# An Empirical Study on AI-Powered Edge Computing Architectures for Real-Time IoT Applications

Awatif Yasmin
*Texas State University*
San Marcos, Texas, USA
nuc4@txstate.edu

Tarek Mahmud
*Texas State University*
San Marcos, Texas, USA
tarek_mahmud@txstate.edu

Minakshi Debnath
*Texas State University*
San Marcos, Texas, USA
stg60@txstate.edu

Anne H. H. Ngu
*Texas State University*
San Marcos, Texas, USA
angu@txstate.edu

***Abstract*—AI-Powered Edge Computing is accelerating the integration of the cyber world with the ever-growing list of new physical IoT devices and will fundamentally change and empower the way humans interact with the world. In this paper, we prototyped and analyzed three edge computing architectures for running SmartFall, a real-time fall detection application that uses accelerometer data from the watch, to compare the trade-off in relationship to battery consumption, potential data loss, machine learning model's prediction accuracy, and latency in model inferencing. Our experiments show that running the machine learning prediction on the server using the TensorFlow native model format has achieved the best model accuracy without draining the battery power of the smartwatches. However, the optimal selection of the software architecture depends on the intended deployment environment, projected user numbers, users' privacy concerns, and network stability.**

***Keywords*-Edge service framework, Edge Computing, IoT Applications, Software Architectures, Fall Detection.**

## I. Introduction

Edge computing is becoming increasingly critical in the practical deployment of IoT applications, harnessing data from billions of connected IoT devices [1]. Given the widespread use of IoT applications, the total number of installed IoT devices is expected to reach approximately 41.6 billion in 2023 [2]. Edge computing enables swift decision-making at the device level, boosting efficiency in manufacturing, effective health monitoring, and safe driving of autonomous vehicles. Edge computing conserves network bandwidth by processing data locally, ideal for resource-limited or costly networks. Edge computing bolsters privacy and security by keeping data on local devices, crucial in surveillance, healthcare, and finance industries. Its adaptability extends to a non-exhaustive list of smart everything which includes smart agriculture, smart transportation, smart home, smart power grid, and smart environments to name a few. Edge computing is critical in turning billions of physical IoT devices connected to the internet into awareness and understanding that will enable seamless integration of the physical and cyber world which will fundamentally change and empower human interaction with the world.

AI-Powered edge computing is increasingly playing a vital role in the revolution of the capabilities of IoT devices, particularly in making machine learning models run efficiently, securely, and effectively on them. One noteworthy trend is its support for various learning methods, including ONNX (Open Neural Network Exchange), Edge AI Accelerators, Deep Learning, and Federated Learning. Recently, there has been a big increase in the use of the Deep Learning model on edge devices compared to traditional statistical learning methods. This is because Deep Learning-based models, once trained offline, require fewer computational resources during real-time classification tasks compared to traditional statistical learning methods that rely on feature extraction and processing. However, it is known that popular deep learning frameworks such as TensorFlowLite [3] which is optimized for resource-constrained edge devices have reduced accuracy performance when deployed on edge devices. It is important to quantify the impact of the TensorflowLite model when choosing the software architecture for IoT applications.

This paper aims to compare different software architectures for AI-powered edge computing applications, with a particular focus on fall detection using a smartwatch as the sensing device. The current generation of smartwatches are equipped with a variety of sensors such as an accelerometer, gyroscope, barometric pressure, skin temperature, ambient light, sound, heart rate, blood oxygen, galvanic skin response, location etc, but for this study, we solely concentrate on the application that make use of accelerometer data. We prototyped and analyzed three edge computing architectures for running SmartFall, an AI-powered fall detection IoT application [4], [5] to compare the trade-off in relationship on battery consumption, data loss, machine learning model's accuracy, and latency in model inferencing.

The first architecture comprises the most common three-layer structure: a smartwatch at the edge, a smartphone in the middle, and a cloud server. The smartwatch captures sensor data and transmits it to the smartphone, where a machine-learning model resides. Predictions are generated on the smartphone and relayed back to the watch for user feedback or additional human input for personalization. Subsequently, the sensed data and user feedback are archived in the cloud server for further analysis to continuously fine-tune the machine learning model. Since prediction happens on the edge device,

i.e. the phone, the TensorflowLite model is used.

The second architecture is watch-based, with the smartwatch responsible for data sensing and local machine-learning model predictions. The sensed data, along with user feedback, is then transmitted to a cloud server for storage and further analysis periodically. In this configuration, the smartphone is primarily used for the one-time user profile setup. Prediction happens on the watch which is a very computational resource-constrained edge device and only the TensdorflowLite model can be used.

The final architecture is server-based, where the smartwatch (an edge device) senses the data and sends it directly to the cloud server. The machine learning model resides in the cloud server and produces predictions on the server. The server subsequently sends the prediction result to the watch for user notification or interaction. The smartphone is again used mainly for the one-time setup of the user profile. The prediction happens in the server and the native TensorFlow model is used.

In general, developing an AI-powered edge computing application involves the following key challenges which we observed from implementing our Fall Detection IoT application: 1) The smartwatch (IoT device) must first be virtualized as a software component and able to obtain data from all available sensors at a specified sampling rate on the device reliability (i.e. with no data loss). 2) The real-time interaction between the application and the physical device must be supported. Data is usually delivered as infinite streams in time-stamped order from the devices. Deep learning inference over a stream is an important component. Stream processing provides complex event detection that turns the collected data (usually in large quantity) into useful actionable information. For example, in the case of sensing accelerometer data for fall detection, the data is processed in an overlapping sliding windows (streams) manner, and the prediction is aggregated over multiple windows for accurate prediction (ie. model accuracy) 3) A monitoring or visualization service (an UI) is needed to allow users to monitor/control the state of the physical devices in a user-friendly manner. This component should provide a notification for the timely delivery of status to users (low latency), in our case an alert when a fall occurs. The UI should also be able to enable users to give feedback for the fine-tuning of the learning algorithm with minimal input. 4) An IoT application can generate a large amount of data that needs to be further processed for continuous improvement of the application or archival. The ubiquitous connection to a cloud infrastructure is needed for further data analytics and archiving. 5) The security and privacy component is needed to provide the integrity of the collected data (stream) and to ensure that the user's privacy is not violated. Users should have the option to archive the collected data in a storage medium of choice or not at all. Users who choose not to send their data to the cloud won't receive the continuous personalized version of the application. 6) The IoT application should be designed to run continuously without draining the battery power of the device to be accepted by any user.

