IMPLEMENTING SERVER-SIDE FEDERATED LEARNING IN AN EDGE-CLOUD FRAMEWORK FOR PRECISION AQUACULTURE

BY

BRYAN NG JING HONG

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONOURS)

Faculty of Information and Communication Technology

(Kampar Campus)

JUNE 2025

COPYRIGHT STATEMENT

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Ts Tan Teik Boon who has given me a lot of guidance in order to complete this project. When I was facing problems in this project, the advice from him always assists me in overcoming the problems. A million thanks to you.

Finally, I must say thanks to my parents and my family for their love, support, and continuous encouragement throughout the course.

ABSTRACT

This project is about a growing need in the aquaculture industry which is precision aquaculture. Precision aquaculture involves the use of smart technologies such as sensors, cloud computing, and artificial intelligence to monitor and manage fish farming environments. However, small-scale farmers still face major challenges such as high implementation costs, poor internet connectivity, and concerns about data privacy. This project focuses on two key problems which are data privacy and system scalability. Most existing systems rely heavily on cloud connectivity and do not provide secure or efficient solutions for farms in remote areas with limited internet access. In this project, research and literature reviews were conducted to explore the current technologies in smart aquaculture, federated learning, and data privacy. Reviews include systems using IoT and AI-based models, along with analysis of different federated learning algorithms such as FedSGD, FedAvg, and FedProx, and privacy-preserving methods such as DA, SA, HE and CKKS encryption. After identifying the gaps in current systems, this project proposes a secure and scalable server-side federated learning framework in an edge-cloud architecture for precision aquaculture. The system is designed to enable encrypted model training at the edge using IoT sensors and Raspberry Pi, while a federated learning server hosted on AWS aggregates updates without accessing raw data. The final product integrates edge computing, federated learning, encryption and cloud storage to create a privacy-preserving and cost-effective solution for monitoring aquaculture environments and supporting small-scale farmers.

Area of Study: Distributed Systems, Federated Learning

Keywords: Precision Aquaculture, Edge-Cloud Framework, Federated Learning, Encryption, Algorithm, IoT

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

μ	Proximal term's regularization parameter
w	Local model weights/parameters
w^t	Global weights of server of epoch t
n_k:	Number of data samples on client k
K	Total number of participating clients
w_{t+1}	Updated global model weighs after round t+1
$w_{(t+1)}^k$	Updated model weights from client k after local training
$F_k(w)$	Local loss function
$h_k(w; w^t)$	Objective function to minimize
y_i	Actual Dissolved Oxygen Value
$\widehat{y_i}$	Model's Predicted Value of Dissolved Oxygen
θ	Model parameters
η	Learning rate
E	Error term
λ	Hyperparameter for penalty term
α	Regularization parameter
Δ	Change or update amount

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
AIoT	Artificial Intelligence of Things
AWS	Amazon Web Services
CKKS	Cheon-Kim-Kim-Song (Homomorphic Encryption Scheme)
DP	Differential Privacy
ECS	Elastic Container Service
FedAvg	Federated Averaging
FedProx	Federated Proximal
FedSGD	Federated Stochastic Gradient Descent
FL	Federated Learning
FLWR	Flower (Federated Learning Framework)
gRPC	Google Remote Procedure Call
HE	Homomorphic Encryption
JSON	JavaScript Object Notation
MAE	Mean Absolute Error
MSE	Mean Squared Error
MQTT	Message Queuing Telemetry Transport
R^2	Coeeficient of Determination
SA	Secure Aggregation
TLS	Transport Layer Security

CHAPTER 1

Introduction

1.1 Problem Statement and Motivation

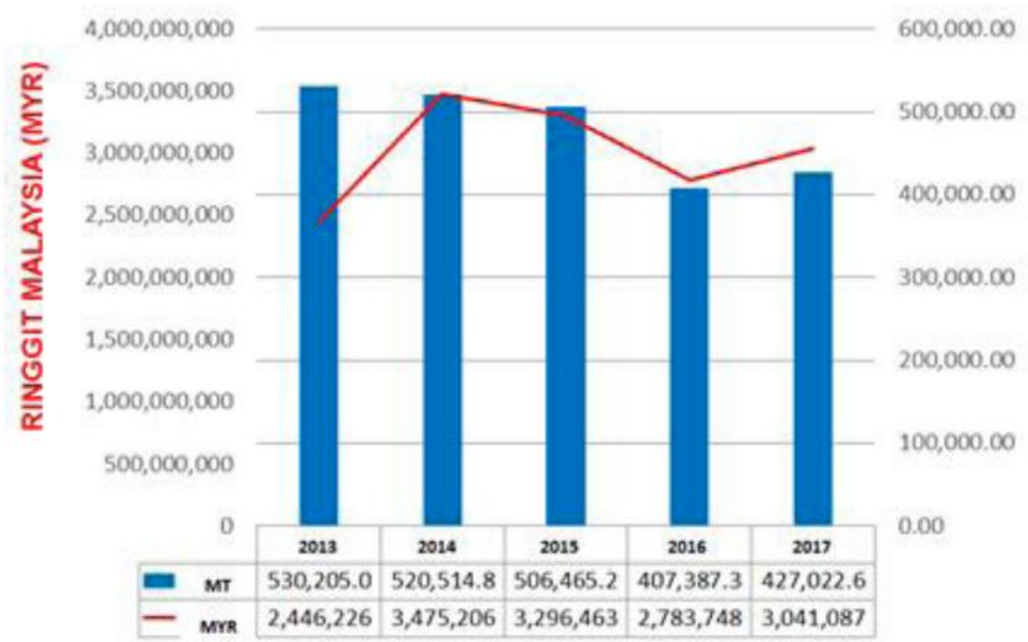


Figure 1.1.1 Aquaculture Production in Malaysia from 2013 to 2017 from [1]

Nowadays, aquaculture industry plays a significant role in contributing a country's economy [1], and it is a lifeline for many communities and a key pillar of Malaysia's economy. As shown in the figure 1.1.1, Malaysia's aquaculture sector has consistently contributed significantly to the national economy and the overall economic value remains high with only minor fluctuations although production volume shows a declining trend from 2013 to 2017. With increasing demands for sustainable fish production, precision aquaculture has emerged as a revolutionary approach, empowering farmers to monitor and manage their ponds using advanced sensors and smart technologies. However, the adoption of these expensive and hard-to-get technologies for precision aquaculture among small-scale farmers is limited due to cost, resource constraints, data privacy concerns and technical barriers [1]. It may cause their productions drop without having proper advance method for farm monitoring and management. Besides, while Internet of Things (IoT) devices have transformed farm monitoring, they bring new challenges which are data privacy, poor connectivity in rural areas, and delayed responses. Hence, traditional cloud-based models cannot always keep up with the real-time demands of aquaculture operations.

The project tackles these limitations by proposing an improved solution which is server-side federated learning within an edge-cloud framework specifically designed for precision aquaculture to ensure effective and secure client-server communication in an edge-cloud framework. Furthermore, the project also aims to implement a scalable, cost-effective, well comprehensive and accurate monitoring farm management system that applies the integration of federated learning and edge-cloud platforms while ensuring data privacy or performance. Other than that, the project will also control the technical complexity of adoption to make sure it is adaptable for the small scale farmer.

1.2 Project Objectives

The main project objective is to enhance and deploy the Server-Side Federated Learning Models within an Edge-Cloud Framework that support broader applicability in precision aquaculture. To explain, the project aims to enhance and deploy Federated Learning Models alongside local deep learning models to improve on-device data processing, predictive performance, and scalability. This includes ensuring that the system can support for multiple aquaculture farms and large number of edge nodes with heterogeneous data distributed across different geographical locations.

Besides, the project aims to refine and implement a flexible and robust data synchronization mechanism to ensure efficient and reliable data transmission and model updates between edge clients and the cloud, even in the presence of intermittent or unstable network connections commonly found in rural aquaculture settings. This mechanism is essential to maintain seamless operations for data backup. By having this, model updates can be achieved periodically to enable the system to learn from new data and adapt to changing conditions in the aquaculture environment, enhancing the effectiveness of precision aquaculture practices. Additionally, reliable data synchronization mechanism also improves data reliability as it reduces the risk of data loss or corruption, which is critical for maintaining and preserving the integrity of farm operations.

Furthermore, the project will improve and implement a secure, efficient, and privacy-preserving edge-cloud framework tailored for precision aquaculture. The framework will address critical challenges related to security and performance. For example, when doing operation on data, the framework will be able to protect sensitive data related to farm management system like environmental conditions from unauthorized access or breaches while simultaneously ensuring that the system performance is not compromised. By integrating lightweight security and privacy-preserving techniques, the proposed framework will allow farmers to leverage the benefits of modern data-driven approaches without being exposed to excessive risks or high infrastructure costs.

1.3 Project Scope and Direction

The project scope focuses on implementing server-side federated learning framework within an edge-cloud architecture, while also enhancing client-side learning to achieve robust, secure, and efficient precision aquaculture applications. Specifically, the project covers that aspects below.

Firstly, the project will establish a robust communication bridge between federated learning clients and central server to support model updates for precision aquaculture. This communication mechanism is designed function reliably while ensuring reliable data transmission and model updates even in farms with poor network conditions to ensure the essential updates are not disrupted. Additionally, this project will also emphasize energy conservation and latency reduction during data communication between federated learning clients and federated learning server to further optimize system performance and operational efficiency.

Besides, this project develops federated learning algorithms with improved workflows to perform aggregation on the local client updates to form global model at the server then send it back to clients to improve their local model performance of the distributed clients. To achieve the improvement on predictive performance, federated learning algorithm also applies into the edge side to incorporate with the server. For example, Federated Proximal will be integrated to edge to address challenges of heterogenous data and resource variability and ensure more stable and efficient local training. In addition, a deep learning network is also implemented at edge side to improve local predictive performance for aquaculture data.

Other than that, this project develops a privacy-preserving data transmission method between clients and server. For example, encryption technique like CKKS encryption is utilized in model updates between federated learning clients and federated learning server to protect and support secure and faster transmission from unauthorized access. By focusing on these improvements, the project aims to create a more accessible and cost-effective solution for small-scale aquaculture farmers.

By focusing on these areas, this project aims to deliver a scalable, cost-effective, and privacy-preserving edge-cloud framework. Ultimately, the project scope ensures that small-scale aquaculture farmers can leverage advanced AI technologies for smarter, more sustainable, and efficient farm management.

1.4 Contributions

The main contribution is pioneering application of federated learning in aquaculture. For example, by adapting federated learning to environmental and technical constraints of fish farming, this project sets a solid foundation for future innovations in smart agriculture and aquaculture.

Secondly, this project contributes by solving real-world challenges faced in aquaculture environments. To explain, the customized synchronization mechanism is developed to addresses poor connectivity problem while preserving the privacy of farm sensitive data by integrating encryption technique named CKKS encryption into the system.

Other than that, this project improves the accessibility of technology for the small-scale farmer. This is because the system lowers the cost and complexity of adopting smart farm monitoring system so it will be viable for small and medium farms. Lastly, a wider adoption of precision aquaculture helps Malaysia to meet its sustainability and productivity goals by boosting the country's economy.

Finally, this project contributes at a broader level by supporting Malaysia’s sustainability and productivity goals. To explain, wider adoption of precision aquaculture technologies can boost efficiency, improve resource management, and strengthen the aquaculture sector’s contribution to the national economy.

1.5 Report Organization

The project chapters are organized in seven chapters. In Chapter 2, smart aquaculture technologies, algorithms, data protection method and interval update in federated learning are reviewed. Then, Chapter 3 is System Methodology which contains the system methodology, use case diagram and activity diagram. For use case diagram, use case description is also prepared for it. After that, Chapter 4 presents the system design of the project. Besides, in Chapter 5, it shows the details of system implementation. Then, in Chapter 6, it shows the system evaluation and discussion with rigorous testing involved. Last, for Chapter 7, it contains conclusion and recommendations.

CHAPTER 2

Literature Review

2.1 Previous works on Server-Side Federated Learning in an Edge-Cloud Framework for Precision Aquaculture

2.1.1 Evolution of Smart Aquaculture Systems: From IoT Monitoring to Federated Learning in Edge-Cloud Architectures

The integration of smart technologies into aquaculture has evolved significantly in recent years, addressing limitations of traditional fish farming practices that rely on manual and error-prone monitoring of environmental factors. Several innovative systems have emerged that leverage Internet of Things (IoT), AI, edge technology, and cloud services to transform aquaculture into more sustainable, efficient, and scalable operations.

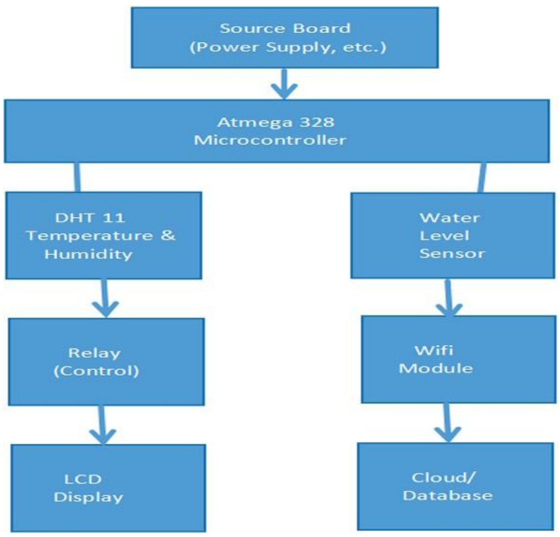
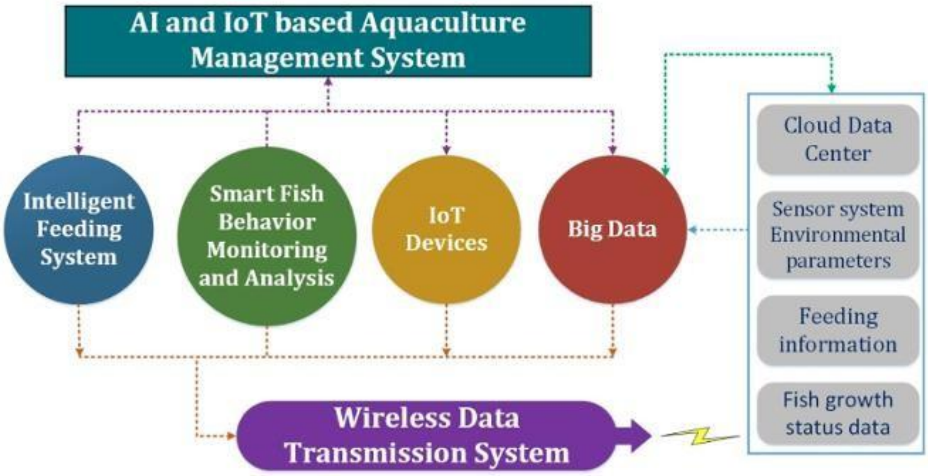


Figure 2.1.1.1: Block Diagram of the IoT Aquaculture Monitoring System

One major direction involves the adoption of smart aquaculture with Internet of Things monitoring systems that automate the collection and analysis of critical environmental parameters as shown in figure 2.1.1.1. Sah et al. [2] proposed a low-cost

IoT aquaculture monitoring system utilizing a variety of sensors, such as the DHT11 for temperature and humidity, and water level sensors, managed via microcontrollers like Arduino Uno. Wireless connectivity through Wi-Fi modules and GSM modems enables real-time data transmission to a cloud database to allow farmers to access timely notifications and maintain optimal water conditions remotely. This system significantly reduces manual labour and human error, improves farm productivity, and supports scalability for operations of varying sizes. However, the study also highlighted limitations which includes data synchronization problem when no Internet connection and a lack of advanced data analytics for predictive insights, dependency on stable



internet connectivity, and the absence of robust data privacy and security mechanisms.

Figure 2.1.1.2: Flowchart of the AIoT System

Building upon basic IoT architectures, more sophisticated approaches have incorporated AI techniques to further optimize aquaculture management as shown in figure 2.1.12. Chiu et al. [3] created a smart aquaculture farm management system by integrating IoT devices with AI-based substitute models to predict fish growth and automate feeding practices. Their system employed multiple sensors, actuators, and IP cameras for real-time monitoring, with data transmitted to a cloud server where deep learning models were used for analysis and prediction. A mobile application was designed to facilitate remote monitoring and control, offering farmers live access to water quality metrics and feeding statuses, alongside alarm notifications for critical thresholds. The integration of predictive models enhanced feeding efficiency, reduced waste, and minimized operational costs while promoting environmental sustainability. Despite these advancements, the approach remained heavily reliant on consistent internet connectivity and cloud services for the data synchronization, and also raised concerns regarding data privacy and security. Furthermore, the system's applicability to species beyond the tested environment requires further validation to ensure generalizability across diverse aquaculture settings.

Subsequent developments have investigated the integration of federated learning with edge-cloud platforms to tackle data privacy, connectivity, and scalability issues. Cheng et al. [4] suggested a precision aquaculture framework that utilizes AWS IoT Greengrass for local data processing and collection at the farm level, along with federated learning to facilitate decentralized model training. In their system, IoT sensors collect water quality and environmental data locally, with machine learning models trained at the edge to ensure sensitive data remains on-site. Model updates are coordinated via the gRPC protocol and exchanged via S3, where a federated server aggregates models from multiple farms using an ECS cluster. After that, the aggregated global model is sent to the farms for continuous improvement. This edge-cloud architecture effectively reduces latency, enhances scalability, and minimizes dependence on constant internet connectivity. Moreover, by keeping raw data local, the system addresses critical data privacy and regulatory concerns. The system also deceloped a data synchronization method. Nevertheless, the framework's effectiveness remains relies on the local data quality and diversity, and limitations like inconsistent sensor accuracy and the narrow range of collected parameters may impact the global model's robustness. In addition, model exchange in S3 is insecure. Periodic stable network connectivity is still required for model aggregation and updates, which may pose challenges in regions with poor infrastructure.

Overall, common obstacles such as internet dependency, data security vulnerabilities, sensor variability, and limited data diversity continue to pose challenges, underscoring the need for further research and system refinement in future smart aquaculture initiatives.

2.1.2 Algorithms for Model Aggregation in Federated Learning

Federated Learning is a method of machine learning for decentralized systems which involves multiple clients as the edge devices that will perform local training and model updates and interact with a server to perform model aggregation for refine the models [5]. Hence, model aggregation algorithm is important in federated learning because it will aggregate the local models updated by clients as a global model then distribute back for client to perform local training to enhance their local models. The final model's performance will be greatly impacted by the aggregation method selection. In this section, model aggregation algorithms will be reviewed and analysed

The first analyzed algorithm is FedSGD (Federated Stochastic Gradient Descent). FedSGD is a baseline approach utilized in federated learning. For FedSGD, there will a server receives the local gradients from a subset of devices in FedSGD and aggregates them [7]. After that, these aggregated gradients are used for updating the global model. Before delivering their model updates, clients will execute several local SGD updates in FedAvg which is a variation of FedSGD [8]. However, when employing non-IID datasets and random device selection in wireless communication systems, one issue with traditional FedSGD is the possibility of weight divergence [7], since the data distribution of the chosen devices is different from the worldwide distribution.

The second algorithm is FedAvg (Federated Averaging) which is a fundamental algorithm that is widely used in FL as it is communication-efficient approach which can works well in decentralized networks [6]. In FedAvg, clients use stochastic gradient descent (SGD) on the local data to perform out multiple local updates of a common global model [8]. They transmit their final models back to the server following these local updates. Then, the server will average the updated local models to form a updated global model [8]. One of FedAvg's primary features is the way it alternates between model averaging at the server and local updates at clients. A weighted average of the models produced by the chosen client is determined by the server.

By letting clients' complete multiple epochs of local updating prior to interacting with the server, FedAvg seeks to improve communication efficiency [9]. Despite being straightforward and widely used, research has indicated that FedAvg may not perform well when statistical heterogeneity is present, , particularly when working with diverse data from various devices [10]. The method may fail to converge or maximize a different objective from the global one as a result of the numerous local updates, which can also result in weight divergence, gradient biases, and objective inconsistencies [6]. Empirical research indicates that the FedAvg output model can generalize effectively in diverse environments, particularly when paired with client-specific fine-tuning, despite these convergence issues with data heterogeneity.

The third analyzed algorithm is FedProx (Federated Proximal). FedProx is an extension of FedAvg method to address the issue of heterogeneous data among clients. It is a well-known and effective distributed proximal point optimization technique that is frequently applied to FL over heterogeneous data [10]. Recently, there has been a proposal to solve the problem that when devices execute too many local updates in the presence of data heterogeneity, approaches such as FedAvg based on local SGD may not converge [10]. FedProx solves this by including a proximal term in each device's local optimization target. Then, more stable local updates are produced as a result of this term's explicit enforcement of the local optimization to remain close to the prior global model [10].

Besides, [10] mentioned that FedProx is tolerant of devices' partial participation and even varying amounts of local updates, and it provides convergence guarantees for both convex and non-convex functions. The proximal point update for local optimization is described as an empirical risk minimization (ERM) sub-problem [6]. In the limiting case, when the influence of the proximal term approaches zero, FedProx reduces to the standard FedAvg framework.

