Comprehensive IoT-Based Water Management System: Cloud Integration for Residential Conservation

1st Mohamad Syafiq Asyraf Bharudin
Department of Computer Science, Kuliyyah of ICT, International Islamic University Malaysia
Kuala Lumpur, Malaysia
msasyraf01@gmail.com

2nd Ahmad Anwar Zainuddin
Department of Computer Science, Kuliyyah of ICT International Islamic University Malaysia
Kuala Lumpur, Malaysia
anwarzain@iium.edu.my

3rd Aliah Maisarah Roslee
Department of Computer Science, Kuliyyah of ICT International Islamic University Malaysia
Kuala Lumpur, Malaysia
aliahmaisarahr@gmail.com

4th Nik Nor Muhammad Saifudin
Department of Computer Science, Kuliyyah of ICT, International Islamic University Malaysia
Kuala Lumpur, Malaysia
saifudinkamal11@gmail.com

***Abstract*— In an era where water conservation is increasingly critical, this paper unveils a groundbreaking IoT-based system that revolutionizes water management through smart automation. By harnessing cutting-edge technologies such as the ESP32 microcontroller, this system seamlessly integrates with cloud services to deliver unparalleled efficiency in water monitoring and control. Equipped with an ultrasonic sensor for precise water level measurements and a rain sensor for immediate leak detection, the system stands at the forefront of innovation in water resource management. Designed to automatically adjust water levels and respond swiftly to leaks, this system minimizes waste and ensures optimal water use with remarkable precision. The advanced cloud infrastructure supports real-time data logging and analysis, facilitated by MQTT for agile data transmission and HTTP/REST APIs for smooth system integration. MongoDB's robust database management enhances the system's ability to handle and interpret vast amounts of data effectively. The user experience is further elevated through a dynamic interface built with React Native, offering a seamless and intuitive interaction across multiple devices. This pioneering solution not only advances smart water management technology but also highlights its transformative potential for conservation efforts. By combining IoT innovation with intuitive software, the system sets a new standard for efficient and sustainable water management in residential settings.**

Keywords—Water level monitoring system, MQTT, NodeMCU ESP32, Internet of Things, Cloud Integration, Residential Conservation

I. Introduction

Water management is a crucial aspect of urban and suburban regional planning, especially in developing countries like Malaysia, where rapid urbanization and climate sensitivity present significant challenges. This is particularly important in residential areas, especially in densely populated neighborhoods with aging infrastructure. Proper water management is vital not only for preventing structural damage in these areas but also for supplying clean and safe drinking water to residents. This aligns with Sustainable Development Goal 6 (Clean Water and Sanitation), which promotes efficient water use, minimizes waste, and emphasizes maintaining water quality and ensuring access to clean, safe water [1]. Therefore, solving problems associated with water management is necessary not only to protect people's health but also to ensure the stability of housing stock in Malaysia.

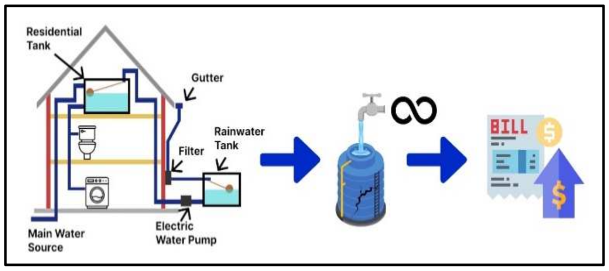


Fig.1. Drawbacks of Current Water Management

Figure 1 illustrates the impacts of poor water management on housing areas. The primary issue in today's infrastructure is on the aging of assets, which lose their strength and flexibility, making them more susceptible to cracks, deformation and eventually can causes leak to occur [2]. In some residential areas, rainwater harvesting systems are used to collect and store rainwater, aiming to reduce surface runoff and ease the burden on urban drainage [3]. But, these systems have some potential flaws in which inadequate tanks size and gutters may lead to overflows due to insufficient capacity. These faulty results in higher water bills and increased the maintenance costs. Indirectly, these problems may also lower the property values which causes structural damage and deterring potential buyers. With that, IoT-Based Water Management System (IWMS) comes into play, offering a solution that leverage real-time monitoring with automated water control and micro leak detection mechanism for an efficient and reliable water management system.

