
PBA [26]	SARIMA	0.628	1.66	1.052	1.109
TLA [27]	LSTM	0.644	1.71	1.060	1.137
CAPTAIN	LGBM	0.464	1.30	0.710	0.586
	RF	**0.414**	**1.33**	**0.616**	**0.468**
SSP PREDICTION					
Work	**Model**	**MAE**	**MAPE**	**RMSE**	**MSE**
SWILD [25]	ARIMA	2.226	0.83	2.226	4.966
PBA [26]	SARIMA	2.310	0.86	2.309	5.348
TLA [27]	LSTM	2.300	0.86	2.304	5.324
CAPTAIN	**LGBM**	**1.190**	**0.44**	**1.190**	**1.974**
	RF	1.240	0.46	1.240	2.202

F. AI Module Performance Comparison

This section initially determines the AI model for the CAPTAIN framework. Teoh et al. [9] used logistic regression (LR) for predictive maintenance, and found that this model performs well when implemented in integrated IoT and fog computing environments. However, for serverless computing environments, RF provides better performance than LR, as reported in our previous work [17]. So, in the CAPTAIN framework, we selected RF and LGBM models due to their robust learning techniques and nonlinear structures, which make them more effective than LR. The LR model tends to be inadequate for handling complex datasets, such as those used in predictive maintenance, where RF and LGBM offer superior performance. To do this, we compare the cold start prediction performance of LGBM and RF models, which show high performance in time-series prediction problems, using all datasets. As a result, we deploy the most successful model to the AI module and compare CAPTAIN and three baselines [25], [26], [27] using the same datasets. In the second part, we deploy the LGBM model, which was found to make the most effective cold start prediction according to the results of the previous experiment, to the AI model in the CAPTAIN framework. Then, we test the success of preventing cold start by deploying it to a real serverless environment.

1) Results Based on the Microsoft Azure Dataset: Table V provides cold start prediction performance comparisons for time series models trained using Microsoft Azure datasets. Since we are interested in outliers, such as cold starts in datasets, we evaluate models using MAE. Fig. 8 shows the performance comparison of multistep prediction (MSP) and single-step prediction (SSP) cold start latency prediction for proposed (CAPTAIN) and baseline frameworks [SWILD, PBA, and two-layer adaptive approach (TLA)] using the Microsoft Azure dataset. As a result of the experiments

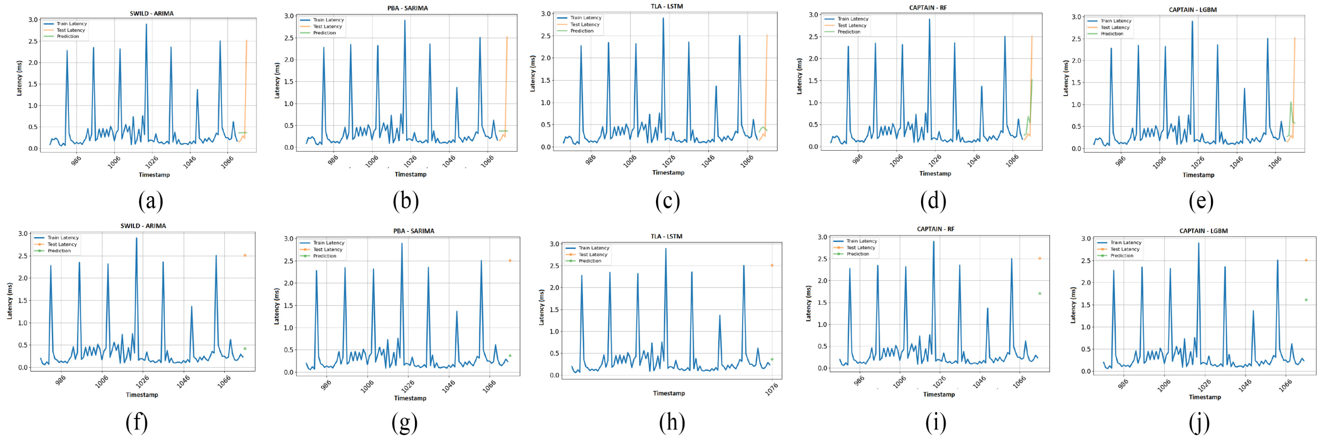


(a) (b) (c) (d)