Comparative Analysis

Feature	FedSGD	FedAvg	FedProx
Server Aggregation	Aggregates local gradients	Averages local model weights	Averages local model weights with stabilized updates
Communication Efficiency	Low	High	High
Handling Non-IID Data	Weak	Moderate	Strong
Convergence Stability	Weak	Moderate	Strong
Robustness to Partial Participation	Limited	Limited	Strong

Table 2.1.2.1: Comparison of FedSGD, FedAvg and FedProx

Table 2.1.2.1 summarizes the differences of FedSGD, FedAvg and FedProx in terms of practical considerations. Based on the analysis, FedAvg and FedSGD are fundamental FL algorithms, with FedAvg being a widely adopted approach that allows clients to perform multiple local updates [6]. However, both conventional FedAvg and FedSGD can face challenges with data heterogeneity data, potentially leading to convergence issues or weight divergence [6]. In this case, FedProx would be able to solve the convergence and stability problems faced by FedAvg and FedSGD. This is because the proximal term introduced by FedProx will provide a more stable updates by ensuring the local models stay close to the global model [10]. Other than that, according to recent theoretical research, FedProx can handle non-smooth functions, and its convergence is invariant to specific forms of local data dissimilarity, addressing the drawbacks of earlier analyses [10]. Last, it is able to tolerate partial participation of devices.

2.1.3 Data Security Protection Methods in Federated Learning

As federated learning (FL) becomes more important for privacy-sensitive applications like precision aquaculture to protect the security and confidentiality of client data during training. So, the data security protection methods in federated learning have become a key research focus. Among various techniques proposed, there are three techniques have emerged as the most widely adopted strategies in federated learning. This section reviews and compares these methods supported by recent studies.

First, Differential Privacy (DP), it protects individual data contributions by introducing regulated noise to model updates prior to transmitting them to the server. For example, Truex et al. [11] mentioned that by integrating local differential privacy into FL, it could limit potential information leakage, even when servers or participants are compromised. Similarly, Geyer et al. [12] also incorporated differential privacy into Google's FL framework for mobile devices and demonstrated that it can provide a strong user-level privacy guarantees with acceptable model performance trade-offs. Despite its theoretical soundness, adding sufficient noise under DP can result in a privacy-utility trade-off: too much noise will result in a degraded model accuracy, which is problematic for systems working with high prediction accuracy, such as fish health monitoring for aquaculture.

Second, Secure Aggregation (SA), the server can calculate the total of several clients' model updates using Secure Aggregation without being aware of each client's unique contribution. For example, in order to provide a workable cryptographic technique that attains security without incurring excessive communication costs, Bonawitz et al. [13] improved SA protocols for large-scale federated networks. Additionally, secure model update aggregation across more dynamic client populations was made possible by Ryffel et al. [14]'s expansion of secure aggregation frameworks through the use of multiparty computation techniques. SA provides robust protection without compromising model accuracy, but its heavy computation and client synchronization requirements may pose a problem for edge devices with constrained resources, like aquaculture farms enabled by the Internet of Things.

Moreover, the third method is Homomorphic Encryption (HE) which is an encryption technique that allows server to direct making computation on encrypted data [15]. It allows model updates to remain confidential throughout transmission and aggregation. For example, after [15] surveyed privacy-preserving techniques in FL, their research concluded that HE provides one of the highest levels of protection but at the cost of increased computational complexity. However, recent lightweight HE libraries such as TenSEAL have optimized tensor encryption for machine learning tasks, significantly lowering latency. HE via TenSEAL offers a good balance between security and practical deployment.

In addition to the general application of HE in federated learning, a more practical variation for resource-constrained environments is the Shared-Context CKKS Hybrid Encryption scheme approach. The shared-key design enables clients and the server to use the same context for encrypting and decrypting model updates, in contrast to multi-key or threshold HE schemes that require complex key management and costly computation. This simplifies implementation and substantially reduces computational overhead, making it feasible for lightweight devices such as Raspberry Pi in aquaculture farms. The shared-key CKKS method prioritizes efficiency and deployability enabling secure transmission and aggregation of model parameters with much lower computational costs. For instance, [20] introduced a single key HE framework for federated learning and which demonstrates that adopting a shared secret key among clients and the server can achieve secure aggregation with reduced computational complexity compared to multi-key approaches. Their results confirm that the server can still decrypt the aggregated model update while preventing exposure of individual client updates under semi-honest assumptions. This method balances security and usability, especially in small-scale federated settings where lowering overhead is crucial for adoption.

Comparative Analysis

Criteria	Differential Privacy	Secure Aggregation	Homomorphic Encryption	Shared Context CKKS Hybrid Encryption
Method	Introduce regulated noise to model updates prior to transmitting them to the server	Calculate the total of several clients' model updates while ignoring each client's unique contribution	Allows server to direct making computation on encrypted data	Use a single shared context for both clients and server to encrypt, decrypt, and aggregate model updates
Communication Overhead	Low	High	Medium	Medium–Low
Model Accuracy	Low	High	High	High
Stability in Intermittent Network	High	Low, many clients must be online at the same time	High, clients can send updates whenever they are connected	High, clients can send updates whenever they are connected
Computation Overhead	Low	High	High	Medium
Server Trust assumption	No	No	No	Yes

Table 2.1.3.1: Comparison between DP, SA, HE and Shared Context CKKS Hybrid Encryption

Based on table 2.1.3.1, when considering stability in intermittent networks, DP offers high stability since each client can operate independently without strict synchronization requirements. For Homomorphic Encryption and Shared-Context CKKS, they provide high stability by allowing clients to send encrypted updates whenever they are connected, which is particularly useful in unstable network environments. However, Secure Aggregation shows low stability because it depends on the simultaneous availability of many clients to perform the aggregation securely. Lastly, Shared-Context CKKS need a "Trustable Server" assumption as server will need to decrypt the received encrypted parameters to perform aggregation whereas other technique does not need this.

2.1.4 Reviews on Interval Updates for Model Updates/Aggregation in Federated Learning in Smart Aquaculture System

In federated learning for precision aquaculture, model updates interval is critical and must be determined appropriately for ensuring the system performance. To determine the interval, communication cost, energy consumption and model accuracy will be the factors that need to be considered. There is a previous implementation of federated learning in smart aquaculture systems proposed a bi-weekly model updates to update the cycle. Their findings revealed that environment factors like PH and dissolved oxygens and so on can dramatically change over time. Hence, [16] suggested bi-weekly update in order to keep model relevance and energy conservation and bandwidth resource in balance.

Besides, [17] also suggests a bi-weekly updates offered the optimum balance between preserving model accuracy and lowering system costs, especially in dynamic environmental conditions like seasonal fluctuations with a real-world aquaculture IoT study. The findings in [17] revealed that bi-weekly aggregation provided the optimal balance between model flexibility and operational costs after they implemented weekly, biweekly and monthly updates experiment.

In contrast, Zhao et al. [18] proposed a longer interval which is a monthly update for environmental monitoring system. This is because it claimed that the slow changes in environmental parameters like temperature and turbidity can reduce the communication overhead. Hence, an infrequent model aggregation in this case will be more efficient with no degrading the model predictive performance.

Comparative Analysis

Criteria	Bi-Weekly Updates	Monthly Updates
Model Freshness	High	Low
Adaptation to environmental changes	Strong	Weak
Communication Overhead	Moderate	Low
Accuracy	High	Low
Energy Consumption	Moderate	Low

Table 2.1.4.1: Comparison of Bi-Weekly and Monthly Model Update

To further illustrate the differences between bi-weekly and monthly model update intervals, a comparison based on several critical criteria is summarized in Table 1. The table highlights factors such as model freshness, communication overhead, energy consumption, adaptability to environmental changes, and overall suitability for precision aquaculture systems. Based on the comparative analysis, the bi-weekly (14-day) interval clearly offers a superior balance between model performance and system efficiency, aligning closely with the dynamic operational requirements of precision aquaculture environments.

2.2 Proposed Solutions

After reviewing some existing smart aquaculture systems, several challenges remain evident. The current smart aquaculture systems continue to experience significant problems in the form of unstable network connectivity resulting in loss of data or delays, lack of sufficient mechanisms to secure sensitive data during transmission, lack of robustness in aggregation techniques in case of heterogeneous farm data, and inefficiency in scheduling model updates that directly influence accuracy, communication costs, and energy usage.

To tackle these limitations, this project presents a federated learning framework based on a server-side architecture that is customized to precision aquaculture. The framework incorporates the FedProx algorithm to enhance the stability of aggregation in heterogeneous data, and it uses an edge cloud architecture, which relies on the AWS services to offer scalability, reliability, and ease of management. The deployment of the federated learning server using the AWS ECS and the Fargate can be installed to enable the system to orchestrate and provide effective client-server communication without exposing raw farm data.

There are a few major improvements that were realized in the proposed solution. First, a Shared-Context CKKS Hybrid Encryption scheme that maintains data privacy but is lightweight enough to be used on the edge device, such as Raspberry Pi, is used to protect model parameters in transit. Second, a deep learning predictive model is used to predict dissolved oxygen level on the basis of water quality parameters to support the use of real-time decisions in the management of aquaculture. Third, the training process is planned on a biweekly basis in order to make a balance between the model freshness and the communication and energy efficiency. Lastly, a strong synchronization pipeline is configured in AWS Cloud, which allows to transmit and store the data reliably even with the intermittent network connections. To conclude, FedProx aggregation, CKKS encryption, predictive modeling, efficient scheduling, and a scalable synchronization pipeline provide a federated learning system that is safe, resource-sensitive, and feasible to an aquaculture farmer on a small scale.

CHAPTER 3

System Methodology/Approach

3.1 Waterfall Methodology

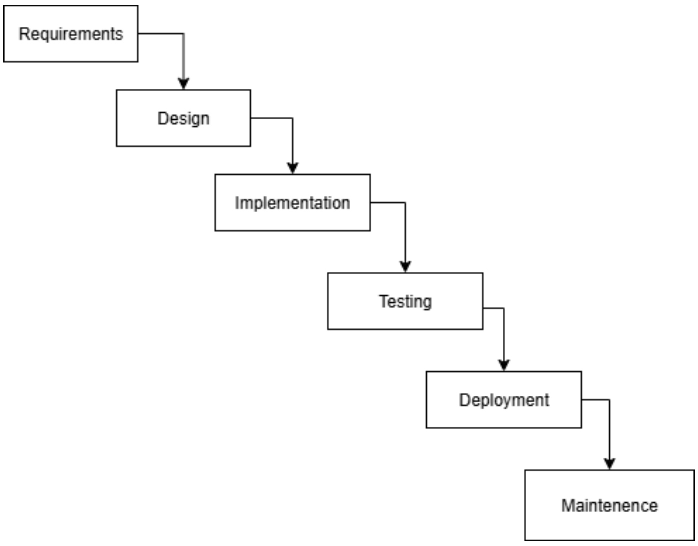


Figure 3.1.1: Waterfall Methodology

This project follows a traditional Waterfall development model. It is chosen for its clear, sequential structure and suitability where objectives and deliverables are defined early [21]. Firstly, the project begins with a requirements analysis to capture functional and non-functional needs and to produce a signed requirements specification that sets acceptance criteria.

Next, the system design phase produces high-level and detailed artifacts like architecture diagrams, data flows, encryption method design and MQTT specifications which act as the blueprint for implementation.

The Implementation phase then builds the federated learning at local. Federated learning code for client and server are developed then build the server to the AWS Fargate. Then, use local to integrate with the server to check if the implementation is workable before integration and deployment.

After that, the project starts integration and system testing, where components are combined and validated through end-to-end tests such as encrypted parameter round-trips, aggregation correctness, intermittent-connectivity resilience and performance benchmarks like MAE, MSE and R^2, communication and resource usage.

After successful testing, the deployment phase places the system into the target environment to deployment the client on the real edge devices which are raspberry pi and completes acceptance testing with supervisor. Finally, maintenance takes place to make improvement and produces operation manuals at the same time.

3.2 Use Case Diagram and Description

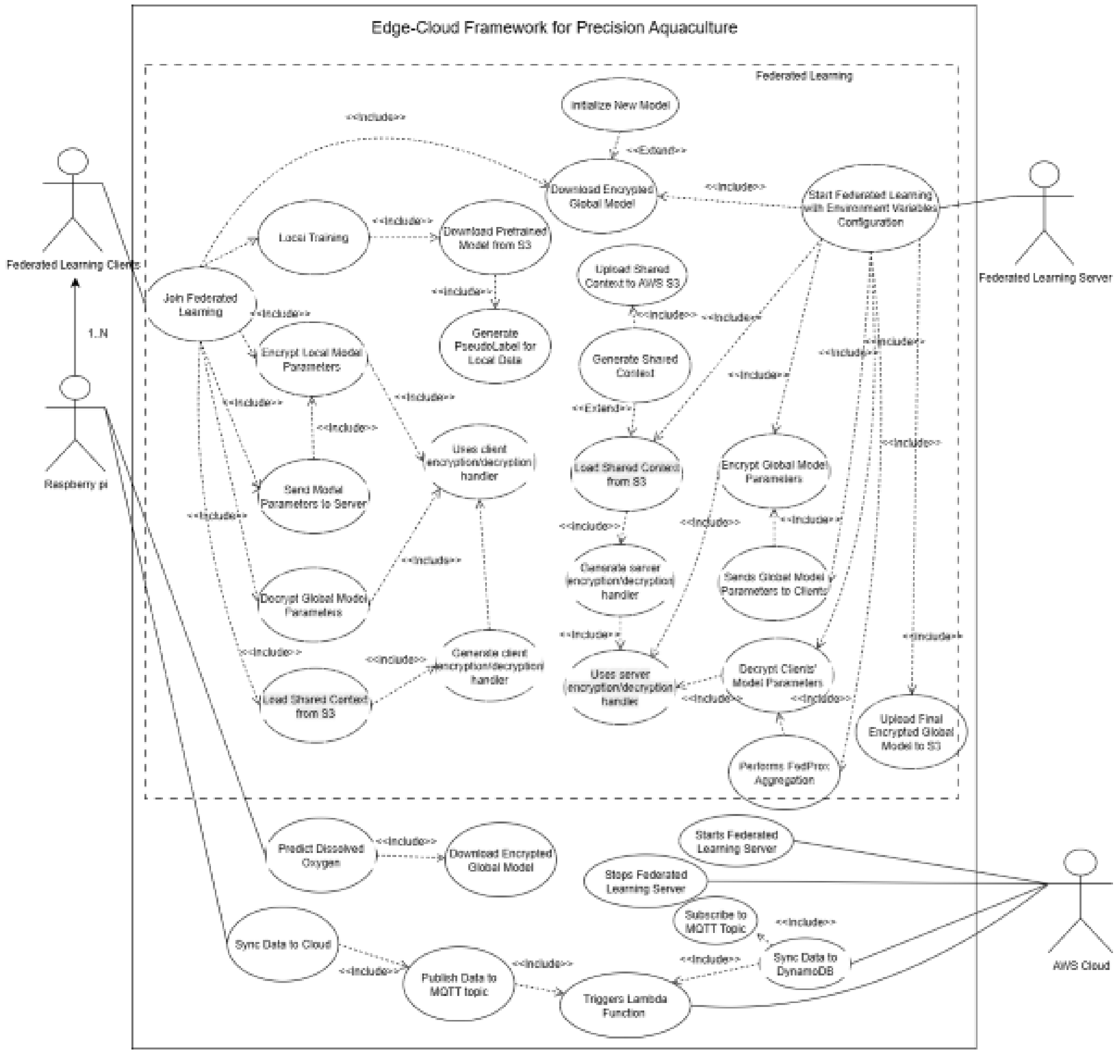


Figure 3.2.1 Use Case Diagram of Edge-Cloud Framework for Precision Aquaculture

Use Case ID	0001	**Use Case Name**	Start Federated Learning with Environment Configuration
Primary Actor	Federated Learning Server		
Brief Description	This use case shows how the federated learning server coordinates collaborative machine learning training with multiple clients and ensure data are decentralized for precision aquaculture monitoring.		
Trigger	Scheduled training cycle begins, or system administrator manually initiates federated learning round.		
Precondition	The services work behind the federated learning server had been configured properly		
Relationships	Association: Federated Learning Server, Federated Learning Client Include: Download Encrypted Global Model, Load Shared Context from s3, Encrypt Global Model Parameters, Decrypt Clients' Model Parameters, Perform FedProx Aggregation, Upload Final Encrypted Global Model to S3		
Scenario Name	**Step**	**Action**	
Main Flow	1	Server initializes new federated learning round and broadcasts invitation to registered edge clients	
	2	Available Raspberry Pi devices respond with participation confirmation and readiness status	
	3	Server load shared context from S3 then generate server encryption/decryption handler	
	4	Server downloads current encrypted global model from S3, decrypt and prepares encrypted model parameters	
	5	Server sends encrypted global model parameters to all participating edge devices	
	6	Once received updates from clients, server decrypt the, then performs federated aggregation (FedProx) on received model updates	
	7	Server logs the performance of training.	
	8	Server stop aggregation when federated learning rounds complete.	
	9	Server uploads final encrypted global model back to S3 storage	
	10	Server notifies all clients of successful training completion and allows them to disconnect.	
Sub Flow 1 – Generate Shared Context	3a.1	Server generate shared context automatically	
	3a.2	Server initialize new encryption/decryption handler by the context	

Sub Flow 2 – Upload Shared Context to S3	3b	After shared context is generated, server directly upload it to the AWS S3
Sub Flow 3 – Initialize Global Model	4a.1	If no global model in s3, server initializes a new global model
Alternate/Exceptional Flows	4b.1	Error when there is the shared context is not able to decrypt the downloaded encrypted global model.

Table 3.2.1: Use Case Description for Federated Learning Server

Use Case ID	0002	**Use Case Name**	Join Federated Learning
Primary Actor	Federated Learning Clients		
Brief Description	This use case shows the flow after client joins the federated learning		
Trigger	After server starts, clients receive messages and join federated learning automatically or manually.		
Precondition	Federated learning server starts		
Relationships	Association: Federated Learning Clients, Raspberry pi Include: Local Training, Encrypt Local Model Parameters, Send Model Parameters to Server, Decrypt Global Model Parameters, Load Shared Context from S3, Download Encrypted Global Model		
Scenario Name	**Step**	**Action**	
Main Flow	1	Clients join federated learning	
	2	Clients load shared context from S3 then initialize client encryption/decryption handler	
	3	Clients downloads current encrypted global model from S3, decrypt and load model	
	4	Clients download pretrained model from s3	
	5	Clients local train to generate dissolved oxygen value for federated learning and build data loader	
	6	Clients decrypt encrypted global model parameters and local train with local data	
	7	Clients send encrypted and updated local model parameters to server	
	8	Clients exit after federated learning complete	
Sub Flow – Initialize New Model	3a.1	If no global model in s3, client initializes a new model as startup and will be replaced once received server's model	
Alternate/Exceptional Flows	4a	If no pretrained model in s3, then client exits federated learning as no dissolved oxygen as target for training unless client has dissolved oxygen value by deploying the global model at real time for do prediction	

Table 3.2.2: Use Case Description for Client Join Federated Learning

Use Case ID	0003	**Use Case Name**	Start Federated Learning Server
Primary Actor	AWS Cloud		
Brief Description	This use case shows how AWS EventBridge Scheduler in AWS Cloud schedules the federated learning server		
Trigger	Trigger every first and mid of the month		
Precondition	The scheduler has been configured properly		
Relationships	Association: AWS Cloud Include: Start Federated Learning Server with Environment Variables Configuration		
Scenario Name	**Step**	**Action**	
Main Flow	1	AWS EventBridge scheduler starts when 11.30am of first and mid of the month	
	2	AWS EventBridge scheduler changes the desired task count of the ECS service to 1 to start the federated learning server	
Sub Flow – Start Federated Learning	2a	The server is invoked to run federated learning task	
Alternate/Exceptional Flows	2b	If server already starts manually, the scheduler will not invoke	

Table 3.2.3: Use Case Description for AWS Cloud to Start FL Server

Use Case ID	0004	**Use Case Name**	Stop Federated Learning Server
Primary Actor	AWS Cloud		
Brief Description	This case shows how AWS EventBridge Scheduler in AWS Cloud schedules the federated learning server		
Trigger	Trigger every first and mid of the month		
Precondition	The scheduler has been configured properly		
Relationships	Association: AWS Cloud Include: Start Federated Learning Server with Environment Variables Configuration		
Scenario Name	**Step**	**Action**	
Main Flow	1	AWS EventBridge scheduler stops when 12.00pm of first and mid of the month	
	2	AWS EventBridge scheduler changes the desired task count of the ECS service to 0 to start the federated learning server	
Sub Flow – Start Federated Learning	2a	The server is invoked to stop federated learning task	
Alternate/Exceptional Flows	2b	If server already stops manually, the scheduler will not invoke	

Table 3.2.4: Use Case Description for AWS Cloud to Stop FL Server

Use Case ID	0005	Use Case Name	Sync Data into DynamoDB
Primary Actor	AWS Cloud		
Brief Description	This use case shows how clouds assist to sync data from raspberry pi into DynamoDB		
Trigger	Trigger when clients sync data to cloud		
Precondition	1. The MQTT topic has been subscribed to cloud and Lambda function has been declared. 2. Clients publish data to the MQTT Topic		
Relationships	Association: AWS Cloud Include: Subscribe MQTT Topic, Triggers Lambda Function		
Scenario Name	Step	Action	
Main Flow	1	Trigger Lambda function when data is published to MQTT Topic	
	2	Lambda function route data into DynamoDB	
Sub Flow – Write Data into DynamoDB	2a	The sync data update the DynamoDB table	
Alternate/Exceptional Flows	1a	Only happens when client is online and synchronizing data	

Table 3.2.5: Use Case Description for AWS Cloud to Sync Data to DynamoDB

Use Case ID	0006	Use Case Name	Predict Dissolved Oxygen
Primary Actor	Raspberry Pi		
Brief Description	This use case shows how raspberry pi predicts dissolved oxygen at the edge		
Trigger	Trigger when clients local train using global model		
Precondition	Encrypted global model is available on the cloud		
Relationships	Association: Raspberry Pi Include: Download Encrypted Global Model		
Scenario Name	Step	Action	
Main Flow	1	Raspberry pi downloads encrypted global model from s3	
	2	Raspberry pi load shared context and decrypt model	
	3	Raspberry pi uses the global model to predict dissolved oxygen	
Sub Flow –	-	-	
Alternate/Exceptional Flows	1a	Error when there is no encrypted global model in s3	
Alternate/Exceptional Flows	2a	Unable to decrypt the global model if no shared context loaded or the shared context mismatch happens.	