Section I introduces the system and its role in sustainable water management through IoT. Section II reviews the hardware components, including the ESP32, ultrasonic sensor, rain sensor module, cloud services, mobile app framework, NoSQL database, and hybrid data transmission methods (MQTT and HTTP/REST API). Building on this foundation, Section III details the methodology, with a focus on cloud connectivity and data transmission. Section IV then presents the results and discussion, highlighting database data, HTTP/REST API examples, and the mobile app interface. Finally, Section V wraps up with a comprehensive summary of the findings and overall discussions.

II. COMPONENT AND DESIGN STRUCTURE

A. Mobile Application Development Framework

Portable is one of the key aspects that define IoT. Without portability, IoT devices would be restricted to fixed locations, limiting their application to scenarios where devices are stationary as well as reducing the flexibility of the device to adapt to changing environments or use cases. Computers, laptops, smartwatches, and smartphones have become essential in today's world and can even be considered as core component in IoT that exemplify the importance of portability.

React Native was opted for this work due to its ability to facilitate cross-platform development, which is crucial in the diverse and interconnected landscape of IoT. Additionally, React Native's hot reloading feature enable developers to see changes in real-time, which is particularly beneficial in the iterative and fast-paced environment of IoT development [4].

B. Utilizing NoSQL Database for IoT

Historical data provides valuable insights into the current condition of a system. By collecting and analyzing this data, trends in system behavior over time can be identified, offering a baseline for detecting anomalies in real-time. Comparing current data against historical norms allows for the quick identification of unusual behavior, potentially signalling issues that require immediate attention.

To effectively manage and analyze such data, a key tool is the database. A database is defined as an organized collection of data that can be easily accessed, managed, and updated [5]. This structured collection allows for efficient storage and retrieval of data, which is crucial for various applications across different fields, including business, healthcare, and scientific research [5]. Databases are typically managed by Database Management Systems (DBMS), which serve as intermediaries between users and the databases themselves, facilitating operations such as data definition, creation, querying, updating, and administration [5]. Database can be divided into two data models which are Relational Data Model and Not only Structured Query Language (NoSQL) Data Model.

TABLE 1. Differences Between Relational Data Model & NoSQL Data Model. [6], [7], [8]

	Relational Data Model	**NoSQL Data Model**
Type	Relational	Non-relational
Schema	Fixed	Dynamic
Scalability	Vertical	Horizontal
Language	Structured Query Language	Unstructured Query Language
Data	Stored in tabular form	Stored in unstructured format using various methods (eg. JSON, key-value pairing, family grouping, graph nodes/edges)
Flexibility	Rigid to defined schema	Flexible
Data Modeling Technique	Normalization (eg. 1NF, 2NF, 3NF)	Denormalization

Relational Data Model can be seen as the root of designing a database architecture, uses the sense of "relationship" between data derived from mathematic and scientific point of view [9]. In a perspective of data structure, relation can be represented as tables of values which consist of attributes to provide characteristic to an element and the content of the attributes are known as rows of tuples [10]. The structure of a relational database can be defined using a template called schema and can be created through DataBase Management System (DBMS). Create, Read, Update and Delete (CRUD) are four basic operations of a database. Generally, most DBMS uses a standard querying language called Structured Query Language (SQL) to perform the CRUD operations. Normally, most SQL database requires normalization process in organizing data to reduce redundancy and improve data integrity. This process involves dividing large tables into smaller tables that are called normal foms (1NF, 2NF, 3NF) and defining relationships between them. In contrast to NoSQL Data Model, it is a non-relational database structure which does not use SQL syntax. MongoDB, one of the most widely used NoSQL databases, stores data in a document-oriented structure using JSON or BSON formats, which can include nested structures [11]. This allows related data to be stored together, reducing the need for complex joins. Unlike relational databases, NoSQL databases often employ denormalization, where data is intentionally duplicated to optimize read performance and simplify queries. NoSQL databases do not require a fixed schema, allowing data to be stored with varying structures within the same database, providing flexibility in how the data is organized [11]. This will offer low latencies data flow and eventually increase the performance of exchanging data between devices which tallied with the objective of this work.