(e) (f) (g) (h)

(i) (j) (k) (l)

Fig. 7. Performance comparison in terms of computational time, energy consumption, and CO_2 emissions for CAPTAIN and baseline frameworks (SWILD, PBA, and TLA) using Microsoft Azure dataset. (a) MSP training (time). (b) MSP training (energy). (c) MSP training (Co2). (d) SSP training (time). (e) SSP training (energy). (f) SSP training (Co2). (g) MSP prediction (time). (h) MSP (energy). (i) MSP prediction (Co2). (j) SSP prediction (time). (k) SSP (energy). (l) SSP prediction (Co2).



(a) (b) (c) (d) (e)

(f) (g) (h) (i) (j)

Fig. 8. Performance comparison of MSP and SSP cold start latency prediction for proposed (CAPTAIN) and baseline frameworks (SWILD, PBA, and TLA) using Microsoft Azure dataset. (a) SWILD-ARIMA (MSP). (b) PBA-SARIMA (MSP). (c) TLA-LSTM (MSP). (d) CAPTAIN-RF (MSP). (e) CAPTAIN-LG (MSP). (f) SWILD-ARIMA (SSP). (g) PBA-SARIMA (SSP). (h) TLA-LSTM (SSP). (i) CAPTAIN-RF (SSP). (j) CAPTAIN-LG (SSP).

conducted for The Azure Dataset, the most successful models are RF with 0.414 MAE in MSP and LGBM with 1.190 MAE in SSP. The models with the lowest cold start prediction performance are LSTM with 0.644 MAE in MSP and SARIMA with 2.310 MAE in SSP. Fig. 9 shows the cold start prediction results for Microsoft Azure Dataset. The results show that the most successful model in MSP is RF, and the most successful model in SSP is LGBM.

2) Results Based on the Predictive Maintenance Dataset: Table VI provides cold start prediction performance comparisons for time series models trained using predictive maintenance datasets. Since we are interested in outliers, such

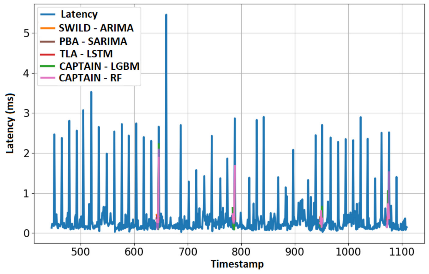


Fig. 9. Cold start prediction results for CAPTAIN and the baselines in Azure dataset.

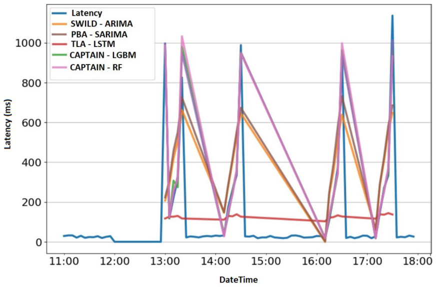


Fig. 10. Cold start prediction results for CAPTAIN and the baselines in predictive maintenance dataset.