Table 3.2.6: Use Case Description for Raspberry Pi Predicting Dissolved Oxygen

Use Case ID	0007	**Use Case Name**	Sync Data to Cloud
Primary Actor	Raspberry Pi		
Brief Description	This use case shows how raspberry pi sync data to the DynamoDB		
Trigger	Trigger when clients sync data periodically or invoked		
Precondition	Cloud has subscribed to the MQTT Topic that raspberry pi will publish to it.		
Relationships	Association: Raspberry Pi Include: Publish Data to MQTT topic		
Scenario Name	**Step**	**Action**	
Main Flow	1	Raspberry pi run synchronization script	
	2	Script publishes the local sensor data to the MQTT Topic	
	3	Lambda function is triggered when data is published to MQTT Topic	
	4	Lambda function route data into DynamoDB	
Sub Flow –	-	-	
Alternate/Exceptional Flows	3a	Error when uploaded data format does not match with JSON format in lambda function	

Table 3.2.7: Use Case Description for Client Sync Data to Cloud

3.3 Activity Diagram

3.3.1 Activity Diagram for Federated Learning

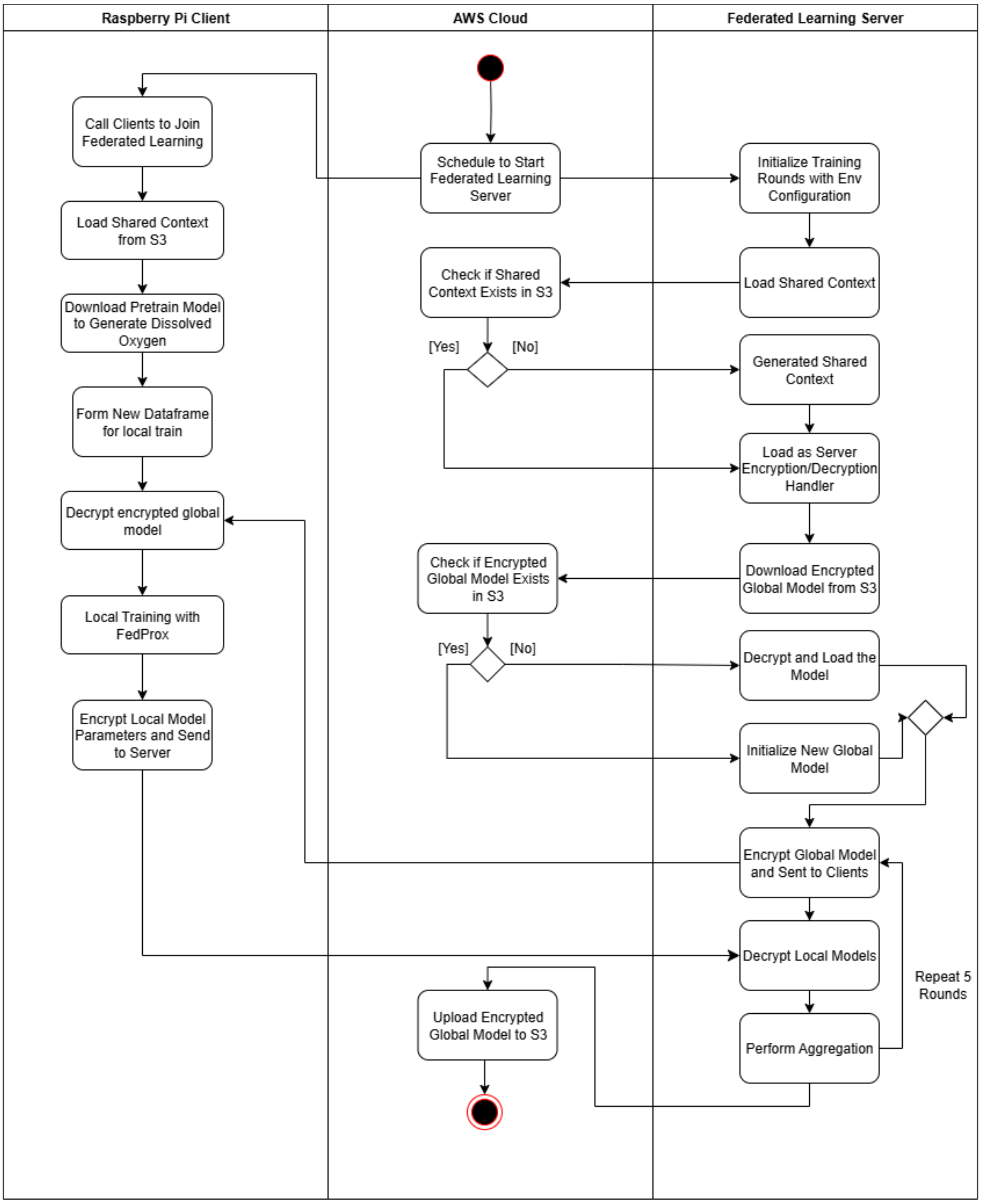


Figure 3.3.1.1: Activity Diagram for Federated Learning

The activity diagram above shows how the federated learning take place within raspberry pi client, AWS Cloud and federated learning server (Fargate) in AWS Cloud.

3.3.2 Activity Diagram for Server

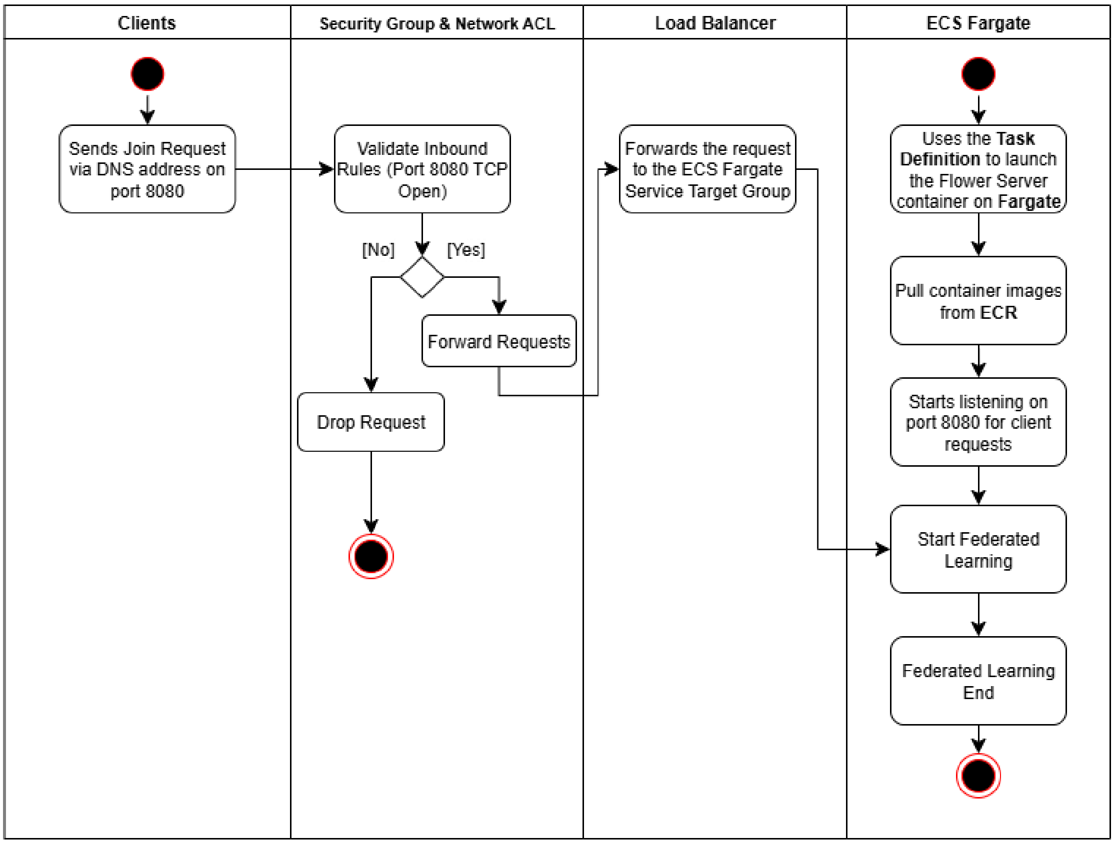


Figure 3.3.2.1: Activity Diagram for Server Components

The activity diagram above shows how the server implementation allows communication between client and server.

CHAPTER 4

System Design

4.1 System Architecture Design

4.1.1 Federated Learning System Flowchart

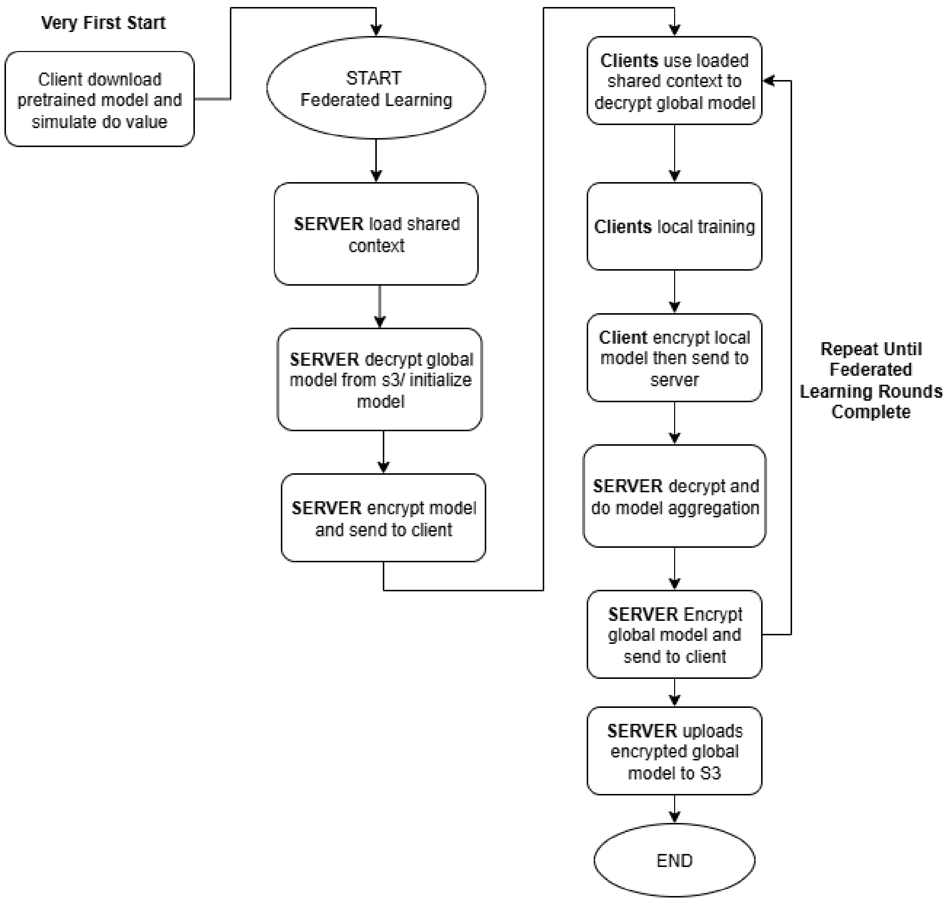


Figure 4.1.1.1 Federated Learning System Flowchart

The figure 4.1.1.1 shows that the flow of the federated learning which starts from server which load shared context to decrypt global model from s3 then load the model. After that, it encrypts it and send to clients. Client will use the same context to decrypt then perform local train. Then, client encrypt back and send to server. Server then decrypt and perform aggregation. This roundtrip continues until the federated learning rounds complete.

4.1.2 System Architecture Diagram

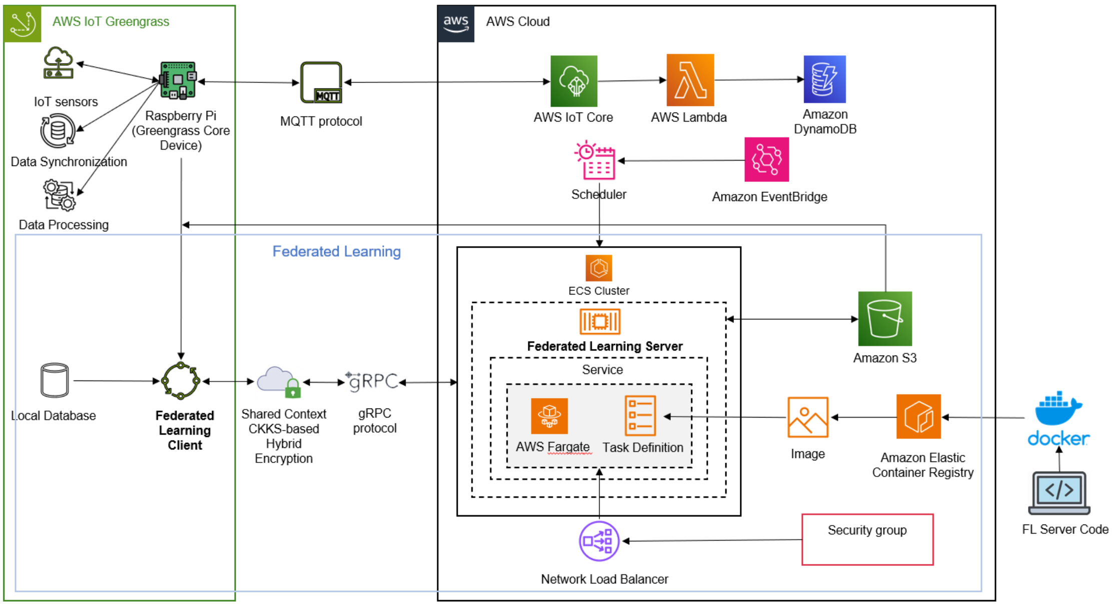


Figure 4.1.2.1: System Architecture Diagram

In this project, it implements a secure federated learning system across edge devices (AWS IoT Greengrass) which acts as clients and an AWS-hosted Flower server (AWS Cloud) which acts as a server that could be accessible by every client. The design ensures that raw sensor data remains strictly on the edge, while only model parameters are exchanged during training. To preserve privacy, all model updates are protected end to end using a implemented CKKS-based hybrid encryption scheme.

Besides, this project also implements a data synchronization pipeline which can real-time synchronize the edge sensor data to the cloud for backup and future monitoring purposes. When network is unstable, the data will be stored in local database in the edge as temporary. When network is stable, it synchronizes the stored data in local database to the DynamoDB via MQTT topic and message routing in AWS IoT Core, Lambda to process data and put into DynamoDB.

To successfully implement a server-side federated learning, edges environment should be setup properly. The Raspberry Pi will be registered as a Greengrass core device. Sensor readings are stored locally in a SQLite database and optionally synchronized to the cloud when connectivity returns. The FL client connects to the server address (DNS address on port 8080) over Flower's gRPC channel while control and notifications can be sent over MQTT via AWS IoT Core. When a client starts, it accesss and load the shared CKKS encryption context from Amazon S3, initializes the same model architecture as the server, and attempts to load an initial model, either from the encrypted global model in S3 or from a local snapshot.

For the server, to setup the ECS properly, there will have several things to setup. First, the flower server code is containerized by Docker and push to Amazon Elastic Container Registry as an image. The service will run behind a Network Load Balancer, and the DNS address will be used by clients as the server endpoint on port 8080. Overview, the server initializes the shared CKKS context and loads the encrypted global model from S3 if it is available before orchestrating training rounds using a custom FedProx strategy and configuration on the environment variables. Optionally, AWS EventBridge Scheduler is used to schedule the federated learning, and the server can publish start signals via AWS IoT Core MQTT.

To start the federated learning, the federated learning server will first configure the FedProx hyperparameters then initiate the communication by call the client via the MQTT protocol or client join manually. Then, the server will encrypt the current global model using the shared CKKS context or randomly initialize and encrypt a model if no encrypted global model before sending it to clients over gRPC. The clients decrypt the parameters with the same context, update their local models, and perform training on their private datasets while applying FedProx regularization. After local training, the updated parameters are re-encrypted with the shared context and sent back to the server. When the encrypted global parameters sent back to the server, it will decrypt these updates, perform a weighted aggregation using FedAvg logic under FedProx, updates the global model in memory, and then re-encrypts the aggregated parameters for distribution in the next round. Clients also report validation metrics, which are aggregated and logged by the server.

At the completion of training, the server encrypts the final global model and stores it in Amazon S3 for future use. It is safe to upload because only the clients who have the shared context able to access the model. For clients, they do not upload their local models but they download the final encrypted global model from S3 for inference or continued training in the next federated learning. The security of model relies on a shared CKKS context stored in S3, ensuring that parameters remain encrypted during transmission, while it still allows the server to decrypt for aggregation and re-encrypt for redistribution. This design enables privacy-preserving federated learning across distributed ponds while maintaining edge-only data retention, encrypted parameter exchange, scalable orchestration through Fargate of ECS, and secure model storage in S3.

Other than that, this architecture diagram also detailly illustrates **how the federated learning server is deployed on AWS** to support federated learning task using a fully managed and serverless approach. Firstly, the FL server code is first containerized using Docker and then pushed to Amazon Elastic Container Registry (ECR) which serves as a private repository for the container images. When a client request arrives, it passes through the Network Load Balancer which provides a single DNS endpoint and distributes traffic across the ECS tasks while enforcing inbound access rules defined in the Security Group. The ECS Cluster hosts a Service configured to run on AWS Fargate. Then, the Task Definition defines the container settings, including the image from ECR, CPU and memory requirements, networking details, and exposed ports. Once deployed, ECS Fargate automatically provisions the containerized FL server inside the VPC, enabling it to handle client requests, orchestrate training rounds, and communicate with other AWS Cloud services.

4.2 System Components Specification

4.2.1 Hardware Specification

Description	**Specifications**
Device Name	LAPTOP-RURK4DU9
System Model	ROG Strix G614JJ_G614JJ
Operating System	Windows 11 (64-bit)
Processor (CPU)	13th Gen Intel(R) i7-13650HX
Graphic Processing Unit (GPU)	NVIDIA GeForce RTX 3050 6GB Laptop GPU
Installed RAM	16.0 GB

Table 4.2.1.1 Specifications of Laptop

Description	Specifications/Functionalities
Raspberry Pi 5	64-bit quad-core Cortex-A76 processor, 8GB LPDDR4X SDRAM, Linux, 800 MHz VideoCore VII GPU
SN-DS18B20 Temperature Sensor	Measures water temperature
GVT-APH-KITV2 pH Level Sensor	Measures the pH level of water in aquaculture environment.
TDS Salinity Sensor	Measure salinity
MQ-137 Ammonia	Measure ammonia level in the aquaculture environment
Dorhea OV5647 Sensor (1080p HD Webcam)	Used for capturing images of prawns for prawn classification and growth stage monitoring.