C. AWS & MongoDB Integration for IoT Applications

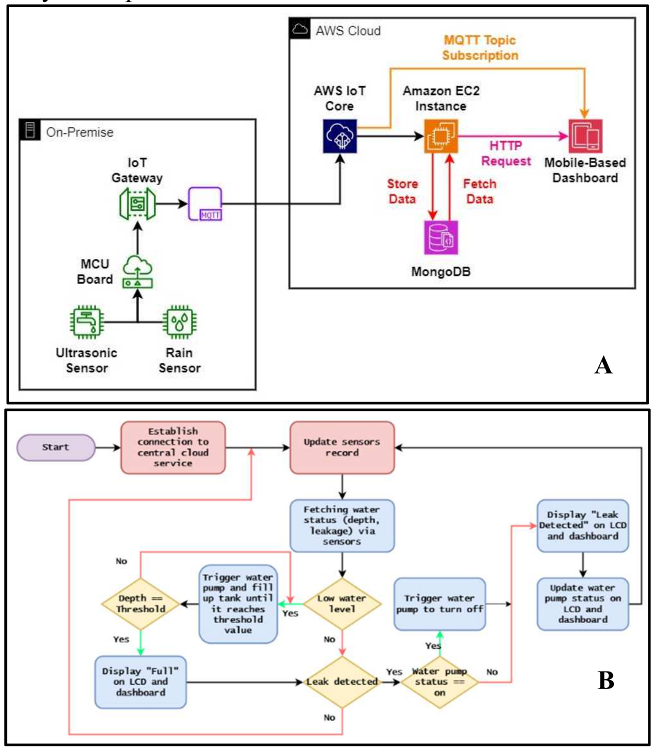
Integrating AWS with MongoDB creates a robust architecture for IoT applications, harnessing the unique strengths of both platforms to improve data management, processing, and scalability. This integration allows for seamless data flow from IoT devices to the cloud, enabling real-time insights and operational efficiency.

AWS is one of the powerful cloud platforms where its infrastructure is specially tailored for scalability and reliability, supporting application deployment without relying on extensive on-premise hardware. The set of services offered are particularly well-suited for IoT solutions, which often generate large volumes of data. Thus, when pairing with MongoDB, this integration can reduce the need for complex data transformation, which helps in streamlining the data ingestion pipeline.

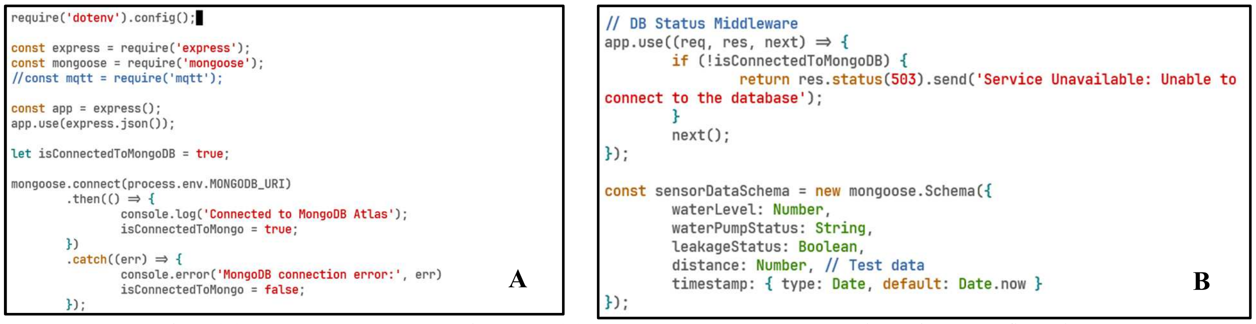
AWS IoT Core, part of the services used in this work, plays a critical role in the architecture. It acts like a bridge, routing messages from these devices in an effective manner while guaranteeing reliable and secure data transmission of that information. This capability becomes particularly important in real-time applications where timely delivery of the data is crucial for making decisions or responsiveness to an operational process. Once the information has been passed through AWS IoT Core, it will be further forwarded to applications running on Amazon EC2 instances, which interact directly with the MongoDB database. In turn, it will enable seamless flow in data. The application will have almost real-time access to the latest data. For instance, this kind of effectiveness is required in applications such as environmental monitoring, condition monitoring of assets, or responding to sensor alerts.