as cold starts in datasets, we evaluate models using MAE. It was the most successful model in predicting cold start with RF with 26.2 MAE for MSP and LGBM with 83.8 MAE for SSP for the predictive maintenance dataset. Likewise, this dataset's least successful prediction model is the LSTM model with 294.1 MAE for MSP and 843.2 MAE for SSP. Fig. 10 shows the cold start prediction results for the Predictive Maintenance dataset. Likewise the result of Microsoft Azure, the results show that the most successful model in MSP is RF, and the most successful model in SSP is LGBM. This is because different algorithms are used for both models. While RF uses a decision tree algorithm, the LGBM model uses a gradient boosting-based algorithm. The decision tree algorithm uses multiple trees for each step estimate in MSP predictions to provide higher performance in longer predictions. Moreover, this algorithm effectively captures complex patterns between input features and target variables to gain significant advantages in MSP prediction. Gradient boosting algorithm, on the other hand, can increase performance in successive time steps, and this feature provides a great advantage for SSP where the next time step is predicted. Additionally, the gradient boosting algorithm can improve prediction performance for future predictions by learning from the prediction errors of past steps. It can be seen that the most unsuccessful models in cold start latency prediction are LSTM, SARIMA, and ARIMA, respectively. This is because the LSTM model is a DL-based time-series model and therefore data dependent. This means that LSTM will fail on datasets that are small and have complex patterns, such as cold start. Likewise, ML-based classical time-series models, such as ARIMA and SARIMA, work with low performance on datasets with complex patterns. This is because both models cannot capture nonlinear patterns well enough as they are modeled under the assumption that the data are stationary. In both datasets, RF yields the highest performance rate in MSP, while LGBM time-series models yield the highest performance rate in SSP. Additionally, the CAPTAIN framework has a higher performance rate in predicting cold starts than other baseline works.

TABLE VI
PERFORMANCE COMPARISON OF CAPTAIN FRAMEWORK AND BASELINES REGARDING COLD START MSP AND SSP PERFORMANCES USING THE PREDICTIVE MAINTENANCE DATASET

MSP PREDICTION					
Work	**Model**	**MAE**	**MAPE**	**RMSE**	**MSE**
SWILD [25]	ARIMA	206.8	0.82	254.12	71388
PBA [26]	SARIMA	211.6	0.88	250.83	70103
TLA [27]	LSTM	294.1	0.97	441.66	197460
CAPTAIN	LGBM	32.8	0.10	50.82	3504
	RF	**26.2**	**0.09**	**38.19**	**1835**
SSP PREDICTION					
Work	**Model**	**MAE**	**MAPE**	**RMSE**	**MSE**
SWILD [25]	ARIMA	310.8	0.31	310.80	112642
PBA [26]	SARIMA	209.8	0.20	209.80	60510
TLA [27]	LSTM	843.2	0.86	843.25	722880
CAPTAIN	**LGBM**	**83.8**	**0.09**	**83.75**	**11818**
	RF	108.3	0.11	108.25	14860

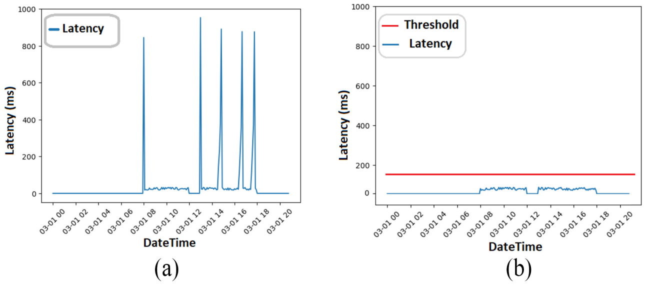


Fig. 11. Cold start prevention performance before and after applying the CAPTAIN framework. (a) Before applying CAPTAIN. (b) After applying CAPTAIN.

G. Cold Start Prevention Performance

In line with the results obtained from the experiments, it was decided to use the model RF with the highest performance in MSP for the CAPTAIN AI module. This is because MSP forecasts are more accurate and reliable and can provide long-term forecasting. In line with the prediction results, the PING mechanism is informed approximately 20 min before the cold start occurs, and thus, the necessary precautions are taken to prevent the occurrence of a cold start. Fig. 11 shows the CAPTAIN framework's success in preventing cold start. This result was achieved by deploying CAPTAIN in a real serverless environment using the parameters previously described in Section IV-A. The threshold value shown with the red line was determined as 180 ms after calculating the variance value of the dataset. This means that if the latency prediction result in the CAPTAIN AI module is 180 and above, a PING will be sent to the server by the proxy module.