Table 4.2.1.2 Hardware in This Project

4.2.2 Software and Library Specification

Software/Library	Software/Library Version
Raspberry Pi OS	Debian Bullseye 64-bit
Python Libraries	boto3, sqlite3, flower, joblib, pandas, torch, numpy, scikit-learn, tenseal, logging, typing
Cloud Computing Platform	AWS Cloud
Coding Platform	Visual Studio Code
Containerization Platform	Docker

Table 4.2.2.1 List of Software and Library Used in This Project

Library	Description
boto3	Official Python SDK for AWS services
Flwr (Flower)	Python framework for federated learning
scikit-learn	Python library for machine learning and data preprocessing
joblib	Python tool for model and data serialization
pandas	Python library for data analysis and manipulation
Torch (PyTorch)	PyTorch is a Python-based deep learning library
numpy	Fundamental Python package for numerical computing
tenseal	Python library for encrypted deep learning using CKKS encryption

Table 4.2.2.2 Library Function in This Project

To explain, boto3 will assist in integration with AWS S3 to upload and download model files and CKKS shared context; flwr will be used as the framework for project's federated learning and its strategy will be implemented; scikit-learn is used to local process and train data; sqlite3 supports database; Flower is python framework for federated learning; pandas for handling selection for feature and target variables; numpy is used for preparation of features and target arrays and calculations like mean absolute error; torch is used for managing machine learning model including training and model parameter manipulation; tenseal is used to achieve CKKS hybrid encryption.

4.2.3 AWS Cloud Service Specification

AWS Cloud Service	Specification
AWS S3	Store the encrypted global model and CKKS shared context
Amazon ECR	Stores containerized FL server image
Amazon ECS (Fargate)	Runs serverless containers with auto-scaling
Amazon VPC	Provides secure, isolated networking for ECS
Application Load Balancer	Balances client traffic to ECS tasks
AWS IoT Core (MQTT)	Act as a intermediary to store data published by the edge for data synchronization
AWS DynamoDB	Stores the data synchronized from the edges
AWS Lambda	Route MQTT received subscribed message to DynamoDB
AWS EventBridge Scheduler	Schedule the Federated Learning

Table 4.2.3.1 AWS Cloud Service Specification

4.3 Core Algorithms and Equations

4.3.1 FedProx Algorithm

Federated Proximal (FedProx) is the key algorithm used in this federated learning system, addressing both local and server components. It is designed to better handle heterogeneity among client data. On the client side, FedProx alters the local optimization target by adding a proximal term that penalizes divergence from the global model [10]. This proximal term acts as a regularization component in the local model to ensure it stays close to the global model.

Below is the core formula of FedProx for local model [10]:

$$\min_{w} \{h_k(w; w^t) = F_k(w) + \frac{\mu}{2}\|w - w^t\|^2\}$$

where

$F_k(w)$: client k's local loss function

w: the local model parameters being optimized on client k

w^t: global parameters of server of epoch t

$h_k(w; w^t)$: the objective function to minimize

$\frac{\mu}{2}\|w - w^t\|^2$: the proximal term

μ: proximal term's regularization parameter

In the system implementation, $F_k(w)$ is defined as the confidence-weighted mean square error (MSE) across batches:

$$F_k(w) \approx mean_{batch}\big(conf \cdot MSE(F_W(x), y)\big)$$

The proximal term can be minimized to force the parameters of local model staying close to the global model parameters. When the proximal term is 0, the system will be the same as FedAvg. Hence, it is an improvement to the FedAvg method.

Then, for the server part, FedProx uses strategy which has the same logic as Federated Averaging (FedAvg) algorithm to aggregate these updates using a weighted average to produce a new global model that involves all contributions of clients. This cycle will repeat for multiple rounds to gradually improve the model.

Below is the core formula of aggregation:

$$w_{t+1} = \sum_{k=1}^{K} \frac{n_k}{n} w_{(t+1)}^{k}$$

where

w_{t+1}: Updated global model weighs after round t+1,

K: Total number of participating clients in the round

n_k: Count of data samples/weights for client k. It represents the total count of samples handled throughout all batches and epochs in the round

$\sum_{k=1}^{K} n_k$: Sum of training samples across all clients

$w_{(t+1)}^{k}$: The revised model weights from client k following local training in round t

In the system design, this formula will ensure that clients with more data to influence the global model more significantly.

4.3.2 Shared Context CKKS-based hybrid encryption/decryption

The utilized scheme for encryption and decryption is Cheon-Kim-Kim-Song (CKKS) via TenSEAL. To explain, a single shared CKKS context is generated and loaded by both server and clients from AWS S3. It includes secret key material.

For the parameter flow during federated learning, the values are scaled before encryption and unscaled after decryption. Each encrypted tensor is serialized with **shape, context fingerprint, and a quantized hash** for integrity.

For aggregation, server decrypts each client's result, computes a weighted average using number of examples then re-encrypts for distribution.

4.3.2.1 CKKS Context Lifecycle (Server and Clients)

When there is no shared context in the S3, a shared CKKS context is generated with parameters below and uploaded to S3. After that, On the future runs, all parties will load the same context. Then, a SHA-256 fingerprint is used to ensure the consistency. When the federated learning start, server and clients will use the context to create their handler for encryption and decryption. For the Context Parameters, it contains polynomial modulus degree with the degree as 8192. This determines the security level and computational capacity of the encryption scheme [22]. Then, the context also contains coefficient modulus bit sizes with the shape of [60, 40, 40, 60]. They define the precision levels available during encryption operations [22]. Last, the context also contains 2^{40} as the global scale. It controls the precision of real number encoding in CKKS. $2^{40} \approx 1.1 \times 10^{12}$ provides sufficient precision for neural network parameters.

4.3.2.2 Parameter Packaging, Scaling, and Integrity

Before encryption, parameters are scaled by $a = 10^3$ and clipped to $[-10^{-6}, 10^6]$. After decryption, they are unscaled by dividing by a. A quantized hash with rounded to 3 decimals is stored to verify decryption integrity tolerant to CKKS noise. The equations are shown below:

Scaling:

$$\tilde{\theta} = a \cdot \theta$$

Unscaling:

$$\theta = \frac{\tilde{\theta}}{a}$$

Hash input:

$$q(x) = round(x, 3)$$

4.3.2.3 Secure Aggregation on Server

The server collects each client's encrypted parameters and the number of examples. It decrypts each set, computes a normalized-weight average per parameter, then re-encrypts the aggregated result for broadcast. For aggregation, server take each client's parameters and average them but give more influence on clients that processed more samples in the round.

For the aggregation equation (FedFrox with per-round weights):

$$\theta_{agg} = \sum_{i=1}^{N} \left(\frac{n_i}{\sum_{j=1}^{N} n_j} \right) \theta_i$$

where

θ_{agg} is aggregated global model parameters

N is number of clients

θ_i are the model parameters that have been decrypted from client i following its local training step

n_i is total weights/ number of samples the client processed across all batches and epochs in that round

$\sum_{j=1}^{N} n_j$ is the total weight across all participating clients

To explain more on this, on the client, number of samples is computed as the aggregate of training samples handled across every batch and all local epochs within the round (summed by batch size). So, if client's dataset has 1000 rows and local epoch is 3, num_examples will be 3000. Hence, the aggregation weights reflect samples processed across all epochs and not just unique dataset size.

4.3.3 Evaluation Metrics

4.3.3.1 Mean Absolute Error

This measure mean absolute variation between forecasted and real dissolved oxygen measurements [23]. It is calculated based on formula in [23]:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2$$

where

y_i : Actual dissolved oxygen value

$\hat{y_i}$: The model's predicted value of dissolved oxygen.

n : Number of samples

4.3.3.2 Mean Square Error

This calculates the average of the squared differences between the anticipated values and the actual values [23]. In this implementation, the training loss function and the evaluation metric for assessing model performance by using MSE. It is is calculated based on formula in [23]:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2$$

where

y_i : Real dissolved oxygen value

$\hat{y_i}$: The anticipated value of dissolved oxygen by the model

n : Number of samples

To compute MSE, in the training phase, MSE is combined with a FedProx regularization term to form the total loss function. However, during evaluation, the pure MSE is computed to assess how accurately the model predicts dissolved oxygen levels. A lower MSE indicates better generalization and predictive performance.

4.3.3.3 Coefficient of Determination (R^2)

R^2 measure the accuracy of target prediction [23]. For example: If $R^2 = 0.85$, the model accounts for 85% of the variation in dissolved oxygen levels, signifying effective predictive capability for monitoring water quality. Hence, R^2 measure the accuracy of target prediction [23]. It is calculated based on formula from [25]:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

where

$$SS_{res} = \sum_{i=1}^{n} (y_i - \hat{y_i})^2$$

$$SS_{tot} = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

$$\bar{y} = \sum_{i=1}^{n} (y_i)^2$$

where

SS_res = Total squared difference between predictions and actual values.

SS_tot = Total squared difference between actual values and their mean interpretation.

$\bar{y}$ = Mean of actual values

y_i = Observed value for the iii-th data point

According to [23], the greater the R^2, the stronger the predictions

4.3.4 Deep Learning Predictive Model Architecture

4.3.4.1 Neural Network Design

The system employs a feedforward neural network that is specifically built to forecast dissolved oxygen levels in aquaculture conditions. It was constructed in three fully interconnected layers, with every layer including rectified linear unit activation functions [24] and has been refined to work efficiently with multiparametric water quality data collected by a variety of sensors.

Network Architecture:

- Input Layer: 5 features (water quality parameters)
- 16 neurons in the first hidden layer utilizing ReLU activation
- 8 neurons in the second hidden layer utilizing ReLU activation
- Output Layer: 1 neuron for dissolved oxygen prediction.

The architecture (5→16→8→1) enables the network to identify non-linear connections between input features, extract relevant patterns from heterogeneous sensor data and maintain computational efficiency for edge deployment on IoT devices.

4.3.4.2 Input Features Process

The model processes five water quality parameters that directly influence dissolved oxygen levels. The five parameters are temperature, ph level, salinity, ammonia and turbidity. These features are standardized using RobustScaler to handle outliers common in sensor data: **X_scaled = (X - median(X)) / IQR(X)** where IQR represents the interquartile range, providing robustness against sensor anomalies.

4.3.4.3 Loss Function and Optimization

The model employs a confidence-weighted Mean Squared Error (MSE) loss function, integrated with FedProx regularization:

$$L(w) = \frac{1}{n}\sum_{i=1}^{n}[c_i \cdot (y_i - \hat{y_i})^2] + (\frac{\mu}{2})\|w - w^t\|^2$$

where

c_i: Confidence weight for sample i (derived from pseudo-labeling uncertainty)

y_i: True dissolved oxygen value

$\hat{y}_i$: Predicted dissolved oxygen value

μ: Proximal regularization parameter

w^t: Global model parameters from server

4.4 Modules of the System

The proposed system is composed of several functional modules and each of the module is responsible for a specific role in the federated learning process.

4.4.1 Data Collection and Preprocessing Module

This module is responsible collecting parameters from IoT sensors which includes temperature, pH, turbidity, salinity, and ammonia levels. The Raspberry Pi device reads and stores this data locally in an SQLite database. At the same time, data preprocessing is also performed at this stage, including scaling, handling missing values, and formatting the dataset to ensure it is ready for local training.

Function:

1. **Load_local_features_from_sqlite**. This function synthesizes and loads sensor rows then extracts 5 features into a NumPy array
2. **generate_pseudo_labels_from_local**. This function runs a pretrained model on locally scaled features to produce pseudo-labels and confidence weights.
3. **build_dataloaders**. This function cleans features, standardize X and y then split into 80%:10%:10. Then, create dataloader with shape (X, y, confidence)
4. **evaluate_loader**. This function runs a model on a dataloader and returns average loss, MAE, MSE and sample count

4.4.2 Synchronization Module

The synchronization module manages the communication between edge devices and the cloud, especially under intermittent network conditions. When the network is unavailable, data is stored in the local SQLite database. Once the connection is restored or become stable, the synchronization module uploads the data to AWS DynamoDB through MQTT messages routed by AWS IoT Core and AWS Lambda. This ensures data consistency and reliability across the system.

4.4.2.1 AWS IoT MQTT Test Client

The AWS IoT MQTT Test Client is a web application with which developers can test MQTT messaging with AWS IoT Core using any computer with Internet access and with no actual devices and complicated configuration [19].

In the system design, it acts as an intermediary between edge and cloud to direct the sensor data to cloud. For example, edge devices publish the sensor data to the designated topic whereas the cloud subscribe the topic will receive the data payload. Then, IoT Core Message Routing rule Routes messages based on topic subscriptions, Processes messages using SQL queries and triggers the Lambda function to process the data.

4.4.2.2 AWS Lambda

AWS Lambda is a computing service that runs code based on events with no server administration required [19].

In the system design, Lambda assists in putting items from IoT topics to DynamoDB during synchronization. The Lambda function receives the event in which it is triggered by IoT Core when a message is published to a subscribed topic. Then, it converts IoT message format to DynamoDB-compatible attribute format. To explain, it maps JSON values to appropriate DynamoDB types. Then, it put data onto the edge table in the dynamodb.

4.4.2.3 AWS DynamoDB

Amazon DynamoDB is a completely managed NoSQL database service offered by AWS [19].

In the system design, DynamoDB serves as the immediate storage for IoT sensor data which will be used for monitoring purposes.

4.4.3 Federated Learning Client Module

Each edge device runs a federated learning client that participates in collaborative training. The client downloads the latest encrypted global model from AWS S3, decrypts them using the shared CKKS context, and performs local training with the preprocessed sensor data. Once training is complete, the parameters of the revised model are re-encrypted. Subsequently, they are sent to the server for aggregation.

Function:

1. **FLClient.__init__** This function sets model, initializes hybrid encryption, loads encrypted global model if available and builds dataloader
2. **FLClient.get_parameters**. This function exports model weights as Numpy then encrypts and returns to server
3. **FLClient.set_parameters**. This function decrypts global parameters and loads into the local model
4. **FLClient.fit**. This function trains with confidence-weighted MSE + FedProx proximal term; reports train/val metrics
5. **FLClient.evaluate**. This function evaluates on test split then report loss, MAE, MSE, and R^2

To support the Federated Learning Client module, there are some **fundamental components listed below** in the Federated Learning Client to work closely.

4.4.3.1 AWS IoT Greengrass

AWS IoT Greengrass takes AWS services to edge nodes so that data generated by them can be acted upon locally while remaining cloud-based for management and analytical tasks and storage in the long run [19].

In the implemented system, AWS IoT Greengrass runs on edge devices such as Raspberry Pi and makes them federated learning clients. This facilitates local data training and processing with models by running the code for federated learning clients locally at the device level. Greengrass itself handles local database and synchronization logic to keep data intact during offline and synchronize when it is reconnecting.

4.4.3.2 AWS IoT Core

AWS IoT Core offers secure and bidirectional IoT device and AWS cloud communications [19]. Due to its synchronization feature, it is suitable for areas with occasional connectivity.

This is achieved by this federated learning system deployment using AWS IoT Core to enable communications with cloud server and Greengrass devices. The IoT Core MQTT messages are acted upon by the federated learning server to instruct edge clients and Greengrass-enabled devices to train or to make updates.

4.4.3.3 Raspberry Pi

In this configuration, the Raspberry Pi serves as the local host for the Greengrass core software and the federated learning client in the system design. The Raspberry Pi captures raw environment data from IoT sensors it is wired to such as temperature, turbidity, pH and so on. The information is stored locally and is used to train local machine learning models. By running locally, the Raspberry Pi is used to preserve data privacy and ensure learning can occur regardless of if there is any connection to the cloud and only synchronize with the latter when it is necessary to do so.

4.4.4 Federated Learning Server Module

The server module which is hosted on AWS ECS with Fargate, aggregates model parameters received from clients using the FedProx algorithm. After aggregation is complete, the updated global model is encrypted and stored in AWS S3.

Function:

1. **get_model_parameters**. This function serializes server model state to NumPy arrays.
2. **set_model_parameters**. This function loads NumPy arrays into the server model state.
3. **bootstrap_global_model_from_s3**. This function downloads, decrypts, and loads the last encrypted global model from S3.
4. **CustomFedProxStrategy.__init__**. This function configures FedProx (fractions, min clients, μ), initializes encryption, tracking, and metric history.

5. **CustomFedProxStrategy.configure_fit**. This function sets per-round hyperparameters and encrypt sent global parameters
6. **CustomFedProxStrategy.aggregate_fit**. This function securely aggregates encrypted client updates (decrypt→weighted average→re-encrypt), updates model, logs and records validation metrics.
7. **CustomFedProxStrategy.aggregate_evaluate**. This function aggregates test metrics across clients and logs and records them.
8. **CustomFedProxStrategy.configure_evaluate**. This function encrypts parameters and sets validation config for clients.

To support the Federated Learning Server module, there are some fundamental Components in the Federated Learning Server to work closely.

4.4.4.1 Amazon Elastic Container Registry

Amazon ECR containerize FL server code using Docker to create a reliable and portable deployment package. The containerized code is then pushed to Amazon Elastic Container Registry (ECR). Pushing to ECR will ensure secure image storage with encryption at rest and motion and it performs automatic vulnerability scans to identify security vulnerabilities..

4.4.4.2 Network Load Balancer (NLB)

In system design, the Network Load Balancer (NLB) serves as the entry point for client requests. It provides a single DNS endpoint for federated learning clients. Furthermore, health checking is also provided to ensure traffic only flows to healthy instances. Lastly, there is no charge for unused capacity.

4.4.4.3 Amazon ECS

In the system design, ECS Cluster is built to run scaling, and securing Federated Learning Server. There are also two components required for the cluster [19].

First, the **Task Definition** is essentially a blueprint to specify the system's container configurations. This includes container image of the system server code, location (ECR), CPU and resource allocation for memory, network configuration and port mappings, environment variables, security configuration, and integration points with other AWS services being offered.

Then, the **Service** keeps a certain number of FL server instances running the task definition at all times. The Service automatically replaces failed tasks, supports rolling deployment for zero-downtime updates, and collaborates with the load balancer to manage traffic efficiently.

4.4.4.4 Amazon Fargate

AWS Fargate is a serverless container engine for compute that takes away provisioning and server management [19].

At deployment for the federated learning system, the ECS cluster executes the AWS Fargate-running Docker container as server. The container is started from Amazon ECR and has execution logic with federated learning coordination management, aggregation of models at edge locations, encryption management, and invocation by clients. This architecture offers a fault-tolerant and scalable server infrastructure to manage distributed edge devices.

4.4.4.5 Security Group

Security Groups act as virtual firewalls managing incoming and outgoing traffic for AWS resources. They are filters at instance-level and are stateful filters so allow return traffic through automatically based on inbound filters. In this federated learning, it impose network access controls during all communications.

4.4.4.6 Amazon S3

Amazon S3 is an object storage service with scale for data durability, availability, and performance [19].

In the system design, it stores the encrypted global model, shared context for encrypting and decrypting. For example, the clients retrieve shared context and encrypted global model from S3. The server puts up the final encrypted global model there too.

4.4.5 Encryption Module

The encryption module implements CKKS-based hybrid encryption to provide privacy-preserving communication of model parameters. It manages context generation, encryption of local updates before transmission, decryption of global models, and re-encryption after aggregation.

Flow:

1. Generate and manage CKKS context and keys.
2. Encrypt the local model before transmission to the server.
3. Decrypt the aggregated global model received from the server.
4. Re-encrypt the global model before redistribution to clients.
5. Verify data integrity using quantized hashing.

Function:

1. **HybridEncryptionHandler.__init__**. This function loads or creates a shared CKKS context (with secret) from S3 and marks ready.
2. **HybridEncryptionHandler.initialize_context**. This function creates CKKS context (params, global scale, Galois keys).
3. **HybridEncryptionHandler._generate_shared_context**. This function
4. **HybridEncryptionHandler._hash_quantized**. This function produces a quantized SHA-256 hash for integrity tolerant to CKKS noise.
5. **HybridEncryptionHandler._scale_parameters**. This function prescaling to improve CKKS precision and stability.
6. **HybridEncryptionHandler._unscale_parameters**. This function postscaling to improve CKKS precision/stability.
7. **HybridEncryptionHandler.encrypt_parameters**. This function converts Flower Parameters to arrays, scales, CKKS-encrypts them then packs with shape, hash and fingerprint.
8. **HybridEncryptionHandler.decrypt_parameters**. This function validates fingerprint, decrypts and unpacks arrays, perform integrity-checks via plaintext hash then unscales to return Parameters.
9. **HybridEncryptionHandler.aggregate_encrypted_parameters**. This function serves as a assistance for server to decrypt, aggregate and re-encrypt the aggregated result

10. **HybridEncryptionHandler.get_decrypted_parameters_for_pytorch**. This fucntion
11. **HybridEncryptionHandler.verify_encryption_roundtrip**. This function perform encrypt to decrypt check with tolerance then returns pass or fail.
12. **create_server_hybrid_handler**. This function is a server helper function to construct and initialize handlers.
13. **create_client_hybrid_handler**. This function is a client helper function to construct and initialize handlers.

4.4.6 Evaluation Module

The evaluation component evaluates the effectiveness of the global model following every training cycle. It computes three regression metrics for evaluation. First, mean absolute error is computed. Second is mean squared error. Third, coefficient of determination to measure predictive accuracy for dissolved oxygen levels.