III. METHODOLOGY

This work seamlessly integrates IoT technology, software engineering, networking concepts, and electronic engineering to develop an innovative water level monitoring system. By harnessing the power of IoT, it transforms traditional water level monitoring practices, significantly enhancing both user efficiency and system performance. A critical step in the prototype development is outlining the system topology, which allows for a thorough conceptualization of the system and an in-depth exploration of requirements prior to real-world deployment. As illustrated in Figure 2A, the diagram shows the physical connections of sensors to the MCU board and the integration of cloud services through AWS. Figure 2B depicts the system operation flows of IWMS.


AWS Cloud
MQTT Topic Subscription
AWS IoT Core
Amazon EC2 Instance
HTTP Request
Mobile-Based Dashboard
Store Data
Fetch Data
MongoDB
On-Premise
IoT Gateway
MQTT
MCU Board
Ultrasonic Sensor
Rain Sensor
A
Start
Establish connection to central cloud service
Update sensors record
Fetching water status (depth, leakage) via sensors
Display "Leak Detected" on LCD and dashboard
No
Depth == Threshold
Trigger water pump and fill up tank until it reaches threshold value
Yes
Low water level
Trigger water pump to turn off
Update water pump status on LCD and dashboard
Yes
No
Yes
Display "Full" on LCD and dashboard
Leak detected
Yes
Water pump status == on
No
No
B
Fig.2. (A) MCU-AWS Integration & Data Flow Diagram (B) IWMS System Operation Flow Chart.

The system's operation begins by establishing a robust connection between the centralized IoT module and AWS cloud services, a critical step for ensuring seamless and reliable data transmission from the network of sensors to the cloud. This connection is facilitated through the MQTT protocol, which is well-suited for IoT applications due to its efficiency in handling small, lightweight data packets. In this architecture, AWS IoT Core acts as the MQTT broker, managing communication between the sensors and the cloud. Specifically, it handles incoming MQTT messages from the NodeMCU ESP32 board—a key microcontroller unit that gathers data from various sensors. AWS IoT Core ensures that these messages are properly routed and distributed to the appropriate subscribers, such as other cloud services, databases, or applications that need to process or store the data. As shown in Figure 3A, the connection setup between AWS and MongoDB uses a secure approach by storing the URI in a separate environment configuration file known as dotenv (.env). The MongoDB URI is accessed by referencing the environment variable MONGODB_URI, ensuring that sensitive information remains hidden and secure.



```
require('dotenv').config();

const express = require('express');
const mongoose = require('mongoose');
//const mqtt = require('mqtt');

const app = express();
app.use(express.json());

let isConnectedToMongoDB = true;

mongoose.connect(process.env.MONGODB_URI)
        .then(() ⇒ {
                console.log('Connected to MongoDB Atlas');
                isConnectedToMongo = true;
        })
        .catch((err) ⇒ {
                console.error('MongoDB connection error:', err)
                isConnectedToMongo = false;
        });
```

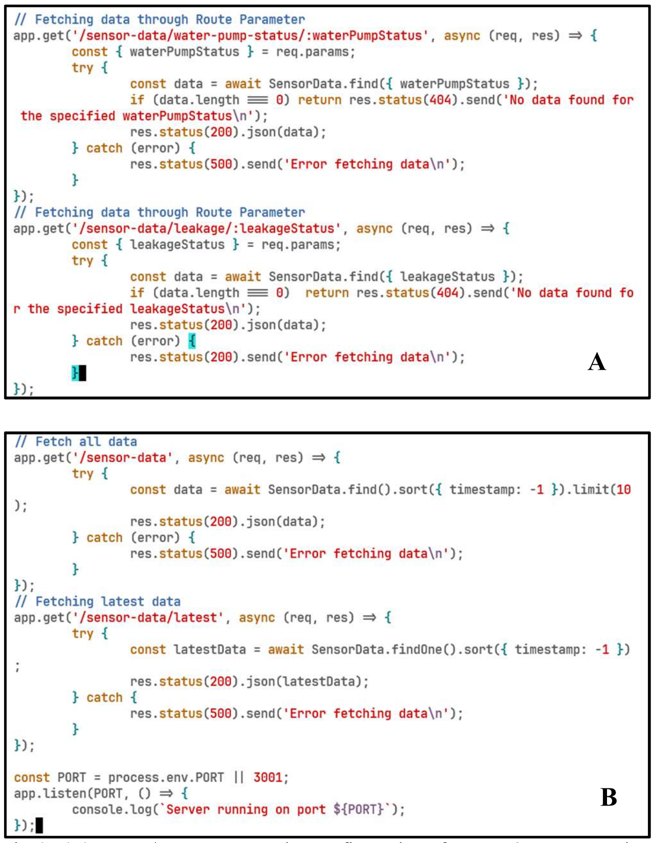
A

```
// DB Status Middleware
app.use((req, res, next) ⇒ {
        if (!isConnectedToMongoDB) {
                return res.status(503).send('Service Unavailable: Unable to
connect to the database');
        }
        next();
});

const sensorDataSchema = new mongoose.Schema({
        waterLevel: Number,
        waterPumpStatus: String,
        leakageStatus: Boolean,
        distance: Number, // Test data
        timestamp: { type: Date, default: Date.now }
});
```