The primary objective of this paper is to conduct a comparative analysis of three software architectures for IoT applications to provide those key features stated above while focusing on analyzing key challenges such as data latency, IoT device battery life, potential data loss, and model accuracy. Currently, there is no standard framework for the development or deployment of IoT applications that allows us to study specific qualities such as latency, model accuracy, privacy, and data transmission. We developed an IoT application (SmartFall) using the native Android framework to enable us the flexibility to benchmark the three different software architectures for making an informed choice in deploying IoT applications with the desired trade-off between model accuracy, battery power, and data loss. We are the first to perform benchmarking of the various software architectures for IoT applications. The main contributions of this paper are:

- We designed and implemented three distinct architectures, each with its unique approach to running fall detection application: Watch-and-Phone-based Architecture (WPA), Watch-based Architecture (WA), and Server-based Architecture (SA).
- We systematically compare these architectures against pivotal edge-computing challenges such as IoT device battery life, data latency, potential data loss, and model accuracy.
- We discuss the inherent trade-offs involved in edge computing. For example, while edge devices provide benefits like reduced latency, they might sacrifice model accuracy.

The remainder of this paper is organized as follows. The closely related works are detailed in Section II. Section III presents the SmartFall system and the implementation details of three different architectures. Section IV presents the experimental setup used to compare these architectures followed by the results and analysis in Section V. Section VI discusses the pros and cons of each architecture and a future direction. Finally, we concluded our work in Section VII.

## II. Related Work

The AI-powered edge computing which we defined as synonymous with the Internet of Things (IoT) is a domain that represents the next most exciting technological revolution since the Internet [2], [6]–[8]. IoT will bring endless opportunities and impact in every corner of our planet. IoT can transform manufacturing, making it leaner and smarter. Real-time IoT applications are going to create massive disruption and innovation in just about every industry segment imaginable.

While there is an explosive number of potential IoT applications that can be built and deployed, currently there are no standard development and deployment strategies for real-time IoT applications.

Ngu et al. [9] studied IoT middleware, the vital link between IoT devices and applications. They highlighted its importance through a real-time blood alcohol content prediction app utilizing smartwatch sensor data [10]. An ideal IoT middleware must support heterogeneous IoT devices from multiple vendors to build real-time IoT applications with on the edge analysis and data storage yet without dictating a particular

communication protocol or be dependent on specific cloud vendor. The availability of edge/local storage is important to avoid the unpredictable latency from wireless transmission of data to the cloud or server for real-time analytics. In addition, to ensure that users' privacy is not violated, users should have the option to archive data generated from their personal IoT devices in a secure local storage of their own choice.

Ngu et al. [11] proposed and implemented an open-source, edge IoT service framework known as Cordova Accessor Host. This framework enables IoT app development with accessors as fundamental elements, showcasing flexibility by gathering sensor data from various devices like smartwatches and Arduino microcontrollers using a unified accessor. Additionally, it demonstrates reduced battery consumption compared to native Android frameworks. While promising, Cordova Accessor Host is still in its research prototype phase, solely tested on Android devices, and not yet suitable for real-time IoT applications in the real world.

In recent years, a common software development framework for IoT application development used in the industry is the various cloud-based frameworks such as AWS IoT from Amazon [12], Watson IoT from IBM [13], ThingSpeak IoT [14] and Google IoT Cloud [15]. These cloud-based frameworks usually provide the following four fundamental services: 1) Web-based administrative console for managing device connection; 2) cloud-based data storage; 3) cloud-based analytic services, and 4) advanced reporting or visualization. We examined GoogleFit cloud service in detail for IoT application development. GoogleFit provides a set of APIs for connecting third-party IoT devices to its cloud storage. For example, it provides APIs for subscribing to a particular fitness data type or a particular fitness source (e.g., Fitbit or Samsung Smartwatch) and APIs for querying of historical data or persistent recording of the sensor data from a particular source (e.g., a smartwatch). With GoogleFit, the user is tied to storing his/her sensor data in GoogleFit's cloud storage, in the format dictated by GoogleFit and in the size limit enforced by GoogleFit. It is not possible to get access to the collected raw data and pre-process them for analysis and visualization using your own custom-designed analytics which is a critical component for many IoT applications. Moreover, GoogleFit requires all collected data to be stored remotely in the Google Cloud which might not be suitable for real-time fall detection that must be performed quickly in real-time on board the edge device.

Another emerging trend in IoT application development is commercial edge-based computing frameworks that leverage container technology. For instance, Microsoft Azure IoT Edge [16] consists of three components: IoT Edge modules, IoT Edge runtime, and a cloud-based interface. The first two components run on edge devices and the cloud-based interface allows remote monitoring of edge devices. The Azure IoT Edge runtime leverages Docker to run IoT Edge modules on the device with the embedded instructions on what modules to download and run via a connection to Microsoft Azure IoT Hub. The current Azure IoT Edge runtime engine only supports Windows and Linux systems, which means an IoT application developed using Azure's cloud cannot be deployed on popular edge devices that run Android, iOS, or WearOS. Azure IoT Edge is thus a cloud-based IoT computing framework like GoodgleFit. The developer is further constrained by the cost of subscribing to the cloud and the type of APIs provided by the cloud's vendor. If an IoT device is not supported by Azure IoT Edge runtime, then it is not possible to use that as a sensing device in this framework.

KubeEdge is the most well-known open-source container-based edge computing framework [17]. It is part of the recent new generation of Cloud-to-Edge infrastructure that is known for its ability to scale out to a large number of devices and the support for security and fault tolerance. KubEdge leverages container technology to bring native cloud capability to the edge. It consists of a cloud part and an edge part. While the cloud part interfaces with Kubernetes' APIs and take care of node management, the edge part has control of container deployment on the edge and provides an infrastructure for storage as well as event-based communication based on MQTT [18]. However, it has been reported in [17] that Kubernetes deployment leads to several performance and stability problems on some low memory edge devices (e.g., Raspberry Pi 3). Moreover, Kubernetes frameworks are not currently available on Android, iOS, or WearOS.

A deep learning model is critical for many practical IoT applications. Despite deep learning models being large and resource-intensive, they need to be run on an edge device to be effective. Google has expanded data processing and machine learning capabilities to edge devices, harnessing TensorFlow Lite and Edge Tensor Processing Unit (TPU) [19]. However, no systematic study has been performed on the impact of edge-based learning model prediction accuracy on a practical IoT application.