CHAPTER 5

System Implementation

5.1 Hardware Setup

5.1.1 Raspberry Pi Setup at Farm

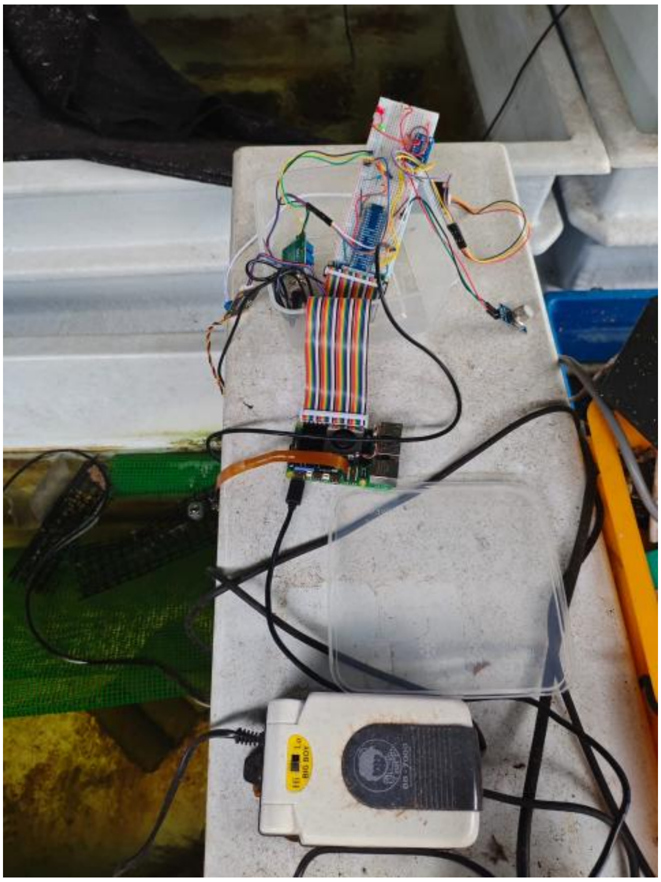


Figure 5.1.1.1 Raspberry Pi Set Up at Farm

The raspberry pi was setup at the pond with IoT sensors to collect sensor data

5.1.2 Local storage (SQLite on Pi)

```
Edge_1|2025-09-20 01:09:19|29.562|21.96|1.019|0.03|74.1|
Edge_1|2025-09-20 01:09:31|29.562|21.96|1.003|0.03|74.1|
Edge_1|2025-09-20 01:09:42|29.562|21.96|0.988|0.48|74|
Edge_1|2025-09-20 01:09:53|29.562|21.96|0.974|0.5|74.2|
Edge_1|2025-09-20 01:10:04|29.562|21.96|0.961|0.51|74.2|
Edge_1|2025-09-20 01:14:29|29.687|21.96|1.006|0.43|73.3|
Edge_1|2025-09-20 01:14:40|29.687|21.96|1|0.43|73.8|
Edge_1|2025-09-20 01:14:52|29.75|21.96|0.986|0.41|0|
Edge_1|2025-09-20 01:15:03|29.687|21.96|0.969|0.41|74.7|
Edge_1|2025-09-20 01:15:14|29.687|21.96|0.948|0.4|74.2|
Edge_1|2025-09-20 01:15:25|29.687|21.96|1.302|0.4|74.4|
Edge_1|2025-09-20 01:15:36|29.687|21.96|1.034|0.4|74.2|
Edge_1|2025-09-20 01:15:48|29.687|21.96|1.025|0.4|73.4|
Edge_1|2025-09-20 01:15:59|29.687|21.96|0.946|0.4|73.6|
Edge_1|2025-09-20 01:16:10|29.687|21.96|0.912|0.41|73.3|
Edge_1|2025-09-20 01:16:21|29.687|21.96|0.888|0.41|73.2|
Edge_1|2025-09-20 01:16:33|29.687|21.96|0.866|0.41|73.3|
Edge_1|2025-09-20 01:16:44|29.687|21.96|0.861|0.41|73.1|
Edge_1|2025-09-20 01:16:55|29.687|21.96|0.854|0.41|73.1|
Edge_1|2025-09-20 01:17:06|29.687|21.96|0.846|0.41|73.2|
```

Figure 5.1.2.1: Snapshot of Data Stored in the Sqlite Database in Raspberry Pi

5.1.3 Laptop used for development

A development laptop was used to write the federated learning server and client code, build Docker images, and manage AWS resources. The laptop was connected to AWS through the Visual Studio Code IDE with AWS Toolkit, enabling direct deployment of container images to Amazon ECR and configuration of ECS services. Additionally, the laptop served as the initial testing environment before deploying the federated learning server to AWS Fargate and the clients to Raspberry Pi devices.

5.1.4 Network setup to connect devices to AWS

```
if __name__ == "__main__":
    server_addr = os.getenv("FL_SERVER_ADDRESS", "flwr-nlb-a6831504fd939e64.elb.ap-southeast-1.amazonaws.com:8080")
    fl.client.start_numpy_client(server_address=server_addr, client=FLClient())
```

Figure 5.1.4.1: DNS Address Attached on Client Script

The Raspberry Pi devices were connected to the AWS cloud through Wi-Fi, enabling communication with cloud services. The federated learning clients accessed the server via a Network Load Balancer (NLB) that provided a single DNS endpoint on port 8080 for gRPC communication.

5.2 Software Setup

5.2.1 Environment Preparation for Implementation

The environment preparation was carried out in both the local development machine and the AWS cloud platform to ensure that the system could be deployed and executed successfully.

5.2.1.1 Local Development Environment

The laptop was configured with Python and the required machine learning libraries, including PyTorch, Flower, TenSEAL, boto3, pandas, and NumPy. These libraries enabled the development of federated learning algorithms, secure encryption mechanisms, and communication with AWS cloud services. Docker Desktop was also installed to containerize the federated learning server, while Visual Studio Code served as the main coding platform.

5.2.1.2 Docker Image Creation and Push to ECR

flwr_server > Dockerfile

```
FROM public.ecr.aws/amazonlinux/amazonlinux:latest

RUN yum update -y && \
yum install -y python3 python3-pip gcc gcc-c++ make cmake3 python3-devel && \
if [ ! -e /usr/bin/cmake ]; then ln -s /usr/bin/cmake3 /usr/bin/cmake; fi && \
yum clean all

WORKDIR /app

COPY requirements.txt /app/requirements.txt
RUN pip3 install --no-cache-dir -r /app/requirements.txt

COPY . /app
COPY .env /app/.env

CMD ["python3", "flower_server.py"]
```

Figure 5.2.1.2.1 Dockerfile

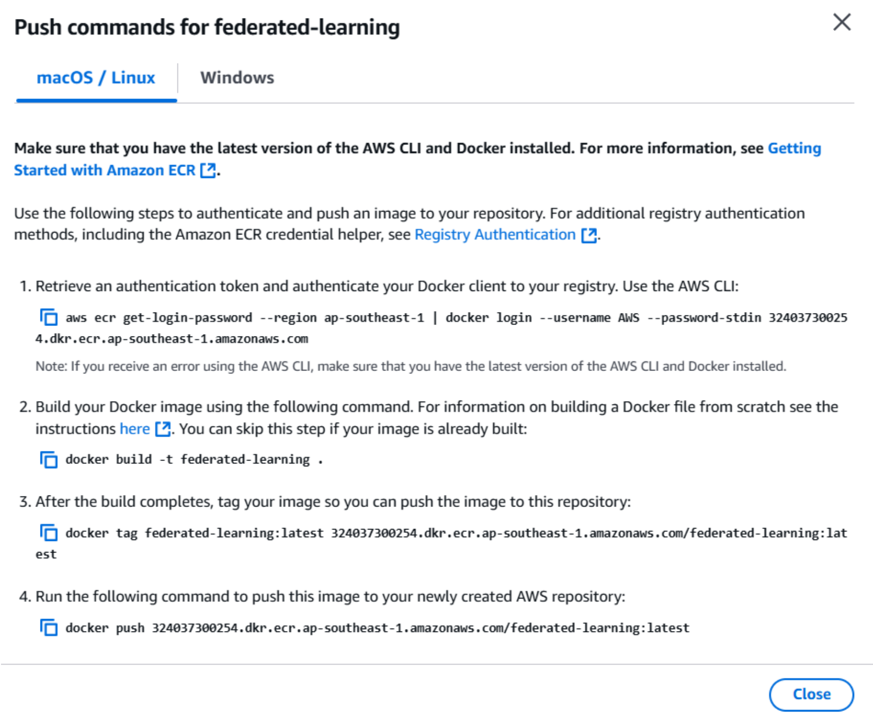


Figure 5.2.1.2.2: Command to Build Image and Push to ECR

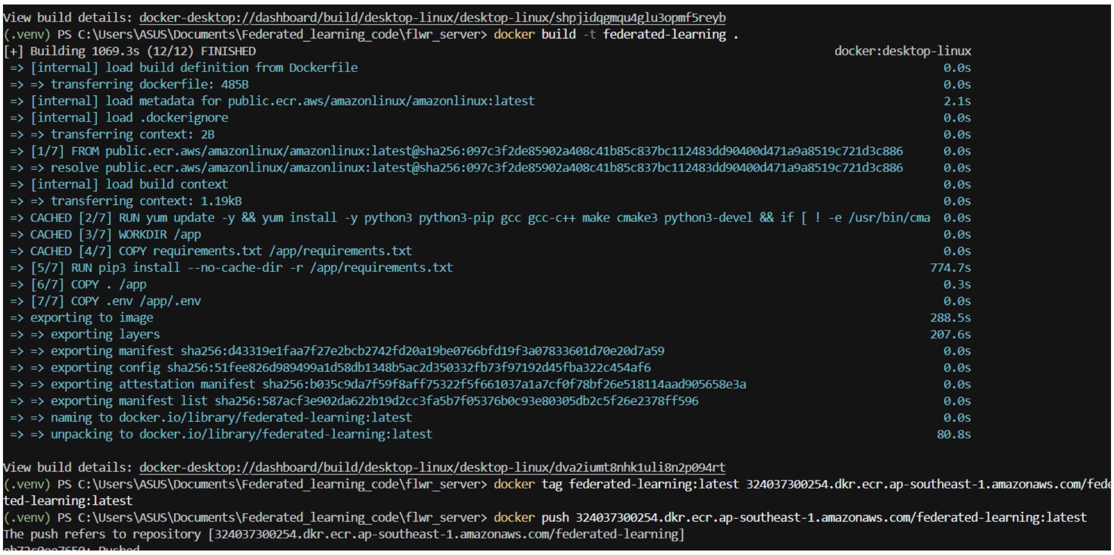


Figure 5.2.1.2.3: Docker Image Build Successfully

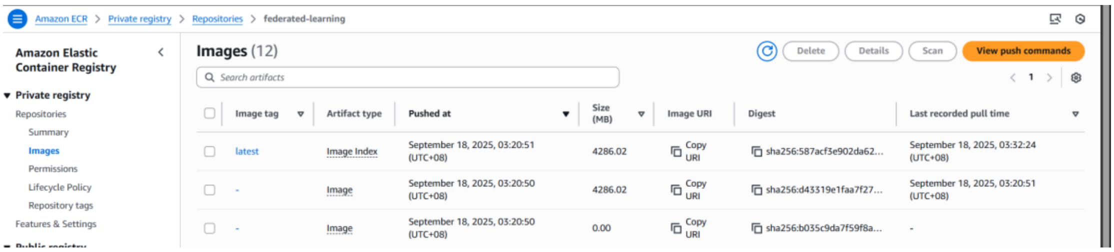


Figure 5.2.1.2.4: Image in the Repository in ECR

The federated learning server code was containerized using Docker. A Dockerfile was created to define the dependencies, runtime environment, and entry point of the server. After building the Docker image locally, it was tagged and pushed to Amazon Elastic Container Registry which acts as a secure, private repository for storing container images. The image stored in ECR was later used by Amazon ECS tasks to launch the server on AWS Fargate.

5.3 Setting and Configuration

5.3.1 Federated Learning Server Setting and Configuration

5.3.1.1 AWS ECR Repository Creation

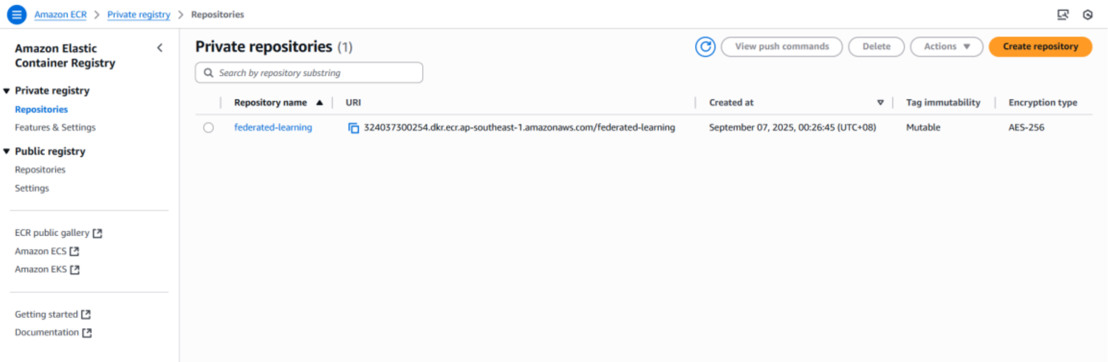


Figure 5.3.1.1.1: Private Repository

The private repository is created to store the image of server code securely

5.3.1.2 VPC Configuration

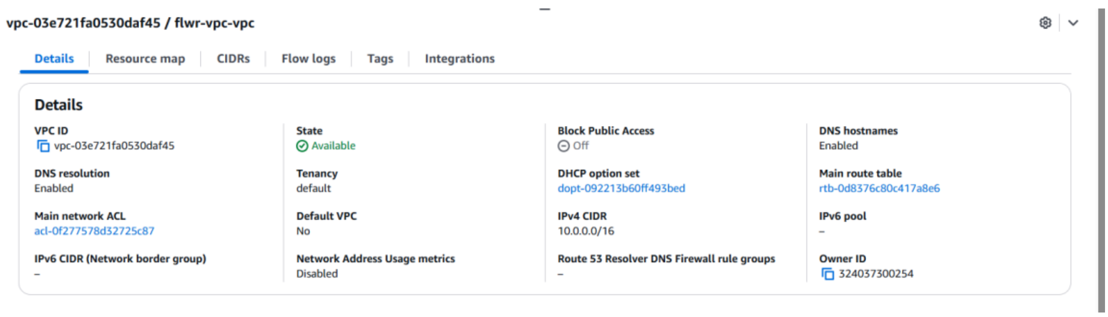


Figure 5.3.1.2.1 VPC Configuration

Configured a new VPC which specifically for the federated learning environment. The vpc id is vpc-03e721fa0530daf45.

5.3.1.3 Security Group

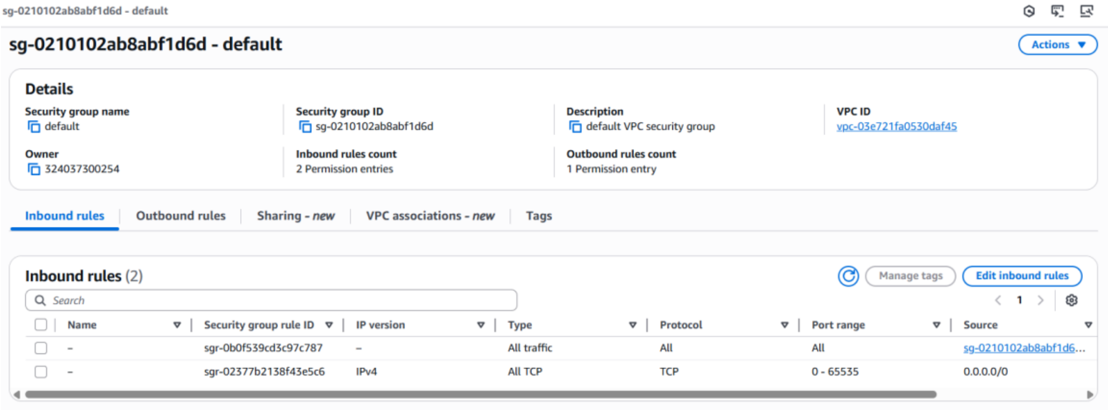


Figure 5.3.1.3.1 Inbound Rule of Security Group

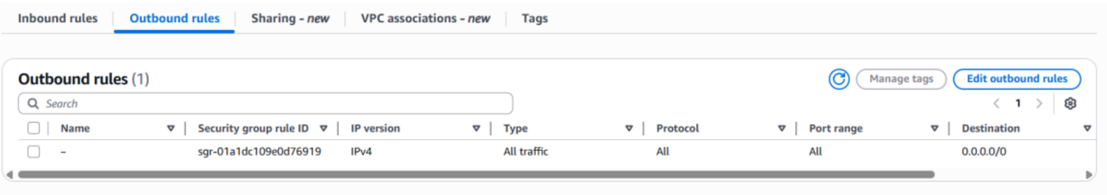


Figure 5.3.1.3.2 Outbound Rule of Security Group

For the security group of vpc-03e721fa0530daf45 (VPC), configure one more the inbound rule with All TCP type, TCP protocol, all TCP ports (0-65535) and 0.0.0.0/0 for source. This is to allows all TCP traffic from any source on the internet. Others just leave as default. The outbound rules allowing all outbound traffic to any destination.

5.3.1.4 Network ACLs

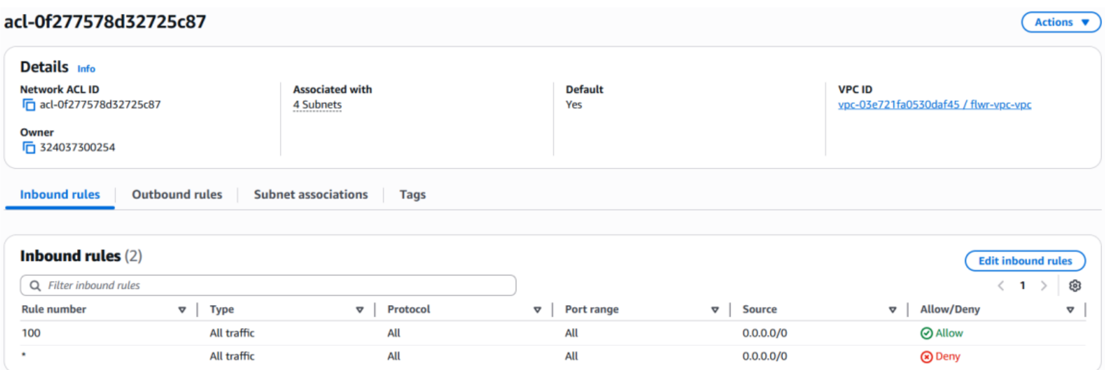


Figure 5.3.1.4.1 Inbound Rules of Network ACLs

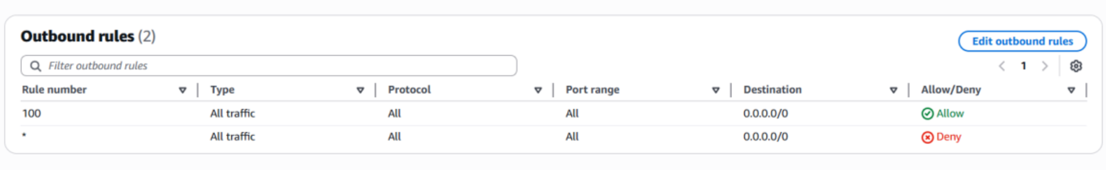


Figure 5.3.1.4.2 Outbound Rules of Network ACLs

For the inbound rule, set the type to all traffic, protocol to all, port range to all, source to 0.0.0.0/0 (from anywhere on the Internet), and configure the outbound rule with identical settings. This is to ensure external federated learning clients can reach NLB and subsequently flower server and flower server can communicate outbound for model updates, logging, or external dependencies.