V. CONCLUSION AND FUTURE WORK

This article presents the new testbed, CAPTAIN, using time-series models, such as LGBM and RF, to solve the cold start problem in the serverless paradigm. The CAPTAIN framework is capable of predicting possible cold start times and preventing their occurrence by pinging the server based on prediction results. We used predictive maintenance and the

Microsoft Azure dataset to test the CAPTAIN framework's performance in GCP-based realistic serverless computing environments for both single-step and multiple-step prediction operations. In both datasets, the most successful ML models were LGBM in SSP and RF in MSP. We evaluated the serverless computing platform's performance in terms of QoS parameters, such as latency and throughput. Finally, the performance of the CAPTAIN framework is compared with three baselines using cold start frequency, computational time, energy consumption, and CO_2 emissions. Experimental results demonstrate that the AI module in CAPTAIN framework outperforms in predicting cold start latency as well as optimizing cold start frequency as compared to state-of-the-art frameworks. Future research can be undertaken to deepen these understandings in three areas: 1) incorporating cutting-edge AI models like Generative AI; 2) evaluating AI models for load distribution in the context of IIoT applications; and 3) expanding the CAPTAIN framework to accommodate a continuum of cloud-to-edge nodes for a wider variety of AIoT scenarios. To increase the CAPTAIN framework's potential, we will investigate alternative dynamic approaches like load sharing and resource consolidation, as well as consider multiple cloud providers.

Software Availability

We released CAPTAIN Testbed as an open source software. The implementation code, datasets, result reproducibility scripts are publicly available as part of a GitHub repository under CC-BY License and can be found at the GitHub repository: https://github.com/MuhammedGolec/CAPTAIN.

References

[1] Q. Shi, Z. Zhang, Y. Yang, X. Shan, B. Salam, and C. Lee, "Artificial intelligence of things (AIoT) enabled floor monitoring system for smart home applications," *ACS Nano*, vol. 15, no. 11, pp. 18312–18326, 2021.

[2] Q. Tang et al., "Distributed task scheduling in serverless edge computing networks for the Internet of Things: A learning approach," *IEEE Internet Things J.*, vol. 9, no. 20, pp. 19634–19648, Oct. 2022.

[3] H. Ko, S. Pack, and V. C. M. Leung, "Performance optimization of serverless computing for latency-guaranteed and energy-efficient task offloading in energy-harvesting industrial IoT," *IEEE Internet Things J.*, vol. 10, no. 3, pp. 1897–1907, Feb. 2023.

[4] G. M. Sang, L. Xu, and P. de Vrieze, "A predictive maintenance model for flexible manufacturing in the context of industry 4.0," *Front. Big Data*, vol. 4, Aug. 2021, Art. no. 663466.

[5] M. Golec, S. S. Gill, A. K. Parlikad, and S. Uhlig, "HealthFaaS: AI based smart healthcare system for heart patients using serverless computing," *IEEE Internet Things J.*, vol. 10, no. 21, pp. 18469–18476, Nov. 2023.

[6] B. Ali, M. Golec, S. S. Gill, H. Wu, F. Cuadrado, and S. Uhlig, "EdgeBus: Co-simulation based resource management for heterogeneous mobile edge computing environments," *Internet Things*, vol. 28, Dec. 2024, Art. no. 101368.

[7] H. Gao, Y. Sun, J. Xiao, D. Fang, Y. Xu, and W. Wei, "Toward effective 3D object detection via multimodal fusion to automatic driving for industrial cyber–physical systems," *IEEE Trans. Ind. Cyber–Phys. Syst.*, vol. 2, pp. 281–291, Jul. 2024.

[8] H. Gao, S. Wu, Y. Wang, J. Y. Kim, and Y. Xu, "FSOD4RSI: Few-shot object detection for remote sensing images via features aggregation and scale attention," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 17, pp. 4784–4796, Feb. 2024.

[9] Y. K. Teoh, S. S. Gill, and A. K. Parlikad, "IoT and fog-computing-based predictive maintenance model for effective asset management in industry 4.0 using machine learning," *IEEE Internet Things J.*, vol. 10, no. 3, pp. 2087–2094, Feb. 2023.

[10] P. Vahidinia, B. Farahani, and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," in *Proc. Int. Conf. Omni-Layer Intell. Syst. (COINS)*, 2020, pp. 1–7.

[11] S. R. Chaudhry, A. Palade, A. Kazmi, and S. Clarke, "Improved QoS at the edge using serverless computing to deploy virtual network functions," *IEEE Internet Things J.*, vol. 7, no. 10, pp. 10673–10683, Oct. 2020.