B

Fig.3. (A) MongoDB-AWS Connection Setup (B) MongoDB-AWS Cross Connection Schema Configuration.



```
// Fetching data through Route Parameter
app.get('/sensor-data/water-pump-status/:waterPumpStatus', async (req, res) ⇒ {
        const { waterPumpStatus } = req.params;
        try {
                const data = await SensorData.find({ waterPumpStatus });
                if (data.length === 0) return res.status(404).send('No data found for
 the specified waterPumpStatus\n');
                res.status(200).json(data);
        } catch (error) {
                res.status(500).send('Error fetching data\n');
        }
});
// Fetching data through Route Parameter
app.get('/sensor-data/leakage/:leakageStatus', async (req, res) ⇒ {
        const { leakageStatus } = req.params;
        try {
                const data = await SensorData.find({ leakageStatus });
                if (data.length === 0)  return res.status(404).send('No data found fo
r the specified leakageStatus\n');
                res.status(200).json(data);
        } catch (error) {
                res.status(200).send('Error fetching data\n');
        }
});
```

A

```
// Fetch all data
app.get('/sensor-data', async (req, res) ⇒ {
        try {
                const data = await SensorData.find().sort({ timestamp: -1 }).limit(10
);
                res.status(200).json(data);
        } catch (error) {
                res.status(500).send('Error fetching data\n');
        }
});
// Fetching latest data
app.get('/sensor-data/latest', async (req, res) ⇒ {
        try {
                const latestData = await SensorData.findOne().sort({ timestamp: -1 })
;
                res.status(200).json(latestData);
        } catch {
                res.status(500).send('Error fetching data\n');
        }
});

const PORT = process.env.PORT || 3001;
app.listen(PORT, () ⇒ {
        console.log(`Server running on port ${PORT}`);
});
```

B

Fig.4. (A) HTTP/REST API Topic Configurations from EC2 Instance using Express.js Framework (B) Fetching The Latest Sensor Data & Opening Port For Cross Connection.

At the same time, Amazon EC2 (Elastic Compute Cloud), powered by Ubuntu – Linux based operating system, is used to handle HTTP requests through a REST API. This component is crucial for interfacing with the MQTT messages at a higher level of abstraction. The REST API serves as a bridge, enabling external applications—such as the mobile-based dashboard—to interact with the IoT system using standard HTTP requests. These requests might include commands, queries, or data retrievals. The EC2 instance processes these HTTP requests and coordinates with other services, such as MongoDB. Figure 3B displays the connection configuration between AWS and MongoDB and schema that will be used for the entire API requests. MongoDB is employed as the system's storage solution. After the MQTT messages are received and processed via the REST API, they are stored in MongoDB, which is particularly effective at managing large volumes of unstructured or semi-structured data. This database is essential for maintaining a historical record of sensor data over time. Such historical data is highly valuable for trend analysis, anomaly detection, and predictive maintenance, enabling the system to identify patterns, spot irregularities, and anticipate maintenance needs. These data should be accessible across devices and services by using resource routing URI as defined in Figure 4A. Figure 4B shows the topic configuration to fetch the latest data from MongoDB and opening port for the incoming traffic from EC2 instance and the mobile app.