Despite the myriad of edge and cloud computing technologies and software architectures, there is currently no study performed on different AI-powered edge computing software architectures that discuss the user's need for a practical IoT application in terms of model accuracy, low data latency, user privacy, and low battery power consumption. We develop and deploy our SmartFall App using the native Android framework that offers the flexibility to benchmark different software architectures to be able to make an informed decision regarding deploying a robust practical real-time IoT application.

## III. SmartFall System Architecture

We implemented three variants of the generic three-layered edge-cloud collaborative IoT architecture. For this study, we used the SmartFall system [20], [21] that collects accelerometer data from wearables for real-time fall detection. The Watch-and-Phone-based Architecture is adopted from Ngu et al. [21]. We explored the capabilities of the new smartwatches in the Watch-based Architecture and the Server-based Architecture is used in many popular AI-powered edge computing studies, e.g. Google's fall detection in mobile devices [22].

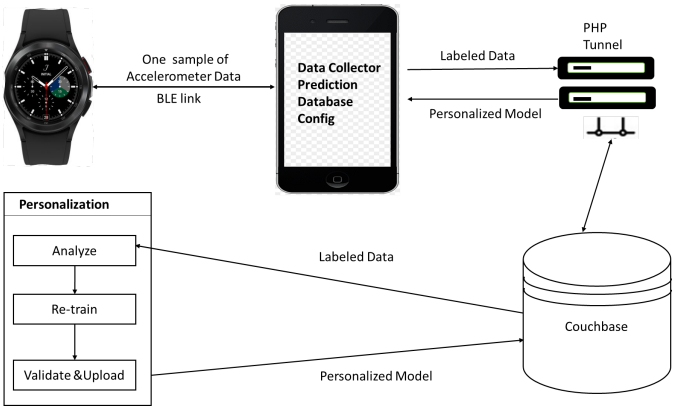


Fig. 1. SmartFall system with Watch-Phone-based Architecture.

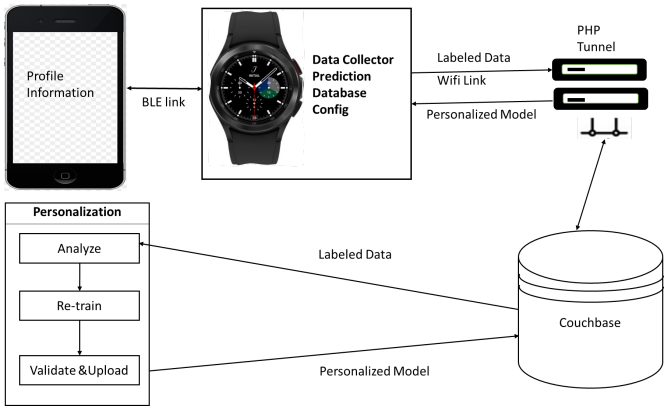


Fig. 2. SmartFall system with Watch-based Architecture.

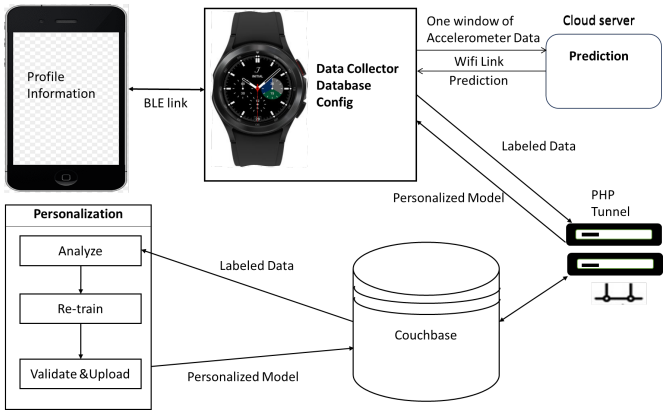


Fig. 3. SmartFall system with Server-based Architecture.

### A. Overview of the SmartFall System

The SmartFall system is an AI-powered edge computing system designed to address the crucial need for efficient fall detection among older adults using a commodity-based IoT wearable device. Central to its design is the user-centric approach that primarily interacts through a smartwatch interface, taking into account the challenges older adults face in handling multiple devices and technologies. Moreover, when an older adult falls, in a moment of panic, it might be difficult to locate the phone in order to interact with the App for the purpose of indicating whether they are fine or need help if the App is running on the phone. A watch's UI, on the other hand, would allow interaction with the SmartFall App at any time and anywhere. The system has been structured to incorporate several vital software components, including:

- **Config Module:** This module manages the version of the deep learning model, personalization training strategy, and the selection of the cloud server, tailoring the system to individual user needs.
- **Database Module:** This module handles the collection, storage, and cloud interactions of sensed data. It also fetches the most optimized personalized models, ensuring users benefit from the best predictive tools available.
- **Data Collector Module:** This module collects accelerometer data (sensed data) from the smartwatch for other components. It can be configured to handle various communication protocols, ensuring robust data transfer.
- **Prediction Module:** This module can be configured to use different machine learning models (such as ensemble recurrent neural networks (RNN/LSTM), single RNN/LSTM, the transformer model, or the multi-modal transformer model) to detect potential falls, aiming for timely and precise alerts to users.

User interactions mainly occur via the smartwatch in all implementations, adhering to the best design practices like a strict color scheme, legible fonts, and simplicity, tailored for the older adult population. The primary aim of the SmartFall system is to detect falls accurately, minimizing false positives while ensuring no fall goes undetected. This three-layer architecture is one of the most flexible architectures for IoT applications as discussed in [9].

### B. Description of Three Different Implementations

*1) Watch-and-Phone-based Architecture (WPA):* In the Watch-and-Phone-based (WPA) architecture, represented in Figure 1, the smartwatch is placed at the forefront, functioning primarily as the sensing device that collects the accelerometer data. This data is then transmitted to the smartphone. Once received, the smartphone processes the accelerometer data, runs the necessary predictions, and subsequently sends the results back to the smartwatch for user interaction. The cloud server in this architecture acts as a tertiary layer, primarily for storing processed data and potentially other advanced analytics. No user-identifiable information is sent to the cloud.

*2) Watch-based Architecture (WA):* As depicted in Figure 2, the Watch-based Architecture (WA) is characterized by its standalone operation on the smartwatch. Here, the smartwatch not only collects the accelerometer data but also processes it and makes predictions locally without relying on any external devices (such as the phone). The smartphone's role is strictly limited to the initialization phase, mainly for user profile setup. Post-processing, if there's any need for archiving or further analysis, the data is directed to the cloud server.