[12] H. Gao, J. Shao, M. Iqbal, Y. Wang, and Z. Xiang, "CFPC: The curbed fake point collector to pseudo-LiDAR-based 3D object detection for autonomous vehicles," *IEEE Trans. Veh. Technol.*, early access, Mar. 12, 2024, doi: 10.1109/TVT.2024.3372940.

[13] H. Ko, H. Jeong, D. Jung, and S. Pack, "Dynamic split computing framework in distributed serverless edge clouds," *IEEE Internet Things J.*, vol. 11, no. 8, pp. 14523–14531, Apr. 2024.

[14] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A reinforcement learning approach to reduce serverless function cold start frequency," in *Proc. IEEE/ACM 21st Int. Symp. Cluster, Cloud Internet Comput. (CCGrid)*, 2021, pp. 797–803.

[15] I. Pelle, J. Czentye, J. Dóka, A. Kern, B. P. Gerő, and B. Sonkoly, "Operating latency sensitive applications on public Serverless edge cloud platforms," *IEEE Internet Things J.*, vol. 8, no. 10, pp. 7954–7972, May 2021.

[16] M. S. Aslanpour et al., "Serverless edge computing: Vision and challenges," in *Proc. Aust. Comput. Sci. Week Multiconf.*, 2021, pp. 1–10.

[17] M. Golec, D. Chowdhury, S. Jaglan, S. S. Gill, and S. Uhlig, "AIBLOCK: Blockchain based lightweight framework for serverless computing using AI," in *Proc. 22nd IEEE Int. Symp. Cluster, Cloud Internet Comput. (CCGrid)*, 2022, pp. 886–892.

[18] M. Golec et al., "ATOM: AI-powered sustainable resource management for serverless edge computing environments," *IEEE Trans. Sustain. Comput.*, early access, Dec. 29, 2023, doi: 10.1109/TSUSC.2023.3348157.

[19] M. Golec et al., "MASTER: Machine learning-based cold start latency prediction framework in serverless edge computing environments for industry 4.0," *IEEE J. Select. Areas Sens.*, vol. 1, pp. 36–48, May 2024.

[20] A. K. Mishra and S. Paliwal, "Mitigating cyber threats through integration of feature selection and stacking ensemble learning: The LGBM and random forest intrusion detection perspective," *Clust. Comput.*, vol. 26, no. 4, pp. 2339–2350, 2023.

[21] H. Ko and S. Pack, "Function-aware resource management framework for serverless edge computing," *IEEE Internet Things J.*, vol. 10, no. 2, pp. 1310–1319, Jan. 2023.

[22] M. Golec, G. K. Walia, M. Kumar, F. Cuadrado, S. S. Gill, and S. Uhlig, "Cold start latency in serverless computing: A systematic review, taxonomy, and future directions," 2023, *arXiv:2310.08437*.

[23] S. Akkaya, E. Yüksek, and H. M. Akgün, "A comprehensive research of machine learning algorithms for power quality disturbances classifier based on time-series window," *Elect. Eng.*, vol. 106, pp. 3983–4001, Jan. 2024.

[24] D. Zhang and Y. Gong, "The comparison of LightGBM and XGBoost coupling factor analysis and prediagnosis of acute liver failure," *IEEE Access*, vol. 8, pp. 220990–221003, 2020.

[25] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2020, pp. 205–218.

[26] A. P. Jegannathan, R. Saha, and S. K. Addya, "A time series forecasting approach to minimize cold start time in cloud-serverless platform," in *Proc. IEEE Int. Black Sea Conf. Commun. Netw. (BlackSeaCom)*, 2022, pp. 325–330.

[27] A. Kumari, B. Sahoo, and R. K. Behera, "Mitigating cold-start delay using warm-start containers in serverless platform," in *Proc. IEEE 19th India Council Int. Conf. (INDICON)*, 2022, pp. 1–6.

[28] S. Lee, D. Yoon, S. Yeo, and S. Oh "Mitigating cold start problem in serverless computing with function fusion," *Sensors*, vol. 21, no. 24, p. 8416, 2021.