Building the RESTful API script alone is not enough; it must run continuously as a background task. To achieve this, a package called PM2, a daemon process manager suitable for Node.js applications, should be used. PM2 is a popular process manager for server side scripting as it has the capability to automatically restart the application after crash, manage multiple instances for load balancing, and provide features like logging, monitoring and simple deployment [12].

The mobile-based dashboard acts as the user interface, providing real-time access to the IoT system's data. It efficiently manages both data transmission and retrieval by utilizing the MQTT protocol for live updates and the HTTP/REST API for on-demand queries and interactions. This dual-protocol approach ensures that the dashboard can present real-time data while also enabling users to look at the historical chart for further information of the system condition.

IV. RESULT AND DISCUSSION

A. *Cloud Data*

Dealing with API request requires a lot of testing to ensures the overall connectivity of system. Postman is one of the available design, build, test and collaborative API platform which can visualize the steps of how the API request been made on the internet.

B. *User Interface*

Figure 6A-C illustrates some samples of user interfaces (UI) made in React Native built on top of JavaScript supported with several packages including Axios (promise based HTTP client) and MQTT-connection packages. Figure 12 portrays the key interface of the system which includes the NodeMCU status panel, water level gauge meter, water pump status, leak detection prompt panel and historical water level chart.

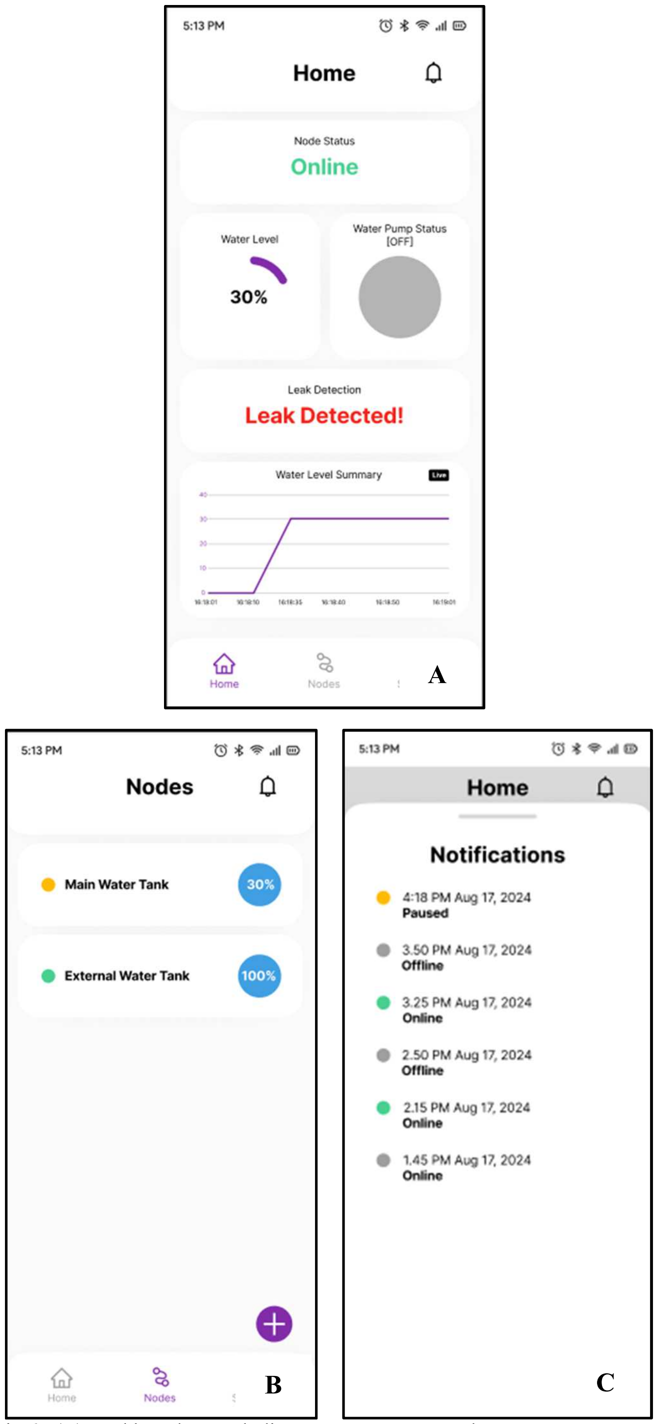


Fig.6. (A) Dashboard UI Including Current Water Level, Water Pump Status, Leakage Status & Water Level Summary (B) Nodes UI: Collection of Multiple Nodes (C) Notifications UI: Display The Timestamp of a Specific Node.