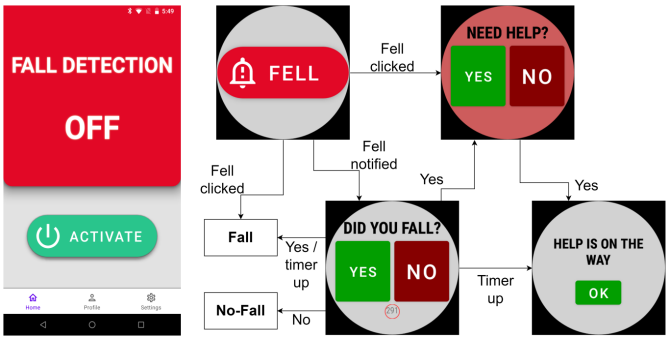


Fig. 4. Phone and Watch's user interface.

*3) Server-based Architecture (SA):* The Server-based Architecture (SA), illustrated in Figure 3, designates the smartwatch as the data collector, gathering the accelerometer data. However, rather than processing and predicting locally as in watch-based architecture, the server-based architecture offloads the data to the cloud server. The server, equipped with more computational power, processes and runs predictions on the data. Once a prediction is made, the result is dispatched back to the smartwatch for the user's attention. In this setup, the smartphone's role remains confined to initial user profile creation. No user-identifiable information is sent to the server.

All user interaction takes place on the smartwatch in all three architectures. The system begins when the user presses the "ACTIVATE" button on the phone UI to set up the profile. After that, all interactions will happen via the smartwatch which displays "Fell" screen on startup. Whenever a fall is detected by the system. The system displays with "Did you Fall?" screen with just the "YES" and "NO" buttons so as to overcome the constraint of small screen space (see Figure 4). If the user answers "NO" to the question "DID YOU FALL?", the data is labeled as a false positive and stored as "FP" in the Couchbase database in the cloud. If the user answers "YES", to this question, the subsequent screen will prompt "NEED HELP?". If the user presses "YES" again, it implies that a true fall is detected and that the user needs help. The collected data will be labeled and stored as "TP" and "HELP IS ON THE WAY" screen will be displayed. If the user presses "NO" on this screen, it suggests that no help is needed and the collected data is still labeled as "TP". The ability to collect and label true positive data (fall events) is very significant. This enables the system to collect real-world fall data just by wearing the watches. If the user does not press either "YES" or "NO" after a specified period of time following the question "DID YOU FALL?", an alert message will be sent out automatically to the designated caregiver.

Our system is structured such that all user-identifying data is only stored locally on the phone to preserve privacy. This information is obtained by prompting the user to set up a profile. Real-time fall prediction can be performed on the phone, on the watch, or on the cloud server depending on the type of chosen IoT software architecture. The main goal of this paper is to evaluate what is the best software architecture to use for deploying such kind of real-world continuous sensing IoT applications in order to minimize battery usage on the devices, prediction accuracy, and latency of prediction. The training/retraining of the prediction model is always performed offline in the cloud server. The UI interface is designed such that there is no need to interact with the App unless the system detects that a fall has occurred, in that case, the watch will vibrate to alert the user that a prediction has occurred and the UI in Figure 4 will appear. The ability to interact with the system when a false or a true prediction is generated allows the system to collect real-world ADL or fall data and fine-tune the fall detection model via personalization.

The ultimate goal is for the Smartfall system to detect falls accurately, i.e. not missing any falls and not generating too many false positive prompts.

## IV. Experimental Setup

In this section, we explain how we run the SmartFall system focusing on edge-computing specific issues. Here we focused on evaluating the performance of three distinct implementations that are based on the software architecture described in Section III.

### A. Key factors for comparison

Our primary goal is to compare their performance with respect to four pivotal issues that are typically encountered in edge-computing systems. These issues are:

- **IoT Device Battery Life:** Since many edge-computing applications involve the use of IoT (Internet of Things) devices that are powered by internal batteries, the battery life of these devices becomes a significant concern. The efficiency and longevity of the battery can determine the device's overall utility and reliability in real-world scenarios. If a device runs out of battery, it cannot monitor the user, creating a period of vulnerability.
- **Data Latency:** This refers to the delay between the generation or sensing of a data point and the prediction generated for that data point by the machine learning model. For a fall detection system, rapid detection and response are essential. If there's a significant delay between the occurrence of a fall and its detection, it could delay the necessary intervention in critical conditions. Real-time processing ensures that assistance can be promptly dispatched or alerted in case of a detected fall.
- **Potential Data Loss:** This refers to the loss of data points or sensor readings during the transmission to the AI model that processes the data for decision-making. In fall detection, feeding the machine learning with the full spectrum of data leading up to an impending fall is crucial for determining if a fall has occurred. For this fall detection architecture, consistent and continuous data capture is paramount. If sensor data gets lost on transmission, the system will fail to detect an actual fall, compromising the safety of the user. Given the potentially life-saving nature of fall detection, ensuring data integrity is non-negotiable.

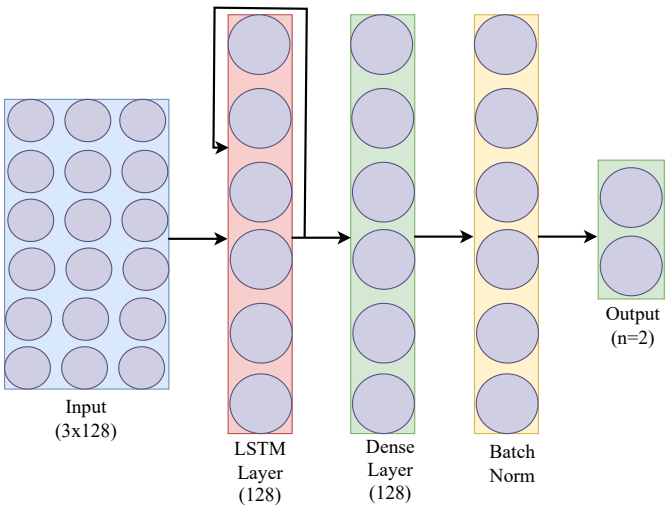


Fig. 5. LSTM Architecture

- **Model Accuracy:** The model accuracy measures the precision of machine learning models in detecting genuine fall events. Given varied computational capacities among watches, phones, and servers, accuracy discrepancies may arise. Machine learning models, when deployed on edge devices like watches or phones, undergo compression for efficiency, often using TensorFlowLite (TFLite) when deployed on devices running WearOS. This compression can result in a slight accuracy reduction compared to the native TensorFlow models available on servers. Also, the computational prowess and memory allocation on an IoT device can influence model processing and, consequently, its accuracy. Servers, with their superior resources (memory, GPU, multiple cores), can potentially harness the full capability of TensorFlow models, unlike watches and phones which operate under constraints. While edge devices benefit from the speed and efficiency of TFLite models, maintaining a high standard of accuracy is also important. If significant accuracy trade-offs are observed, a computational offload to more resourceful devices, such as servers, should be considered.