5.3.1.5 Target Group Configuration

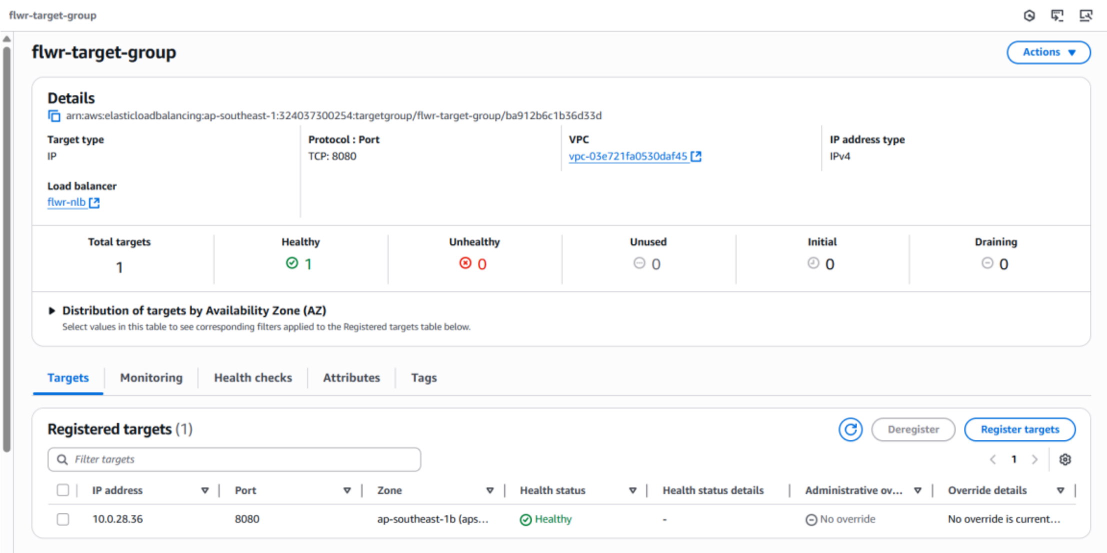


Figure 5.3.1.5.1 flwr-target-group

The target group (flwr-target-group) is configured for the federated learning. The target type chosen is IP addresses to support load balancing to VPC and facilitates routing to multiple IP addresses in federated learning. The protocol and port chosen is TCP: 8080 for target group to allow load balancer to route traffic to the target group. Last, the IP address type is IPv4.

5.3.1.6 Network Load Balancer Configuration

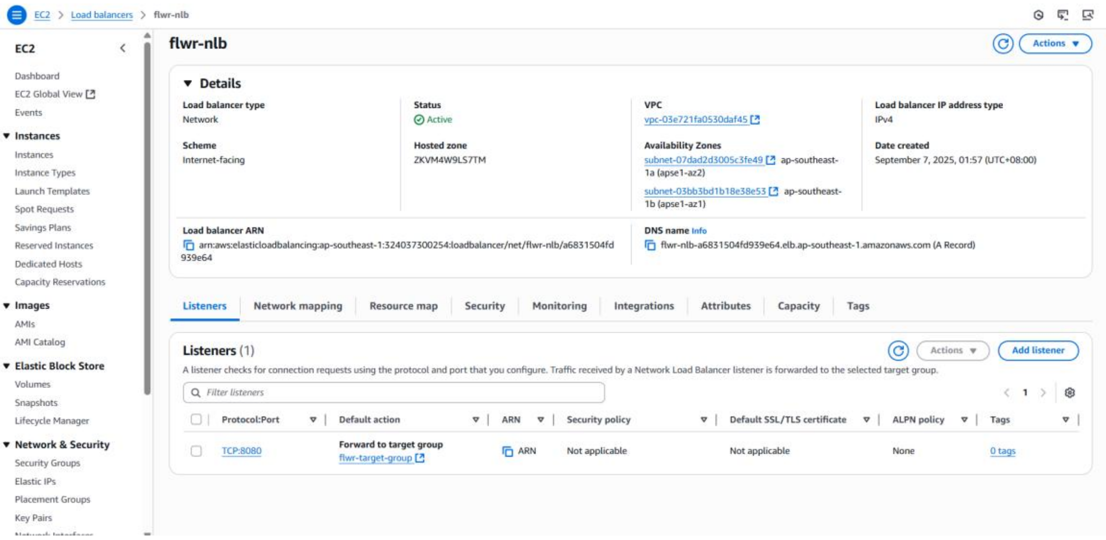


Figure 5.3.1.6.1: NLB Configuration

The Load Balancer (flwr-nlb) is configurated for Flower federated learning framework. For the configuration, the type of load balancer chosen is Network Load Balancer because of its high performance and low latency which is suitable for federated learning workloads that require efficient client-server communication. Then, the Internet-facing scheme is used to allows external federated learning clients to connect from outside the set VPC. The type of IP Address Type is IPv4.

For the network configuration for load balancer, vpc-03e721fa0530daf45 is configured which is a specialized network setting Availability Zones to implement across various Availability Zones for enhanced availability.

For listener configuration, TCP:8080 is configured to forward federated learning traffic through port 8080 to the federated learning server in the target group.

After that, save the generated DNS name which will be configured in client side to provide a stable endpoint for federated learning clients to connect to.

5.3.1.7 ECS Task Definition

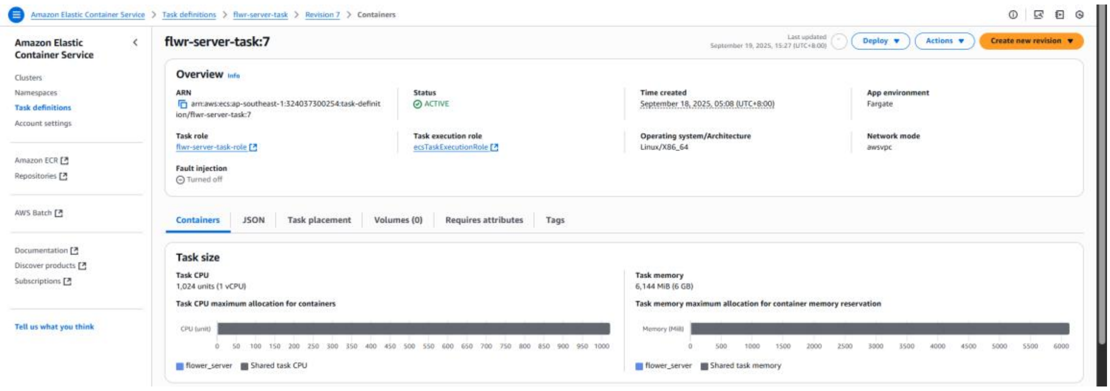


Figure 5.3.1.7.1: ECS Task Definition Configuration

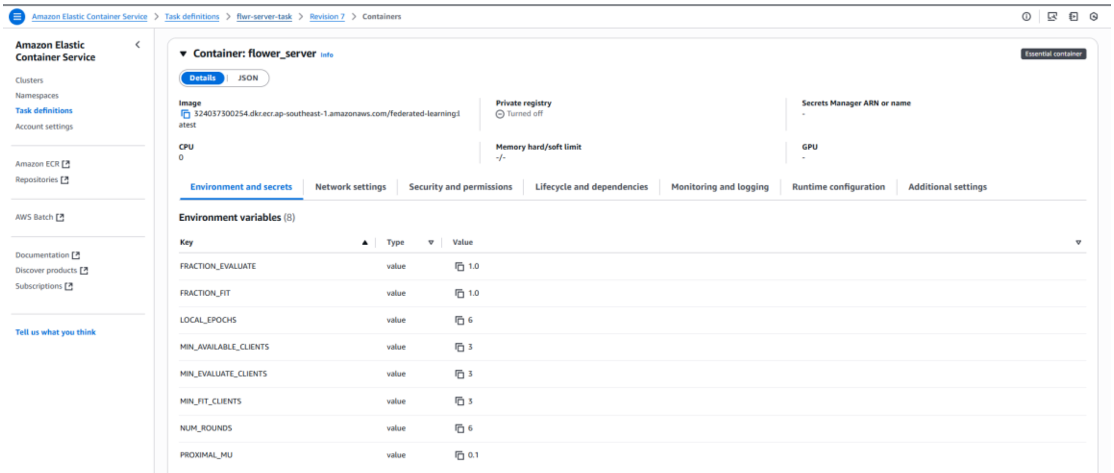


Figure 5.3.1.7.2: ECS Task Definition Environment Variables Configuration

The federated learning server has been launched on AWS Fargate using an ECS task definition called flwr-server-task. It was assigned 1 vCPU (1024 units) and 6 GB of memory, which is enough to have sufficient computational capacity to aggregate the model and communicate with various clients. The network mode was also enabled as awsvpc, which gives the task an independent elastic network interface and a unique IP address to securely connect to the VPC. The task role (flwr-server-task-role) was used in the task definition to grant access to the Amazon S3 to store the encrypted global models and shared context whereas ecstaskexecutionrole was used to allow ECS to fetch the server container image (flower_server) in ECR and control CloudWatch logs.

To modify the configuration of the federated learning server like min_available_clients, num_rounds, local_epochs and so on, admin can create a new task definition revision to modify then modify service to run the latest task. This setup guaranteed the secure, stable, flexible and reliable implementation of the federated learning server in the cloud.

5.3.1.8 ECS Cluster Setup

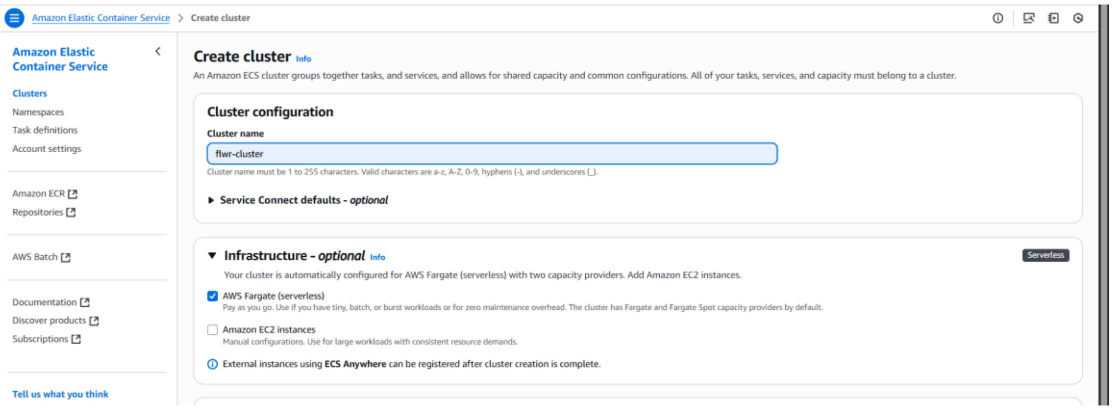


Figure 5.3.1.8.1: ECS Cluster Setup

For ECS, create cluster with name "flwr-cluster" and choose AWS Fargate as Infrastructure.

5.3.1.9 Fargate service running in ECS

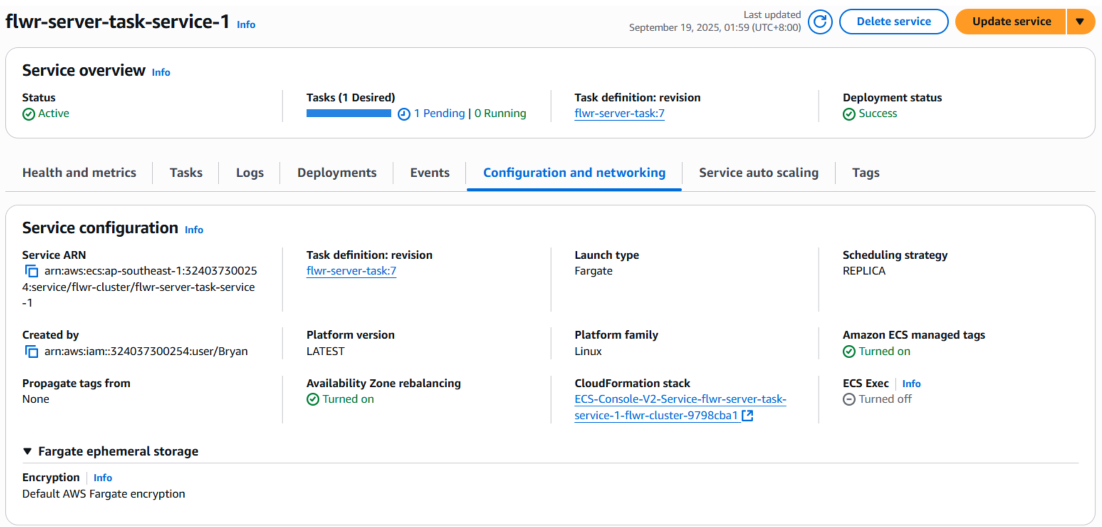


Figure 5.3.1.9.1: Service of ECS

After that, create service within ECS Cluster. The federated learning server was deployed as a Fargate service named flwr-server-task-service-1. The application type was set as Service and the type of launch was to be Fargate, where the EC2 instances were not necessary as the containerized Flower server was capable of being deployed without serverless launch. The service was defined to keep only single task running at any given time with the task definition of flwr-server-task:7. The scheduling plan was set to Replica, which will make the service execute the desired number of tasks on a regular basis. There was a successful deployment, with the service being active. Availability Zone balancing was activated to enhance reliability and default AWS Fargate encryption was placed on the ephemeral storage. Such a setup made sure that the federated learning server is constantly accessible to edge devices to train and aggregate models.

5.3.1.10 AWS EventBridge

a. AWS EventBridge Scheduler to Start Federated Learning Server

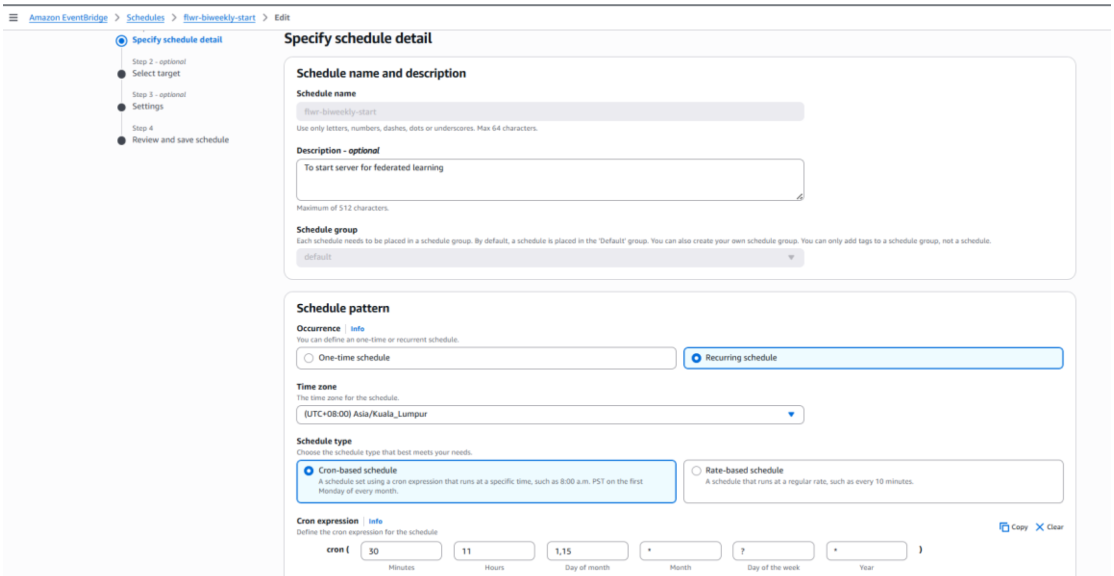


Figure 5.3.1.10.1: EventBridge Scheduler Setting for Start

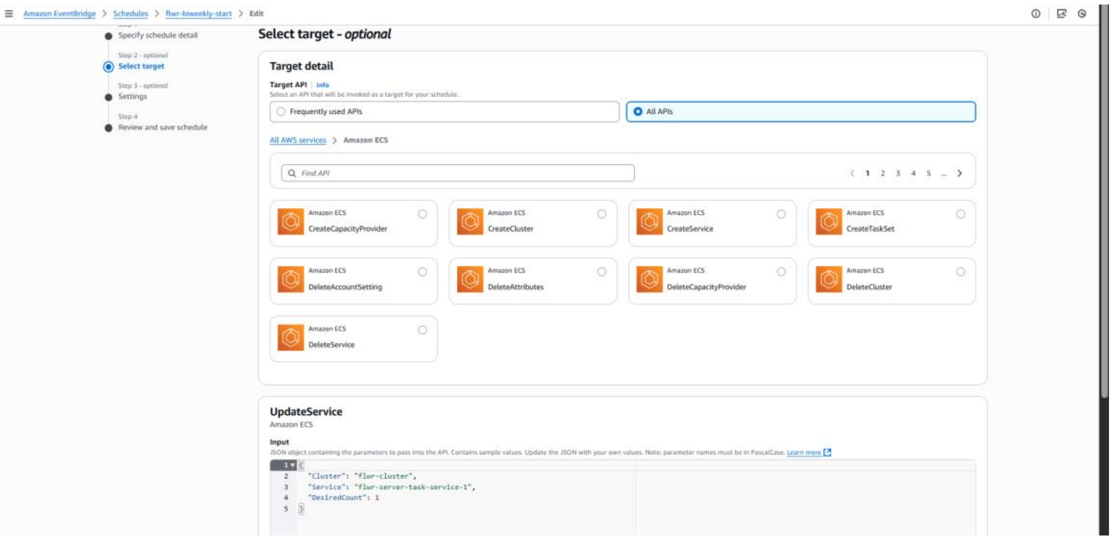


Figure 5.3.1.10.2: EventBridge Scheduler Setting for Start (Target)

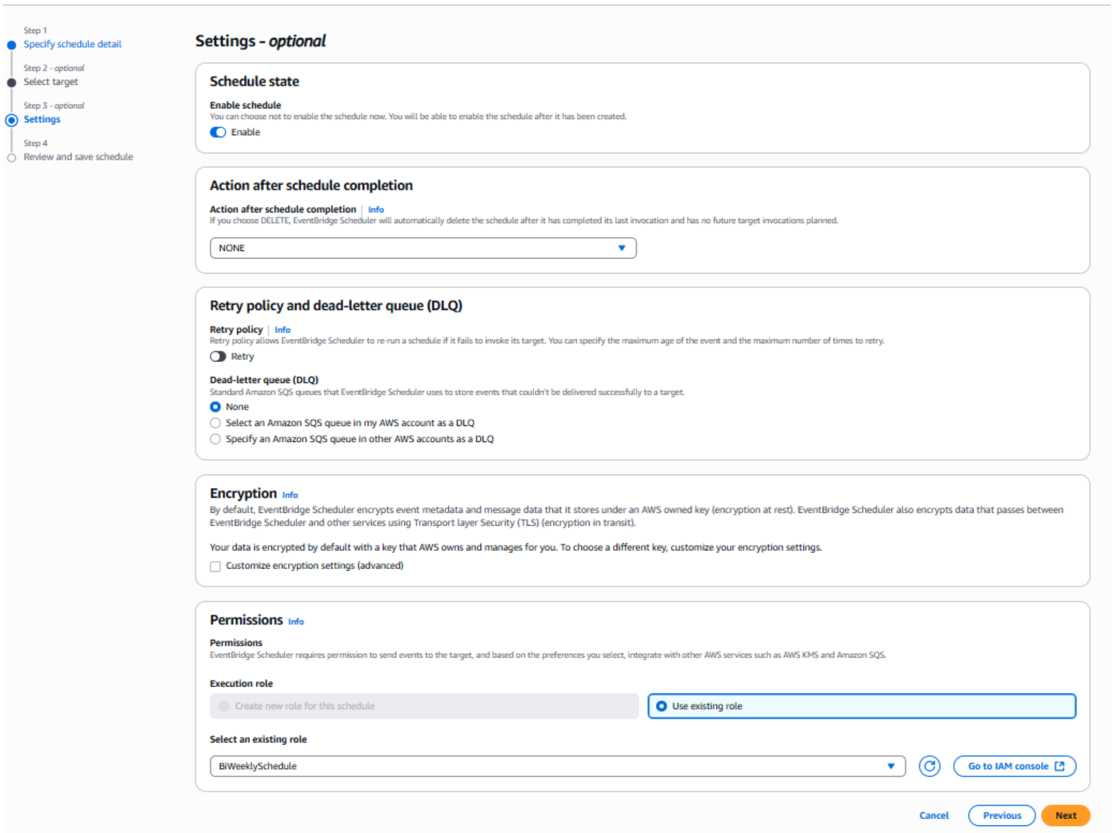


Figure 5.3.1.10.3: EventBridge Scheduler Setting for Start (Setting)

For the EventBridge Scheduler, create a new scheduler named flwr-biweekly-start to start the server for federated learning. For schedule pattern, choose “Recurring schedule” as occurrence, “Asia/Kuala_Lumpur” as time zone, “Cron-based schedule” as schedule type. For cron expression for schedule, configure “30” for minutes, “11” for hours, “1 and 15” for day of month, “*” for month, “?” for day of week “*” for year. It means that it will start the server at 11.30am of every 1st and 15th of the month. Last, set “Off” for flexible time window to make sure the server directly start on time. Then, for target API that will be invoked by schedule, choose “Amazon ECS Update Service” and paste the JSON object containing the cluster, service and the desired count for change, at the UpdateService part. Last, enable the schedule and choose the “BiWeeklySchedule” role for the permission.