C. Data Security

When dealing with IoT systems, especially those handling critical infrastructure, are often vulnerable to security breaches, making data security a significant concern.

Despite having an enhanced processing power over its predecessor, ESP32 does include secure boot, flash encryption

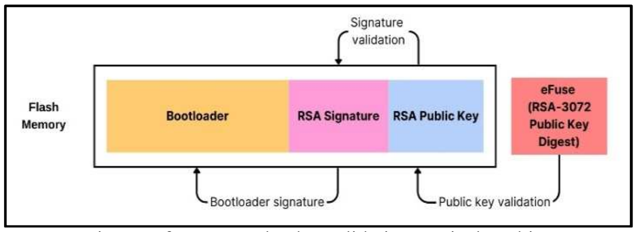


Fig.7. Software Bootloader Validation Logical Architecture

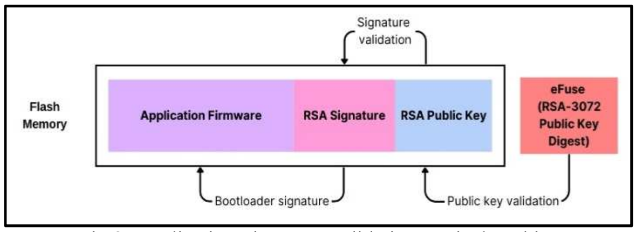


Fig.8. Application Firmware Validation Logical Architecture

and cryptographic hardware acceleration which supports Advanced Encryption System (AES), Hash (SHA-2), Rivest-Shamir-Adleman Encryption Algorithm (RSA), Elliptic Curve Cryptography Algorithm (ECC) and Random Number Generator (RNG) included in its package. Secure Boot is a feature that will ensure only trusted firmware can be run on the chip, preventing unauthorized code being executed. This means that after the compilation of firmware, it will be signed with a cryptographic key known only to the firmware developer. The bootloader of ESP32, responsible for loading the main firmware, will verify and validate the integrity and authenticity of the firmware by checking the digital signature using corresponding public key (RSA-3072) stored securely in the device and can be generated only once during the manufacturing [13]. With that, the device is immune to passive side attacks as there is no corresponding RSA-PSS private key stored on the chip. Additionally, configuring Flash Encryption adds an extra layer of security where it is used to encrypt the contents of ESP32's flash memory. Figure 7 illustrates the logical architecture of software bootloader validation process.

The ROM code looks up the Public Key, stored in 3072-bit format, from the Bootloader's image and cross-validate with the digest in the eFuse. The ROM then transfer the execution control to the Bootloader and begins the application firmware validation as shown in Figure 8.

Not only limited to physical security of the hardware, AWS also plays a crucial role in ensuring the protection of data over the air by implementing robust security measures. A good example of such implementation in AWS is the transport layer security protocols known as TLS for protection of data in transit from attacks like eavesdropping and man-in-the-middle attacks. Besides, access to AWS resources can be tightly controlled; for instance, AWS provides a facility for the restriction of SSH access to AWS EC2 instances to particular IP addresses, thereby reducing the chances of unauthorized access. Besides, MQTT at AWS IoT Core depends on keys and certificates to provide security regarding communication between IoT devices and AWS, ensuring that only authenticated devices are able to transmit and receive data. This is such a multilayered approach to security that on one hand protects the data in transit but at the same time authenticates the devices to provide an integrated protection against the threats.

These security features are helpful and becomes essential to ensure the integrity, confidentiality and authenticity of the data being transmitted are preserved.

V. CONCLUSION

After all, this paper addresses critical water management challenges in Malaysia, particularly the issues stemming from aging infrastructure and inefficient systems that compromise both water quality and infrastructure stability. The IoT-Based Water Management System (IWMS) introduced here provides a transformative solution by leveraging advanced technologies to effectively tackle these problems.

ACKNOWLEDGMENT

This work is supported by the Department of Computer Sciences, KICT, IIUM, Centre of Excellence Cybersecurity, KICT, IoTeams, KICT and Silverseeds Lab Network.