### *B. Machine learning model*

We utilized a basic two-layer LSTM deep learning model for the training and creation of the fall detection model. This model is lightweight and can be deployed on computation-constrained edge devices easily. Our main motivation is not about enhancing the accuracy of the fall detection model but comparing the architectures.

The input layer in our LSTM has 3 nodes for accelerometer x, y, and z vectors, with a variable input shape of ($W$, 3, 64), where $W$ is the window size which is set to 128 and represents around four seconds of data. The 64 is the batch size. The training involves Binary Cross Entropy (BCE) loss function with the ADAM optimizer. The functions utilized by the two dense layers are Relu and Sigmoid. For each prediction, the model provides a probability of a fall between 0 and 1. We used the threshold 0.5. Fig. 5 shows the details of its architecture.

The trained model has an F1 Score of 0.7873 using falls (300 samples) and ADL data (435 samples) of 12 participants (7 males and 5 females) for training. We trained the model on the server and generated the TensorFlowLite version for the phone and watch.

### *C. Experiments*

We recruited six participants (all students aged between 27 and 33 years) to use the three different architected SmartFall App for a specified amount of time and performed a prescribed list of falls and ADL activities. In particular, the participants would wear the watch on their left wrist and carry the phone on the right waist or have it nearby with the SmartFall App running for at least an hour for each experiment. Each participant will perform 5 different types of falls (i.e. back fall, front fall, left fall, right fall, rotate fall) and 8 different types of ADL tasks (i.e. walking, waving, washing hands, putting on/taking off a jacket, drinking water, picking up an object from the floor, sweeping the floor, sitting and standing up from a chair).

In the Battery Life experiment, participants were instructed to wear the watch for an hour, beginning with a fully charged device. After the hour, we noted the remaining battery percentage. From this, we determined the battery consumption. During this hour, participants weren't required to engage in specific activities; they went about their daily routines.

In the Data Latency experiment, the transmission time represents the duration between the data sensed and the completion of the prediction. We instrumented all three different architected SmartFall App to collect the time between each generated accelerometer data and the prediction. The participants were instructed to wear the watch for an hour for each architecture of the SmartFall system. Finally, we calculated the average time of these collected data transmission times to report the data latency. In this experiment, we did not record the transmission time for the first 127 data points as the system has to accumulate the required amount of historical data before it can make the first prediction.

For the Data Loss experiment, we asked the participant again to wear the watch for an hour for each architecture type and perform regular household chores at home. After an hour, we downloaded the saved data collected from the participant while wearing the watch from the Couchbase server where we archived the data after each prediction. Data is sensed once every 32 ms. We should have 112500 data points for an hour. Usually, if a prediction triggers an alert, all three apps stop data collection until the user responds to the UI. This could be a challenge and might affect the amount of data archived in three different apps. Even setting some specific activities to address this issue can be a problem as our model can trigger false positives. To overcome this problem, we instrumented all three apps not to stop data collection at the time of gathering user feedback. After downloading the data, we check the data

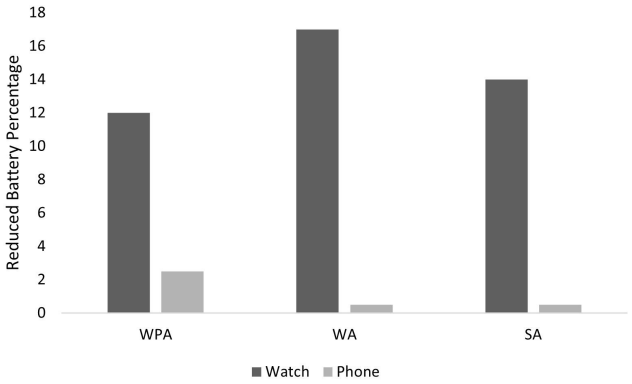


Fig. 6. Battery life consumption in phone and watch for each architecture.

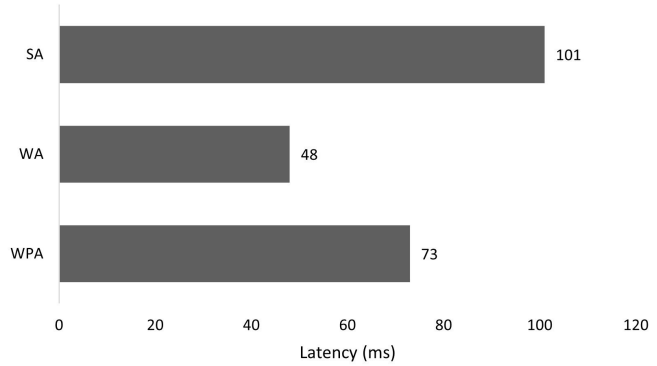


Fig. 7. Data latency for each architecture.

point count in all three architectures to determine potential data loss.

Lastly for the Model Accuracy, we asked the users to wear the watch and perform 5 different types of falls (i.e. back fall, front fall, left fall, right fall, rotate fall) and 8 different types of ADL tasks (i.e. walking, waving, washing hands, wearing a jacket, drinking water, picking up an object from the floor, sweeping the floor, sitting and standing). All the activities were performed in our lab and the fall activities were performed on a queen-sized air mattress. This testing activity with subjects is approved by Texas State IRB number 7846.

The goal is to check whether the type of software architecture has an influence on the accuracy of fall detection. This means that the ADL activities should not trigger a fall and most falls should not be missed. We calculated the precision, recall, and accuracy of the model from these sessions.

To ensure consistency in our experiment, we used the same kind of devices for all the participants. For the watch application, we selected the TicWatch Pro 3. This watch is renowned for its performance and reliability, making it a suitable candidate for our tests. On the other hand, for the phone application, we chose the Google Pixel 6, which comes with 8GB of RAM and 128GB of internal storage. As for the server-side architecture, the implementation ran on a Dell Precision 7820 server with 128 GB of memory and one Nvidia GeForceGTX 1080 GPU.

To maintain a uniform data management platform across all devices - the watch, phone, and server - we utilized Couchbase as the storage medium for the collected data. Couchbase is known for its scalability and performance, making it a good fit for our experiment.