b. AWS EventBridge Scheduler to Stop Federated Learning Server

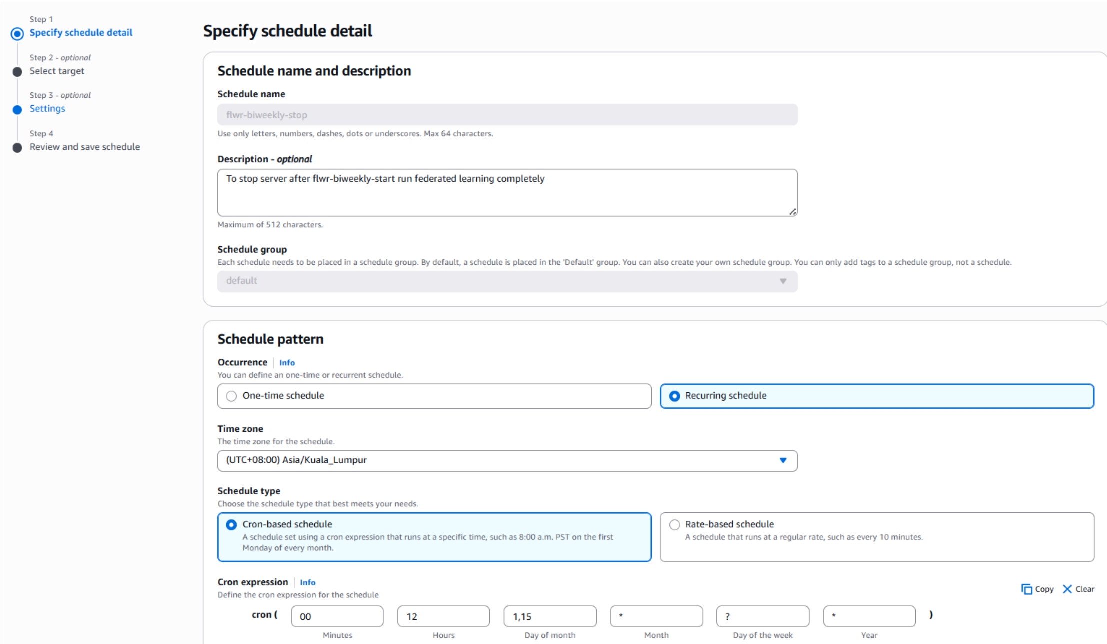


Figure 5.3.1.10.4: EventBridge Scheduler Setting for Stop

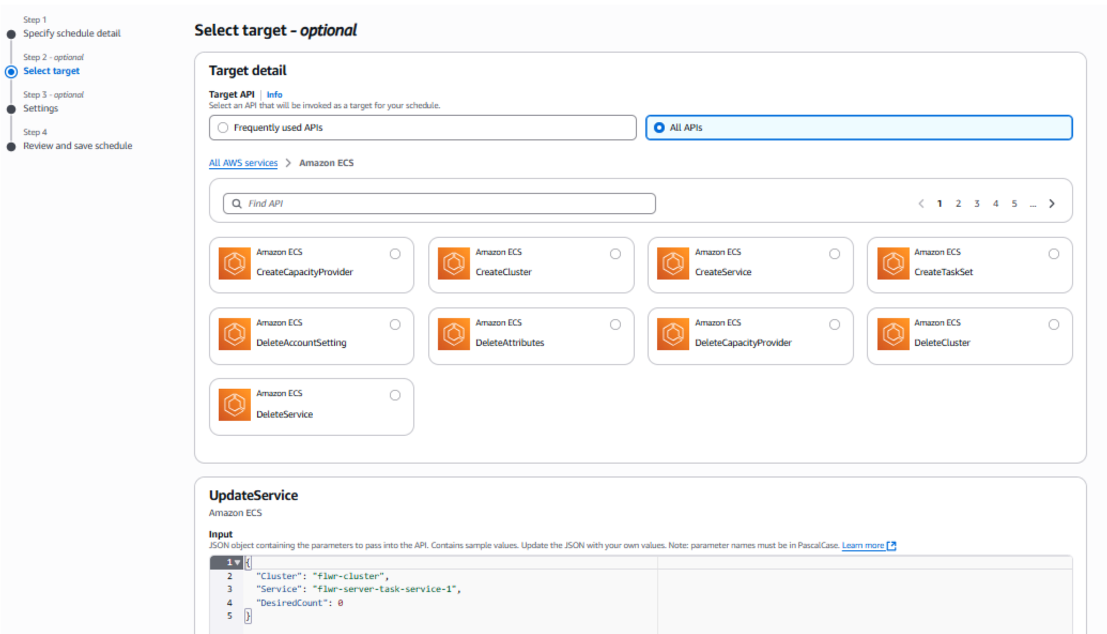


Figure 5.3.1.10.5: EventBridge Scheduler Setting for Stop (Target)

Then, create a new scheduler named flwr-biweekly-stop to stop the server for federated learning. For schedule pattern, configure “00” for minutes, “12” for hours and others same with start one. It means that it will stop the server at 12.00pm of every 1st and 15th of the month. Last, set “Off” for flexible time window to make sure the server directly stops on time. Then, for target API that will be invoked by schedule, choose “Amazon ECS Update Service” and paste the JSON object containing the cluster, service and the desired count for change, at the UpdateService part. Last, enable the schedule and choose the "BiWeeklySchedule" role for the permission.

5.3.1.11 AWS S3 Setup

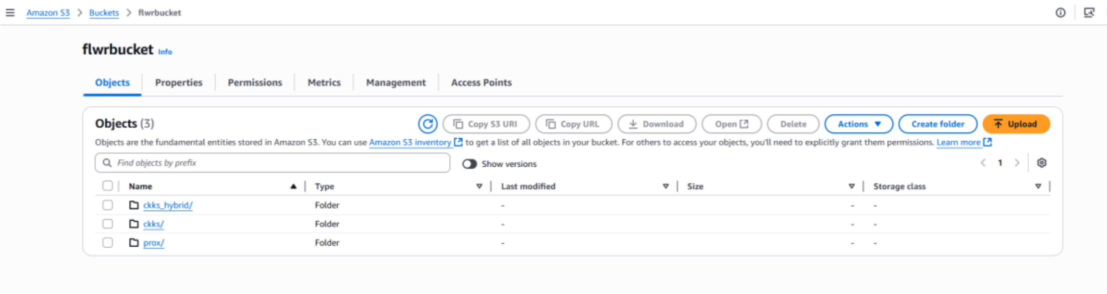


Figure 5.3.1.11.1: AWS S3 Bucket Setup

For the model storage, create a flwrbucket as the main bucket in S3. Then, create ckks_hybrid folder that will store the shared context for client and server used for encryption and decryption. Besides, also create a prox folder that will store the encrypted global model for the federated learning.

5.3.2 Serverless Data Synchronization Configuration

5.3.2.1 AWS IoT – MQTT Test Client

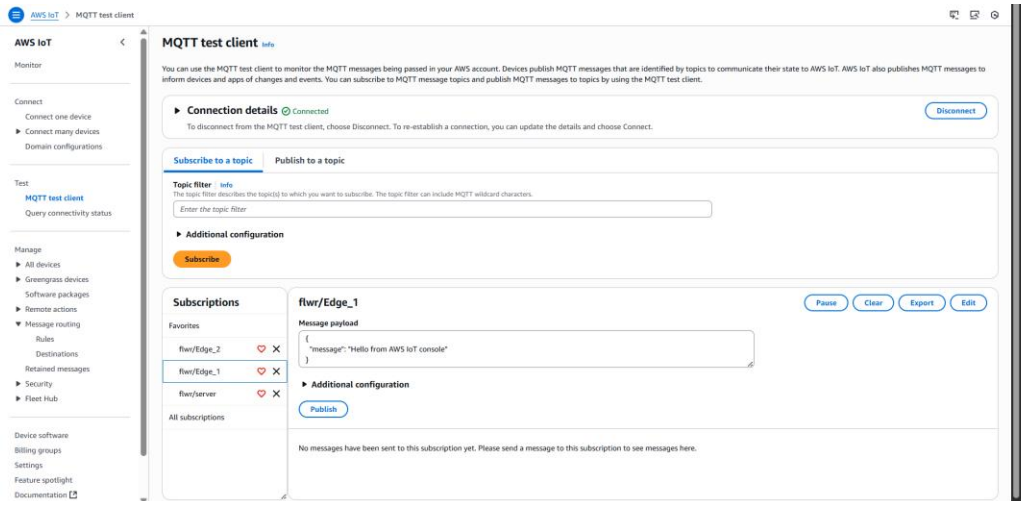


Figure 5.3.2.1.1: MQTT Topic

Subscribing to the topic "flwr/Edge_1" and "flwr/Edge_2" to receive messages. It will view the incoming MQTT messages in real-time

5.3.2.2 AWS IoT - Message Routing Rules

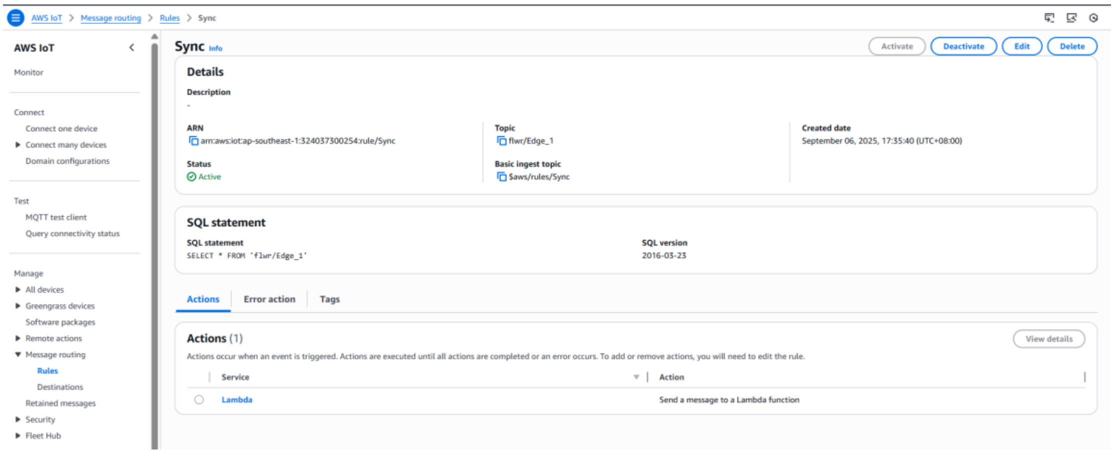


Figure 5.3.2.2.1: Message Routing Rule

Configuring a message routing rule called Sync to listen for the messages on the topic “flwr/Edge_1”. It uses a SQL statement (SELECT * FROM 'flwr/Edge_1') to process incoming messages. Then, set an action to send messages to Lambda function “data_sync_Edge_1” to handle data when triggered. After that, configure the same configuration for second edge to make sure configure separately.

5.3.2.3 AWS Lambda

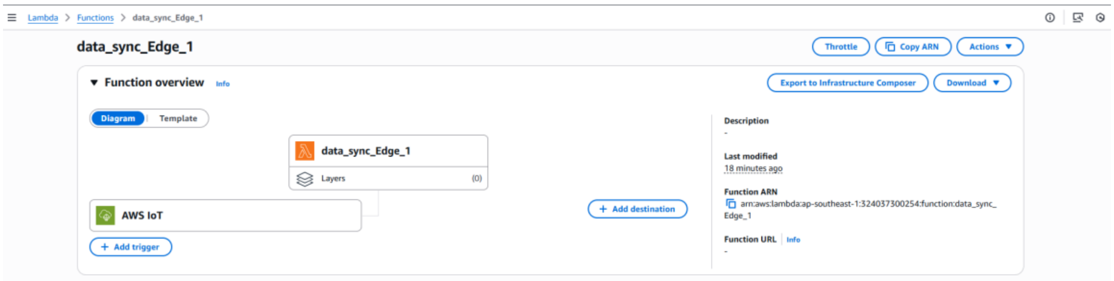


Figure 5.3.2.3.1 Lambda Add IoT Trigger

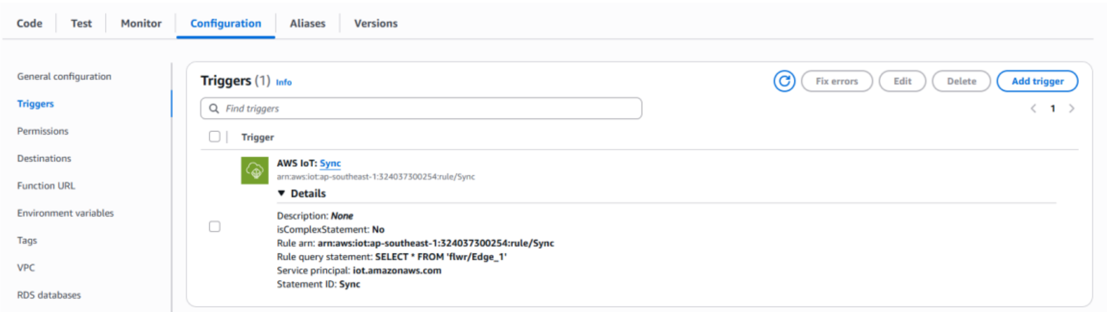


Figure 5.3.2.3.2 Trigger Details

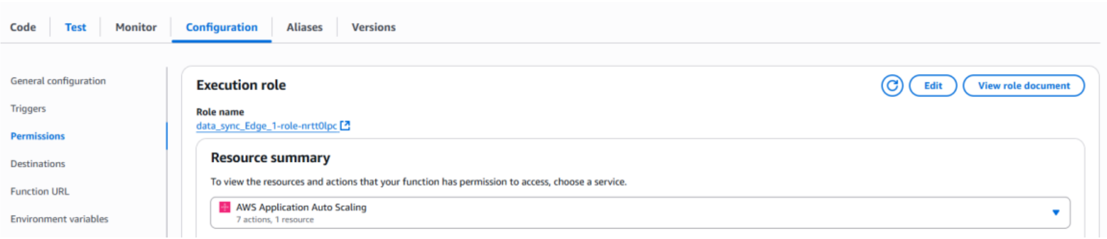


Figure 5.3.2.3.3 Execution Role for Lambda to Process Data

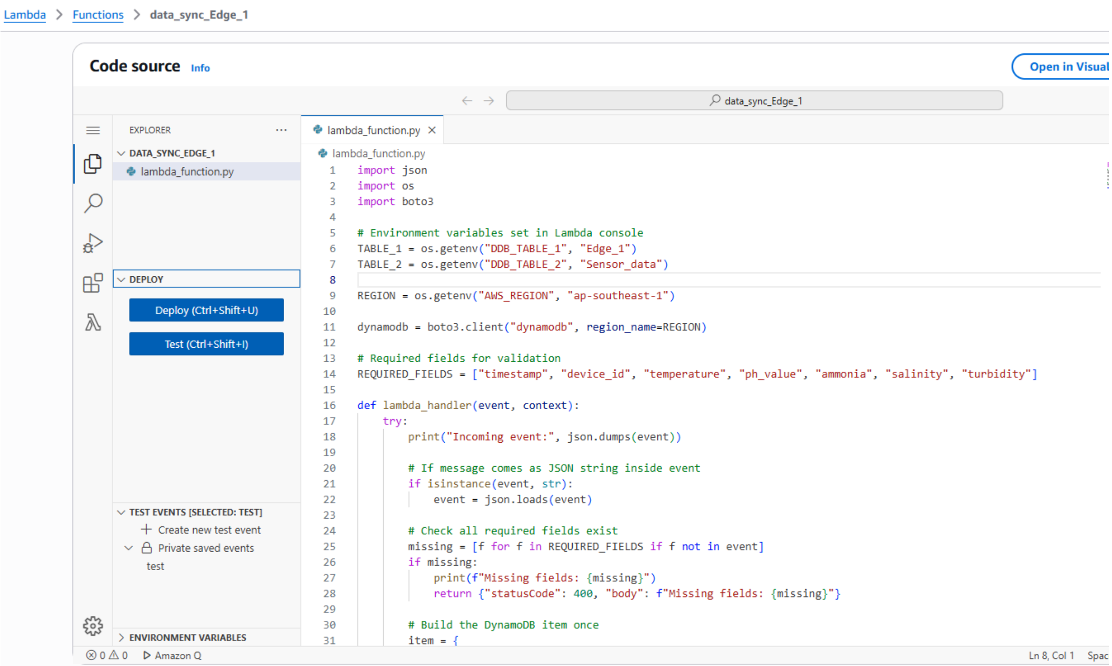


Figure 5.3.2.3.4: Lambda Function to Process Data into DynamoDB

Creating a lambda function named data_sync_Edge_1 then connect to AWS IoT as trigger so that the function that gets triggered by the IoT rule whenever a message is published to the flwr/Edge_1 MQTT topic. Then, select the data_sync_Edge_1-role-nrtt0lpc as execution role. Lastly, repeat the same configuration for Edge 2 with different lambda name “data_sync_Edge_2”, “flwr/Edge_2” MQTT topic and “data_sync_Edge_2-role-3w0otyga” as execution role.

5.3.2.4 DynamoDB

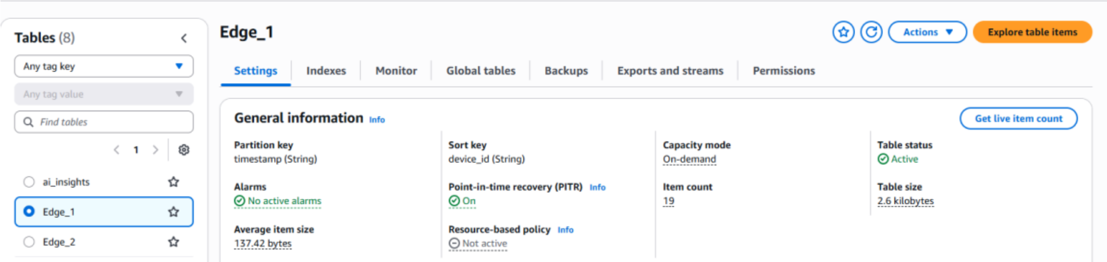


Figure 5.3.2.4.1: DynamoDB Setting

Create table "Edge_1" and "Edge_2" with timestamp as partition key and device_id as sort key. This table is to store the synchronized data.

5.3.3 Policies and Role Setting to Support System

5.3.3.1 ECS Task Role and Task Execution Role and Policies

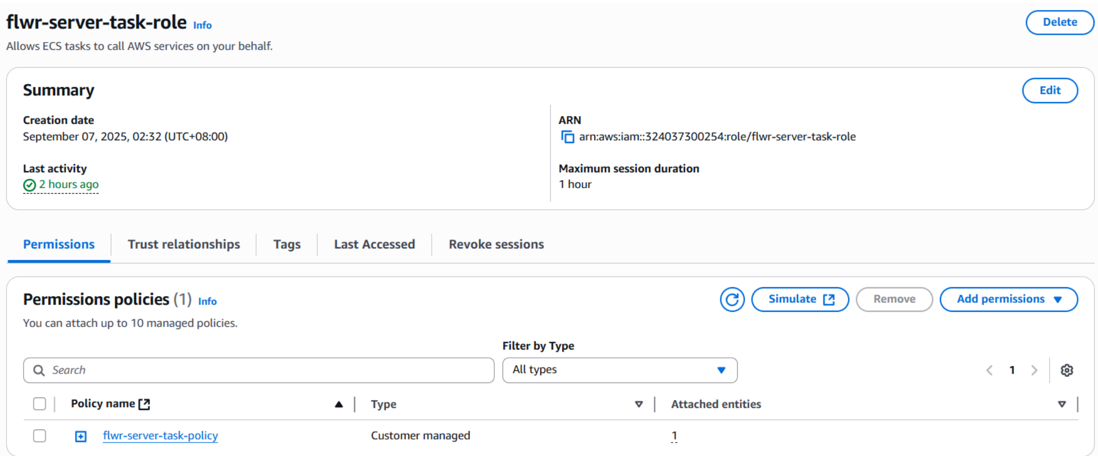


Figure 5.3.3.1.1: ECS Task Role and Task Execution Role and Policies

Configuration of a new role named **flwr-server-task-role** with the attached **flwr-server-task-policy** to handle any AWS services Flower server needs

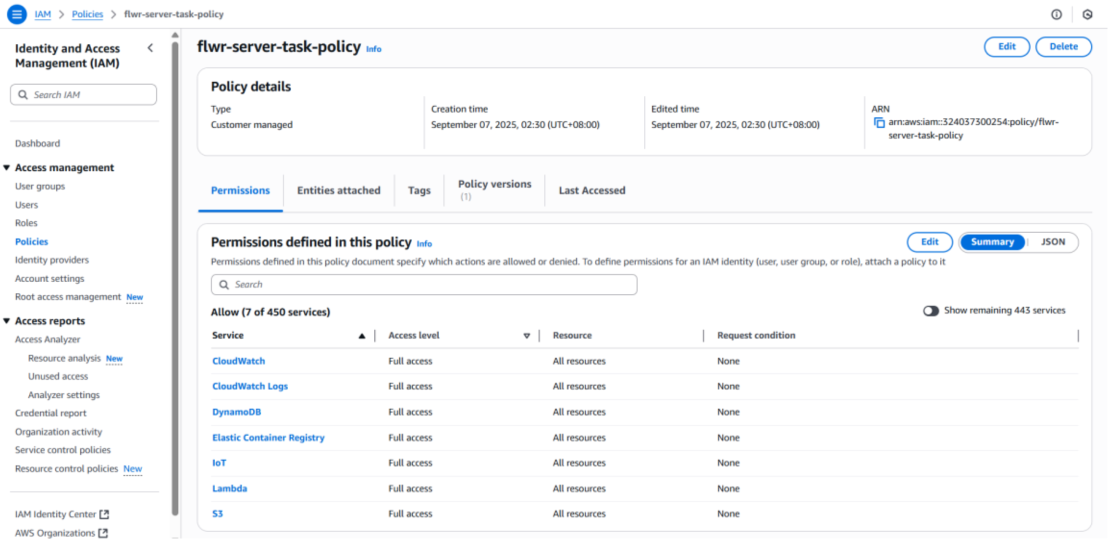


Figure 5.3.3.1.2: flwr-server-task-policy with Permissions

Configuration of a policy named **flwr-server-task-policy** with permission to ECR, Lambda, S3, IoT, DynamoDB, CloudWatch and CloudWatch Logs

5.3.3.2 BiWeeklySchedule Role and Policies

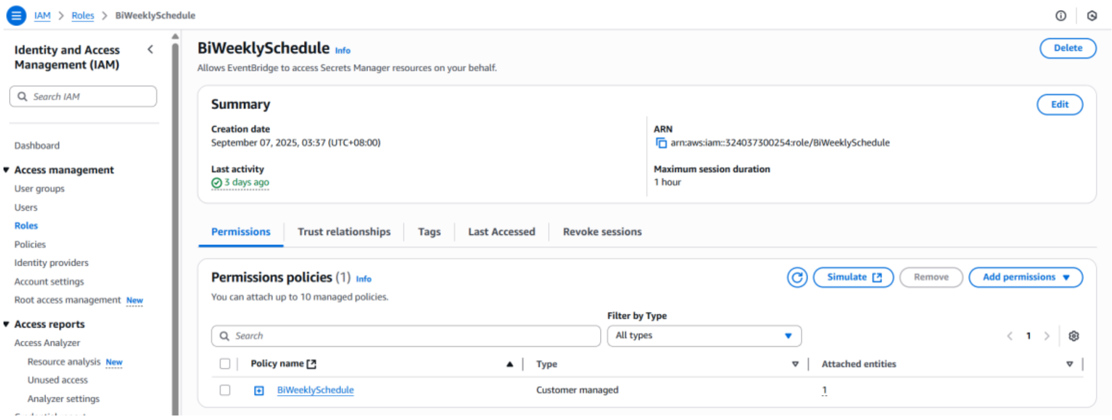


Figure 5.3.3.2.1: BiWeeklySchedule Role and BiWeeklySchedule Policy

Configuration of a BiWeeklySchedule role with attached BiWeeklySchedule policy to allow EventBridge Scheduler to make changes on the ECS.