REFERENCES

[1] I. Martínez, B. Zalba, R. Trillo-Lado, T. Blanco, D. Cambra, and R. Casas, "Internet of Things (IoT) as Sustainable Development Goals (SDG) Enabling Technology towards Smart Readiness Indicators (SRI) for University Buildings," *Sustainability*, vol. 13, no. 14, p. 7647, Jul. 2021, doi: 10.3390/su13147647.

[2] G. Asirvatham, "A Study on Deficiencies Causing Water and Energy Losses in the Roof Top Water Storage Tank Installations in India," *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 6, no. 3, pp. 871–881, Mar. 2018, doi: 10.22214/ijraset.2018.3138.

[3] G. Freni and L. Liuzzo, "Effectiveness of Rainwater Harvesting Systems for Flood Reduction in Residential Urban Areas," *Water*, vol. 11, no. 7, p. 1389, Jul. 2019, doi: 10.3390/w11071389.

[4] "What is Cloud Computing?," Google Cloud. Accessed: May 02, 2024. [Online]. Available: https://cloud.google.com/learn/what-is-cloud-computing

[5] G. A. Oguntala and R. A. Abd-Alhameed, "Systematic Analysis of Enterprise Perception towards Cloud Adoption in the African States: The Nigerian Perspective," vol. 9, no. 4.

[6] R. Payne, "Developing in Flutter," in *Beginning App Development with Flutter*, Berkeley, CA: Apress, 2019, pp. 9–27. doi: 10.1007/978-1-4842-5181-2_2.

[7] A. B., "DATABASE CONNECTOR: A TOOL FOR MANIPULATION ON DIFFERENT DATABASES," *Int. J. Res. Eng. Technol.*, vol. 04, no. 22, pp. 7–10, Sep. 2015, doi: 10.15623/ijret.2015.0422003.

[8] C.-H. Lee and Y.-L. Zheng, "SQL-to-NoSQL Schema Denormalization and Migration: A Study on Content Management Systems," in *2015 IEEE International Conference on Systems, Man, and Cybernetics*, Kowloon Tong, Hong Kong: IEEE, Oct. 2015, pp. 2022–2026. doi: 10.1109/SMC.2015.353.

[9] "NoSQL Vs SQL Databases," MongoDB. Accessed: Aug. 25, 2024. [Online]. Available: https://www.mongodb.com/resources/basics/databases/nosql-explained/nosql-vs-sql

[10] A. Kanade, A. Gopal, and S. Kanade, "A study of normalization and embedding in MongoDB," in *2014 IEEE International Advance Computing Conference (IACC)*, Gurgaon, India: IEEE, Feb. 2014, pp. 416–421. doi: 10.1109/IAdCC.2014.6779360.

[11] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 26, no. 1, pp. 64–69, Jan. 1983, doi: 10.1145/357980.358007.

[12] V. F. De Oliveira, M. A. D. O. Pessoa, F. Junqueira, and P. E. Miyagi, "SQL and NoSQL Databases in the Context of Industry 4.0," *Machines*, vol. 10, no. 1, p. 20, Dec. 2021, doi: 10.3390/machines10010020.

[13] A. B. M. Moniruzzaman and S. A. Hossain, "NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison," 2013, *arXiv*. doi: 10.48550/ARXIV.1307.0191.

[14] B. Wukkadada, K. Wankhede, R. Nambiar, and A. Nair, "Comparison with HTTP and MQTT In Internet of Things (IoT)," in *2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*, Coimbatore: IEEE, Jul. 2018, pp. 249–253. doi: 10.1109/ICIRCA.2018.8597401.

[15] A. Ehsan, M. A. M. E. Abuhaliqa, C. Catal, and D. Mishra, "RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions," *Appl. Sci.*, vol. 12, no. 9, p. 4369, Apr. 2022, doi: 10.3390/app12094369.

[16] T. Ambler and N. Cloud, *JavaScript Frameworks for Modern Web Dev*. Berkeley, CA: Apress, 2015. doi: 10.1007/978-1-4842-0662-1

[17] "Secure Boot V2 - ESP32 - — ESP-IDF Programming Guide v5.3.1 documentation." Accessed: Sep. 24, 2024. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/security/secure-boot-v2.html