## V. Results and analysis

### A. IOT Device Battery Life

For this study, we had 6 participants wearing the watch for an hour per architecture type and monitored the battery consumption on both the watch and phone.

Figure 6 shows the reduced battery percentage for watches and phones across three distinct architectures. In the Watch-and-Phone-based Architecture (WPA), the phone's battery drains more than in other systems because it is responsible for data processing and model prediction. Additionally, with the watch transmitting data to the phone through the energy-efficient Bluetooth, there is only 12% battery consumption in the watch. On the other hand, the Watch-based Architecture (WA) sees minimal phone battery consumption, as the phone's main role is to transfer the profile to the app on the watch; all other activities, including data processing, model prediction, and feedback, are managed in the watch. Consequently, the watch in the WA consumes a more substantial portion of the battery, specifically 17% in an hour. In the Server-based Architecture (SA), the watch buffers the sensed data at every 32 ms and interacts with the server to send 128 consecutive data points for the prediction, retrieve predictions, and subsequently ask for user feedback. This results in a 14% battery use on the watch and less than 1% on the phone, which, similar to the Watch-based Architecture (WA), is primarily employed to collect user's profile data from the user and initiate the watch application.

### B. Data Latency

In this experiment, we measured the duration starting from when a data point is collected for analysis until the prediction for this data point is received. This timing encompasses the overall collection, processing, and the model's inference duration.

Figure 7 shows the data latency in three different architectures. Here we can see the server-based architecture has the most data latency and the watch-based has the least data latency. For the phone-based application, the watch sends the data to the phone for the model to predict via Bluetooth Low Energy(BLE). The inference happens on the phone and the phone finally sends the prediction result to the watch using BLE again. In this case, it is 73 ms. But the phone is multi-threaded. So it is running a few threads at a time. When the inference for one window is happening the watch is continuing to collect the data for the next window and sending it to the phone. In the watch-based application, the latency time is less than WPA. Here the timer starts when the window is ready for prediction and in this case, the inference happens

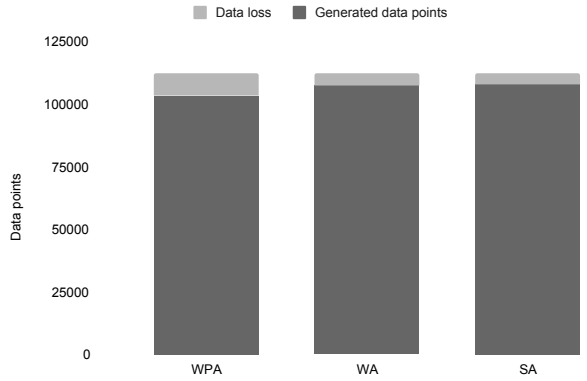


Fig. 8. Potential data latency for each architecture.

in the watch itself so there is no delay for Bluetooth data transmission twice. After the inference, the watch displays the prediction result for user's feedback.

Compared to WPA and WA, the server-based application has the largest data latency time. Transmitting data from the watch and receiving the prediction takes about 101 ms on average. The increased latency is attributed to the data being sent from the watch to the server, making it more reliant on network (Wifi) communication speeds. With the model's analysis occurring on the server, there's an additional delay when relaying the prediction result back to the watch. This return transmission faces similar latency challenges.

### C. Potential Data Loss

For this study, we asked our participants to wear the watch running the application on three different architecture types for an hour and do some ADL activities such as everyday chores. Since these Apps stop recording data when there is a fall predicted to wait for the user input, we instrumented the Apps to continuously record data without stopping for user feedback. After an hour we stopped the data collection and fetched the data from our Couchbase server.

The generated data points and potential data loss in each architecture are shown in Figure 8. According to the figure, we can say that the Server-based Architecture (SA) records 108,057 data points, which generated more data in an hour than the other two approaches. The Watch-based Architecture (WA) closely trails with 107,386 data points, while the WPA lags a bit behind, recording 103,662 data points.

However, when it comes to data loss, which is a critical metric for the integrity of these systems, the WPA registers the highest loss with 8,838 points. This considerable data loss can be influenced by its dual-device setup. Despite Bluetooth Low Energy's (BLE) rapid transmission capabilities, its vulnerability during data transfer, especially when the phone's processing capacity is overwhelmed, loses some data points received via BLE.

In contrast, the WA, which centralizes both processing and prediction tasks on the watch, experiences a reduced data loss of 5,114 points. The limited computational capacity of the watch does occasionally obstruct continuous data collection, manifesting in these losses. when the watch reaches the maximum number of processes, it stops collecting data from the sensor until a process is released. The SA reported the least data loss at 4,443 data points. While the WA and SA should theoretically generate similar data points as in both architectures data is collected and processed in the watch. The success of SA can be credited to its design that offloads prediction tasks to an external server. Offloading predictive functions to a server evidently minimizes instances where the watch hits its processing peak, reducing the amount of data loss.

TABLE I
REAL TIME MODEL EVALUATION RESULTS FOR EACH ARCHITECTURE

| | | WPA | WA | SA |
|---|---|---|---|---|
| Participant 1 | Accuracy | 0.78 | 0.82 | 0.83 |
| | Precision | 0.84 | 0.84 | 0.88 |
| | Recall | 0.68 | 0.72 | 0.73 |
| | F1 Score | 0.75 | 0.78 | 0.79 |
| Participant 2 | Accuracy | 0.75 | 0.77 | 0.80 |
| | Precision | 0.84 | 0.84 | 0.84 |
| | Recall | 0.64 | 0.66 | 0.70 |
| | F1 Score | 0.73 | 0.74 | 0.76 |
| Participant 3 | Accuracy | 0.71 | 0.74 | 0.80 |
| | Precision | 0.80 | 0.84 | 0.88 |
| | Recall | 0.59 | 0.62 | 0.69 |
| | F1 Score | 0.68 | 0.71 | 0.77 |
| Participant 4 | Accuracy | 0.75 | 0.78 | 0.80 |
| | Precision | 0.84 | 0.88 | 0.88 |
| | Recall | 0.63 | 0.66 | 0.69 |
| | F1 Score | 0.72 | 0.75 | 0.77 |
| Participant 5 | Accuracy | 0.77 | 0.78 | 0.81 |
| | Precision | 0.84 | 0.84 | 0.84 |
| | Recall | 0.65 | 0.67 | 0.72 |
| | F1 Score | 0.73 | 0.74 | 0.77 |
| Participant 6 | Accuracy | 0.80 | 0.82 | 0.85 |
| | Precision | 0.84 | 0.84 | 0.88 |
| | Recall | 0.70 | 0.78 | 0.75 |
| | F1 Score | 0.76 | 0.78 | 0.80 |
| Average | Accuracy | 0.76 | 0.79 | **0.82** |
| | Precision | 0.83 | 0.83 | **0.87** |
| | Recall | 0.65 | 0.69 | **0.71** |
| | F1 Score | 0.73 | 0.75 | **0.78** |

### D. Model Accuracy

For this experiment, we asked the participants to wear the watch running the application on different architectures and asked them to do a prescribed set of activities. We asked them to do 5 different falls on an air mattress and 8 different ADL activities to test the model accuracy. Each activity is repeated five times. Table I shows the results of evaluating the real-time model in each architecture. We employed the main model (native TensorFlow PB version) in Server-based Architecture (SA) and TFLite version in the Watch and Phone-based Architecture (WPA) and Watch-based Architecture (WA).