[29] S. Wu et al., "Container lifecycle-aware scheduling for serverless computing," *Softw., Pract. Exp.*, vol. 52, no. 2, pp. 337–352, 2022.

[30] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," 2019, *arXiv:1903.12221*.

[31] X. Liu et al., “FaaSLight: General application-level cold-start latency optimization for function-as-a-service in serverless computing,” *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, pp. 1–29, 2023.

[32] S. K. Battula, M. M. O’Reilly, S. Garg, and J. Montgomery, “A generic stochastic model for resource availability in fog computing environments,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 4, pp. 960–974, Apr. 2021.

[33] M. Gotin, F. Lösch, R. Heinrich, and R. Reussner, “Investigating performance metrics for scaling microservices in cloudiot-environments,” in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2018, pp. 157–167.

[34] M. Golec, S. Iftikhar, P. Prabhakaran, S. S. Gill, and S. Uhlig, “QoS analysis for serverless computing using machine learning,” in *Serverless Computing: Principles and Paradigms*. Cham, Switzerland: Springer, 2023, pp. 175–192.

[35] P. Vahidinia, B. Farahani, and F. S. Aliee, “Mitigating cold start problem in serverless computing: A reinforcement learning approach,” *IEEE Internet Things J.*, vol. 10, no. 5, pp. 3917–3927, Mar. 2023.

[36] A. M. Shilkov. “Cold starts in Google cloud functions.” 2021. [Online]. Available: https://mikhail.io/serverless/coldstarts/gcp/

[37] D. Dossot, *RabbitMQ Essentials*. Birmingham, U.K.: Packt Publ. Ltd, 2014.

[38] “Build fast, reliable, and efficient software at scale.” Accessed: Mar. 22, 2024. [Online]. Available: https://go.dev/

[39] S. Sivaramakrishnan, T. F. Fernandez, R. G. Babukarthik, and S. Premalatha, “Forecasting time series data using ARIMA and Facebook prophet models,” in *Big Data Management Sensing, Applications in AI and IoT*. Denmark, U.K.: River Publ., 2022, p. 47.

[40] U. H. Rahman et al., “State of art of SARIMA model in second wave on COVID-19 in India,” *Int. J. Agric. Stat. Sci.*, vol. 18, no. 1, pp. 141–152, 2022.

[41] S. Chaturvedi, E. Rajasekar, S. Natarajan, and N. McCullen, “A comparative assessment of SARIMA, LSTM RNN and Fb prophet models to forecast total and peak monthly energy demand for India,” *Energy Policy*, vol. 168, Sep. 2022, Art. no. 113097.

[42] M. Massaoudi, S. S. Refaat, I. Chihi, M. Trabelsi, F. S. Oueslati, and H. Abu-Rub, “A novel stacked generalization ensemble-based hybrid LGBM-XGB-MLP model for short-term load forecasting,” *Energy*, vol. 214, Jan. 2021, Art. no. 118874.

[43] A. Lahouar and J. B. H. Slama, “Day-ahead load forecast using random forest and expert input selection,” *Energy Convers. Manag.*, vol. 103, pp. 1040–1051, Oct. 2015.

[44] Y. Zhang et al., “Faster and cheaper serverless computing on harvested resources,” in *Proc. ACM SIGOPS 28th Symp. Oper. Syst. Princ.*, 2021, pp. 724–739.

[45] M. Golec, R. Ozturac, Z. Pooranian, S. S. Gill, and R. Buyya, “IFaaSBus: A security-and privacy-based lightweight framework for serverless computing using IoT and machine learning,” *IEEE Trans. Ind. Informat.*, vol. 18, no. 5, pp. 3522–3529, May 2022.

[46] X. Han, R. Schooley, D. Mackenzie, O. David, and W. J. Lloyd, “Characterizing public cloud resource contention to support virtual machine co-residency prediction,” in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, 2020, pp. 162–172.

[47] A. A. Mir et al., “An improved imputation method for accurate prediction of imputed dataset based radon time series,” *IEEE Access*, vol. 10, pp. 20590–20601, 2022.