According to the results on the model's performance across various architectures, we can see that only the server-based application's average F1 Score of 0.78 closely mirrored the results obtained during training, which is 0.79. However, a noticeable drop in F1 score was recorded for the phone-based and watch-based applications. This discrepancy could

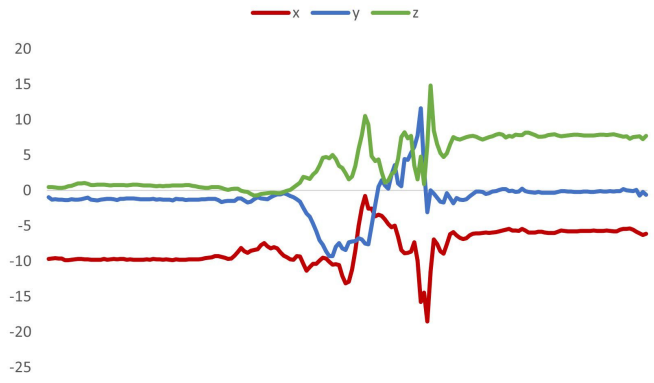


Fig. 9. Accelerometer data of Left Fall

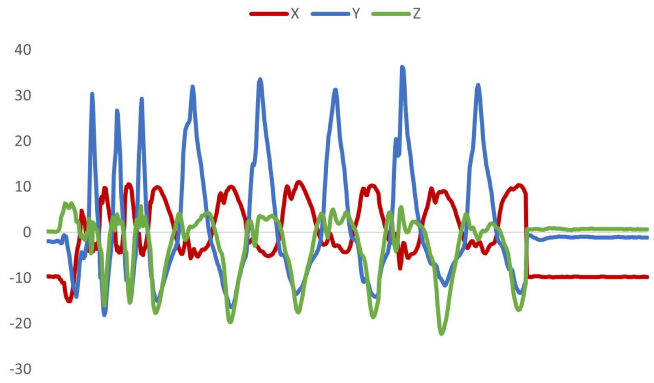


Fig. 10. Accelerometer data of an ADL activity(waving)

potentially be attributed to the transformation of the trained model into a lighter TFlite model. The TFLite is a pruned version of the main model optimized for running on the edge device, which is expected to drop some accuracy. Another reason could be due to the capacity or computation power of the hardware. The watch and phone we are using have limited memory which could result in data loss which impacts the model accuracy.

In our experiments, most of the fall and ADLs were correctly detected by the server-based architecture, we have not found any false positives or false negatives in server-based, but there were false positives or true negatives in the other two architectures. Figure 9 shows a visualization of a left fall. This fall was correctly detected by the server-based architecture but the other two architectures missed it. Figure 10 shows another visualization of an ADL task, waving hand. Here user waved both hands 7 times. The watch and phone-based (WPA) and watch-based (WA) architectures detected it as a fall but the server-based architecture correctly detected it as an ADL.

On the other hand, precision value remained almost consistent across all architectures, indicating the model's ability to correctly identify falls (True Positives), though challenges lie in terms of distinguishing ADL activities (resulting in False Positives).

## VI. Discussion

Our experimental evaluation across the Watch-and-Phone-based (WPA), Watch-based (WA), and Server-based (SA) architectures revealed distinct advantages and constraints for each. WPA displayed considerable battery drain due to dual-device interaction, while WA's localized processing led to high watch battery consumption. However, SA displayed optimal model accuracy, closely mirroring training results. In contrast, the edge-based WPA and WA witnessed reduced F1 scores, potentially due to their transition to TensorFlow Lite and inherent hardware limitations. Data latency was most pronounced in SA, attributable to its reliance on wifi communication with the server.

Moreover, data loss was notably higher in WPA. Conclusively, while WPA and WA provide proximity and privacy benefits, they compromise on model accuracy. SA, despite its superior performance on accuracy, faces challenges with latency, underscoring the need for careful architectural selection in real-world implementations.

### A. Messaging Server

According to the results and their analysis, the Server-based Architecture (SA) offers several merits, particularly in the realms of IOT-device battery conservation, data integrity, and prediction accuracy of the model. By offloading the prediction task to the server, there is a notable conservation of the smartwatch's battery life. Further, the robust processing capabilities of the server ensure that model accuracy is optimized, thereby enhancing the reliability of predictions. However, a tangible challenge presented by the SA approach is data latency. Each data transmission to the server necessitates a handshaking process. This repetitive procedure, while ensuring secure communication, introduces a latency that can be consequential, especially in applications where real-time predictions are vital.

To address this limitation, an evolution of the architecture can be envisioned. Transitioning to a messaging server infrastructure such as NATS.IO [23], which is a lightweight messaging system designed for modern distributed systems. Messaging servers, inherently designed for rapid, continuous asynchronous data exchanges, can significantly trim the data transmission times. By bypassing the traditional handshaking procedure in every communication cycle, data latency can be substantially reduced. Furthermore, by processing raw data directly on the server-side, instead of the watch, there is potential for a dual benefit: achieving improved battery longevity for the edge device and leveraging the server's computational prowess for more intricate data processing.

### B. Scalability, Deployment, Maintenance, and Network Fluctuations

We designed all three architectures to be easily deployed and maintained. Also, all three architectures are scalable. The Watch-and-Phone-based Architecture (WPA) integrates functionalities between the smartwatch and the smartphone. In scalability terms, WPA offers a middle ground. As the number of users increases, smartphones, responsible for significant

computations, might face resource challenges. Deployment for WPA is generally seamless, taking advantage of typical smartwatch-phone pairing procedures. Yet, synchronizing both devices effectively is crucial. Maintenance in WPA requires consistent monitoring since both devices play integral roles. Given the reliance on real-time communication between the watch and phone, network inconsistencies can introduce operational challenges.

In contrast, the Watch-based Architecture (WA) operates with considerable autonomy. This independent nature provides WA with an edge in scalability. Each smartwatch self-sufficiently processes its data, negating centralized processing issues. Deployment for WA is user-friendly, with users needing only an initial setup on the smartwatch. WA will be easy for older adults to use because they only need to manage a single device. Meanwhile, maintenance is primarily affected by smartwatch software updates. The architecture's inherent design makes it less vulnerable to network issues, though data transfers to the cloud for archival for continuous model refinement and personalization might face occasional interruptions.

The Server-based Architecture (SA) emphasizes the capabilities of cloud servers. Regarding scalability, SA offers potent data processing advantages due to server capacities. However, as more users join, the server might require additional resources. Deployment in SA presents initial complexities, especially in setting up a robust server, but onboarding users subsequently is straightforward. Maintenance for SA is mainly server-centric. While the smartwatch demands minimal attention, the server's periodic updates, security, and scalability become pivotal. Of the three, SA has the most pronounced network reliance. The continuous data transmission from the watch to the server can be impacted if in unstable internet network environments.

In summary, each architecture presents distinct advantages and limitations. The optimal selection depends on the intended deployment environment, projected user numbers, users' privacy concerns, and network stability.

## VII. Conclusions

We implemented the SmartFall system, a real-time IoT application using three different software architectures. We compared and analyzed the trade-off in battery consumption, model accuracy, potential data loss, and prediction latency of these three architectures. Server-based architecture demonstrated the best model prediction accuracy, but the worst in terms of data privacy and data transmission time. This suggests that deploying a robust real-time IoT application on the edge requires the support of a native cloud on the edge rather than a general cloud/server provided by vendors. The native cloud on the edge will bring all the computing power of the cloud while maintaining data privacy and maintain the quality of services. A native cloud on the edge will localize all processing of data on the edge and leverage native cloud services like automatic scaling of resources and resiliency. The open source projects such as KubeEdge [24], TinyEdge [25], and EdgeX [26] are all promising platforms to explore in the future if they can be made available on the smartwatch, smartphone, or on a set-top box that consumers can use in their home.


## Acknowledgement

We thank the National Science Foundation for funding the research under the NSF-SCH grant (2123749).


## References


[1] "The growth in connected iot devices is expected to generate 79.4 zb of data in 2025, according to a new idc forecast." https://www.iotcentral.io/blog/iot-is-not-a-buzzword-but-necessity, 2023.

[2] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[3] (2019) Introduction to tensorflow lite. [Online]. Available: https://www.tensorflow.org/mobile/tflite/

[4] T. R. Mauldin, M. E. Canby, V. Metsis, A. H. Ngu, and C. C. Rivera, "Smartfall: A smartwatch-based fall detection system using deep learning," *Sensors*, vol. 18, no. 10, 2018.

[5] T. Mauldin, A. H. Ngu, V. Metsis, and M. E. Canby, "Ensemble deep learning on wearables using small datasets," *ACM Transactions on Computing for Healthcare*, vol. 2, no. 1, pp. 1–31, December 2020.

[6] D. Raggett, "The Web of Things: Challenges and Opportunities," *IEEE Computer*, vol. 48, no. 5, pp. 26–32, May 2015.

[7] R. Want, B. N. Schilit, and S. Jenson, "Enabling the Internet of Things," *IEEE Computer*, vol. 48, no. 1, pp. 28–35, 2015.

[8] L. Baresi, L. Mottola, and S. Dustdar, "Building Software for the Internet of Things," *IEEE Internet Computing*, vol. 19, no. 2, pp. 6–8, 2015.

[9] A. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and Q. Z. Sheng, "IoT Middleware: A Survey on Issues and Enabling Technologies," *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 1–20, Feb 2017.

[10] M. Gutierrez, M. Fast, A. H. Ngu, and B. Gao, "Real-time prediction of blood alcohol content using smartwatch sensor data," in *IEEE International Conference on Smart Health*. Phoneix, Arizona: Springer, November 2015.

[11] A. H. H. Ngu, J. S. Eyitayo, G. Yang, C. Campbell, Q. Z. Sheng, and J. Ni, "An iot edge computing framework using cordova accessor host," *IEEE Internet of Things Journal*, vol. 9, no. 1, pp. 671–683, 2022.

[12] *AWS IoT*, 2020, https://aws.amazon.com/iot.

[13] *IBM Watson IoT*, 2020, https://www.ibm.com/internet-of-things.

[14] *ThingSpeak for IoT*, 2020, https://thingspeak.com/.

[15] Google cloud: Connected device solutions. https://cloud.google.com/iot-core.

[16] A. Das, S. Patterson, and M. Wittie, "Edgebench: Benchmarking edge computing platforms," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 175–180.

[17] T. Goethals, F. DeTurck, and B. Volckaert, "Extending kubernetes clusters to low-resource edge devices using virtual kubelets," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020.

[18] R. A. Light, "Mosquitto: server and client implementation of the mqtt protocol," *Journal of Open Source Software*, vol. 2, no. 13, p. 265, 2017.

[19] Google cloud: Edge tpu. https://cloud.google.com/edge-tpu.

[20] A. H. Ngu, V. Metsis, S. Coyne, P. Srinivas, T. Salad, U. Mahmud, and K. H. Chee, "Personalized watch-based fall detection using a collaborative edge-cloud framework," *International journal of neural systems*, vol. 32, no. 12, p. 2250048, 2022.

[21] A. H. Ngu, A. Yasmin, T. Mahmud, A. Mahmood, and Q. Z. Sheng, "P-fall: Personalization pipeline for fall detection," in *Proceedings of the 8th ACM/IEEE International Conference on Connected Health: Applications, Systems and Engineering Technologies*, 2023, pp. 173–174.

[22] Detecting falls using a mobile device. https://patents.google.com/patent/US20190103007A1.

[23] (2024) Nats.io. https://nats.io. [Online]. Available: https://nats.io

[24] A kubenertes native edge computing framework. https://kubeedge.io/. Accessed: 2023-10-25.

[25] W. Zhang, Y. Zhang, H. Fan, Y. Gao, and W. Dong, "A low-code development framework for cloud-native edge systems," *ACM Trans. Internet Technol.*, vol. 23, no. 1, feb 2023. [Online]. Available: https://doi.org/10.1145/3563215

[26] Edgex foundry. https://www.edgexfoundry.org/. Accessed: 2023-10-25.