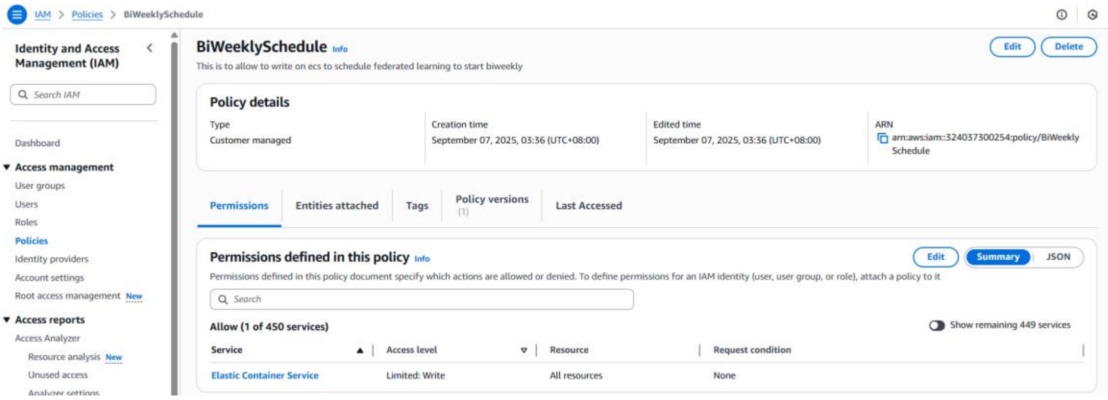


Figure 5.3.3.2.2: BiWeeklySchedule Policy

Configuration of BiWeeklySchedule policy to allow to write on the ECS service.

5.3.3.3 Data Synchronization Role

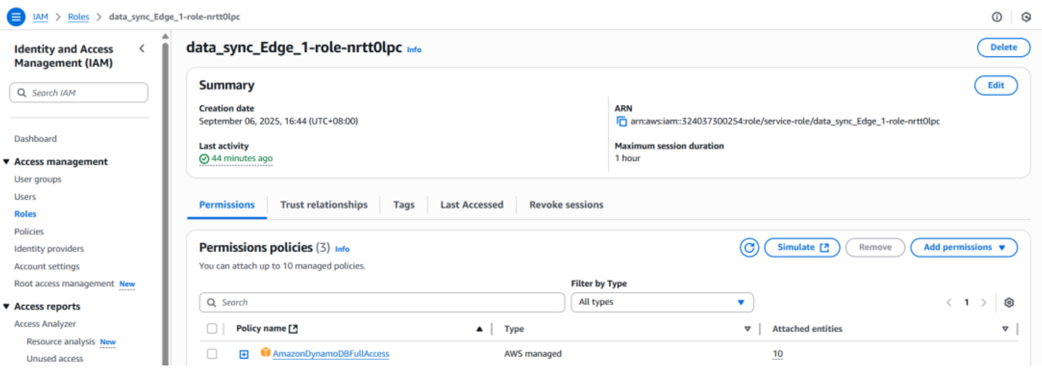


Figure 5.3.3.3.1: Data Synchronization Role

Configuration the data_sync_Edge_1-role-nrtt0lpc with AmazonDynamoDBFullAccess to allow Lambda function to use to write on the DynamoDB.

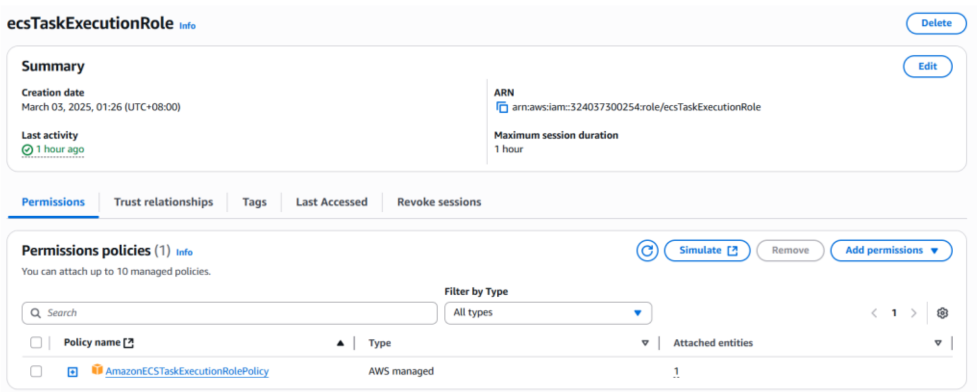


Figure 5.3.3.3.2: ECS Task Execution Role

Also, the system will use this default task execution role named ecsTaskExecutionRole which will handles pulling flower server docker image and send logs to CloudWatch.

5.3.4 Server Code Setting and Configuration

5.3.4.1 Server Code Configuration

flower_server.py

flwr_server > flower_server.py

```
class CustomFedProxStrategy(FedProx):
    def __init__(
        self,
        initial_parameters: Parameters,
        proximal_mu: float = 0.5,
        hybrid_handler: HybridEncryptionHandler = None,
    ) -> None:
        # Read from environment variables, with defaults
        min_fit_clients = int(os.getenv("MIN_FIT_CLIENTS", "2"))
        min_evaluate_clients = int(os.getenv("MIN_EVALUATE_CLIENTS", "2"))
        min_available_clients = int(os.getenv("MIN_AVAILABLE_CLIENTS", "2"))
        proximal_mu = float(os.getenv("PROXIMAL_MU", "0.5"))
        fraction_fit = float(os.getenv("FRACTION_FIT", "1.0"))
        fraction_evaluate = float(os.getenv("FRACTION_EVALUATE", "1.0"))

        super().__init__(
            fraction_fit=fraction_fit,
            fraction_evaluate=fraction_evaluate,
            min_fit_clients=min_fit_clients, # min clients to participate in training
            min_evaluate_clients=min_evaluate_clients, # min clients for start a round
            min_available_clients=min_available_clients, # min clients for start a round
            initial_parameters=initial_parameters,
            proximal_mu=proximal_mu,
        )
        if hybrid_handler is None:
            raise RuntimeError("Hybrid encryption handler must be provided and initialized")
        self.hybrid_handler: HybridEncryptionHandler = hybrid_handler
```

Figure 5.3.4.1.1: Code Snippet of Server Environment Variable Setting

The default configuration for server code in shown in the figure. It can be modified by the admin at the environment variable part of task definition based on the demand for the federated learning. The default configuration for server is set as shown in the Figure 5.3.4.1.1.

5.3.4.2 Hybrid Encryption Handler Configuration

```
hybrid_encryption_handler.py ×
flwr_server > hybrid_encryption_handler.py

# S3 Configuration
BUCKET_NAME = "flwrbucket"
BASE_PREFIX = "ckks_hybrid"

#   This handler uses a shared CKKS context that allows the server to decrypt param
#   for aggregation while maintaining encryption during transmission for privacy.
class HybridEncryptionHandler:

    def __init__(self, is_server: bool = False, client_id: Optional[int] = None):

        self.is_server = is_server
        self.client_id = client_id
        self.s3 = boto3.client('s3')
        self.context = None
        self.is_initialized = False
        self.context_fingerprint: Optional[str] = None

        # Parameter scaling for precision
        self.scale_factor = 1e3  # Further reduced scale for better stability
```

Figure 5.3.4.2.1: Code Snippet of HybridEncryptionHandler Class

```
hybrid_encryption_handler.py ×
flwr_server > hybrid_encryption_handler.py

# Utility functions for easy integration
def create_server_hybrid_handler() -> HybridEncryptionHandler:
    # Create a server-side hybrid encryption handler.
    handler = HybridEncryptionHandler(is_server=True)
    if handler.initialize_context():
        return handler
    else:
        raise RuntimeError("Failed to initialize server hybrid encryption handler")


def create_client_hybrid_handler(client_id: int) -> HybridEncryptionHandler:
    # Create a client-side hybrid encryption handler.
    handler = HybridEncryptionHandler(is_server=False, client_id=client_id)
    if handler.initialize_context():
        return handler
    else:
        raise RuntimeError(f"Failed to initialize client {client_id} hybrid encryption handler")
```

Figure 5.3.4.2.2: Code Snippet of Handler Initialization Function

In the hybrid encryption handler code, configure the correct path for s3 bucket to make sure server able to access to the shared context for usage or upload in the federated learning. Then, the CKKS-based hybrid encryption code was packed with server code to pushed to Docker to ECR. Its corresponding functions will be called when flower server need it.

```
try:
    param_arrays = parameters_to_ndarrays(parameters)
    original_shapes = [param.shape for param in param_arrays]
    scaled_params = self._scale_parameters(param_arrays)
    encrypted_params = []
    for i, param in enumerate(scaled_params):
        flat_param = param.flatten().tolist()
        encrypted_param = ts.ckks_vector(self.context, flat_param)
        encrypted_data = {
            'data': encrypted_param.serialize(),
            'shape': original_shapes[i],
            'param_index': i,
            'ctx_fp': self.context_fingerprint,
            'plain_hash': self._hash_quantized(np.asarray(param, dtype=np.float32)),
        }
        serialized_data = pickle.dumps(encrypted_data)
        encrypted_params.append(serialized_data)
    serialized_params = []
    for encrypted_data in encrypted_params:
        byte_array = np.frombuffer(encrypted_data, dtype=np.uint8)
        serialized_params.append(byte_array)
    encrypted_parameters = ndarrays_to_parameters(serialized_params)
    logger.info(f"{'[Server]' if self.is_server else f'[Client {self.client_id}]'} Encrypt
    return encrypted_parameters
```

Figure 5.3.4.2.3: Code Snippet of How Parameters Are Turned into Encrypted Parameters

To configure to conduct encryption, the HybridEncryptionHandler begins by **converting the Flower Parameters object into numpy arrays** while recording their original shapes. To ensure numeric stability, the parameter values are **scaled by a factor $\boldsymbol{a = 10^3}$** and **clipped within the range $\boldsymbol{[-10^{-6}, 10^6]}$.** Then, each array is then **flattened** into a one-dimensional list and **encrypted into a CKKS vector** using the CKKS scheme via ts.ckks_vector [22]. The resulting encrypted vector is **serialized into bytes and combined with essential metadata to form a package dictionary** which includes serialized encrypted parameters, the original array shape, the parameter index, a SHA-256 fingerprint of the shared CKKS context (ctx_fp), and a SHA-256 hash of the parameter after three-decimal quantization (plain_hash) to allow **for noise-tolerant integrity verification**. This package is **serialized using pickle, converted into a np.uint8 array, and finally wrapped back into the Flower Parameters format**, ready for secure transmission.

For server, encryption happens when it needs to encrypt the sent parameters each round for training and evaluation. While for client, encryption happens when it needs to encrypt local model parameters before sent back to server.

```
decrypted_params = []
for i, encrypted_array in enumerate(encrypted_arrays):
    try:
        if isinstance(encrypted_array, np.ndarray):
            encrypted_bytes = encrypted_array.astype(np.uint8).tobytes()
        else:
            encrypted_bytes = encrypted_array
        encrypted_data = pickle.loads(encrypted_bytes)
        if not isinstance(encrypted_data, dict) or 'data' not in encrypted_data or 'shape' nc
            raise ValueError("Invalid encrypted parameter structure")
        if 'ctx_fp' not in encrypted_data or self.context_fingerprint is None:
            raise ValueError("Missing context fingerprint; cannot decrypt")
        if encrypted_data['ctx_fp'] != self.context_fingerprint:
            raise ValueError("Context fingerprint mismatch; cannot decrypt with current conte
        original_shape = encrypted_data['shape']
        encrypted_param_data = encrypted_data['data']
        encrypted_param = ts.ckks_vector_from(self.context, encrypted_param_data)
        decrypted_flat = encrypted_param.decrypt()
        expected_size = np.prod(original_shape)
        if len(decrypted_flat) < expected_size:
            raise ValueError(f"Decrypted data size {len(decrypted_flat)} < expected {expectec
        else:
            decrypted_param = np.array(decrypted_flat[:expected_size], dtype=np.float32)
        decrypted_param = decrypted_param.reshape(original_shape)
        if 'plain_hash' not in encrypted_data:
            raise ValueError("Missing plaintext hash; cannot verify decryption integrity")
        calc_hash = self._hash_quantized(np.asarray(decrypted_param, dtype=np.float32))
        if calc_hash != encrypted_data['plain_hash']:
            raise ValueError("Plaintext hash mismatch after decryption")
        decrypted_params.append(decrypted_param)
    except Exception as e:
        logger.error(f"Failed to decrypt parameter {i}: {e}")
        raise
unscaled_params = self._unscale_parameters(decrypted_params)
decrypted_parameters = ndarrays_to_parameters(unscaled_params)
```

Figure 5.3.4.2.4: Code Snippet of How Decryption is Conducted with Integrity Check

To configure to conduct decryption, for both the server and client, their HybridEncryptionHandler **convert the Flower Parameters object into numpy arrays** of type np.uint8. For each array, the raw bytes are extracted and **deserialized with pickle.loads** to recover the original package containing the encrypted data and its metadata. The function then validates the package structure, ensuring that both the **context fingerprint (ctx_fp) and the plaintext hash (plain_hash)** are present. To guarantee that the correct CKKS context and keys are being used, the **local context fingerprint is compared with the ctx_fp stored in the package**. Once verified, the encrypted data is **reconstructed into a CKKS vector** and **decrypted into a list of floating-point numbers** [22]. The function enforces shape consistency by truncating or reshaping the list to match the exact number of elements specified by the original array shape. To confirm data integrity, a new **quantized plaintext hash is computed and compared against the stored plain_hash**, which allows for tolerance to CKKS noise. Finally, the values are **unscaled by dividing by the scaling factor α**, and the results are converted back into the Flower Parameters format for further use.

For server, the decryption happens when it decrypts incoming client updates for aggregation, aggregation of result to update its in-memory global model and loading the encrypted model from S3 when bootstrapping.

For clients, the decryption happens it needs to decrypt server's global parameters before local training or evaluation.

5.4 System Operation

5.4.1 Federated Learning Server in ECS

To start federated learning server, there are two methods which are starting at schedule or starting with manual.

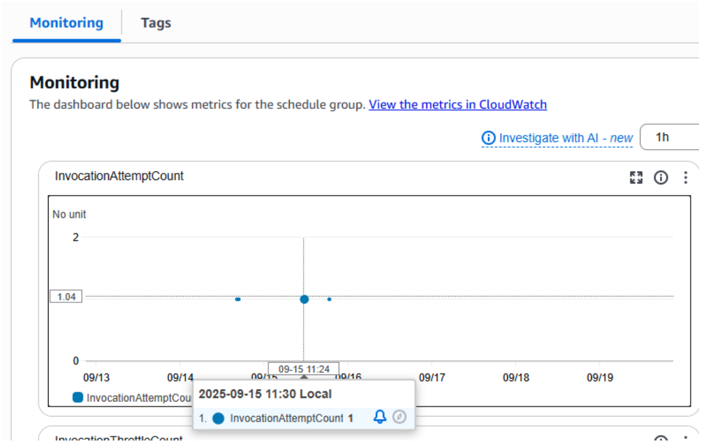


Figure 5.4.1.1: Invocation Attempt Count of Start Scheduler

To start with schedule, the EventBridge Scheduler with start at 1 and 15 of every month at 11.30 a.m. The invocation attempt count shown in the figure shows that the scheduler invoked the schedule to start the federated learning server successfully.

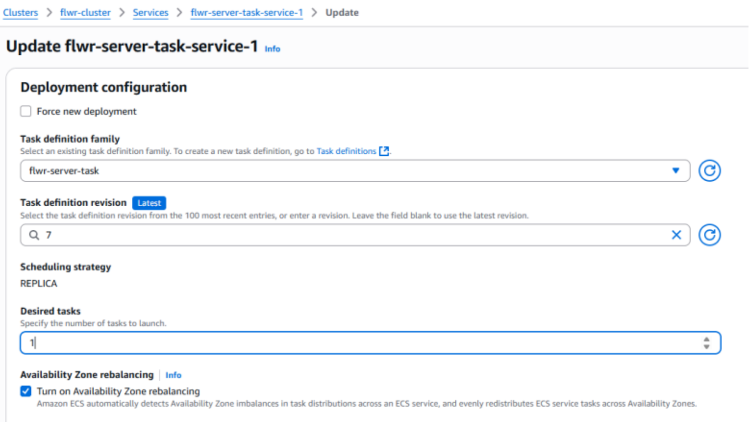


Figure 5.4.1.2: Update ECS Service

To start with manual, admin can change the desired tasks from 0 to 1 in the flwr-server-task-service-1 to start the federated learning server.

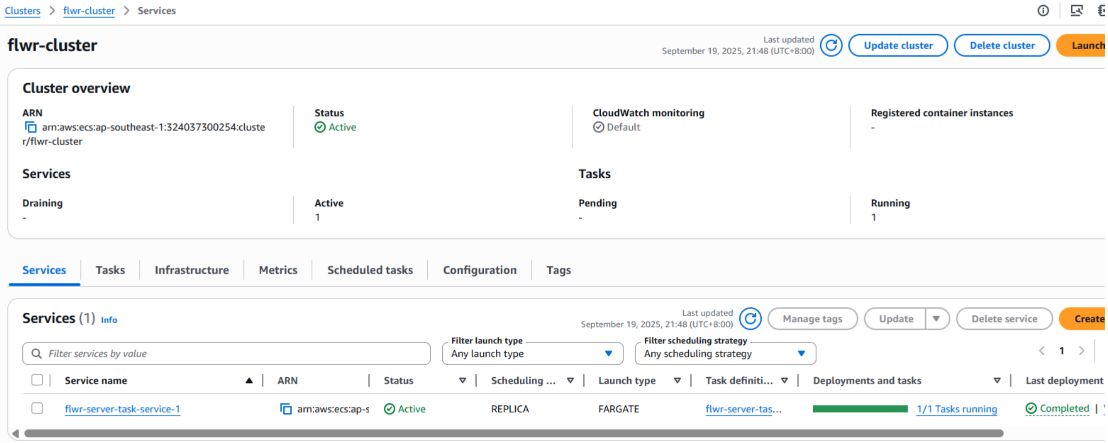


Figure 5.4.1.3: Showing Running Server

The figure 5.4.1.3 shows that the service is running. It means the server start the federated learning and able to receive client requests.

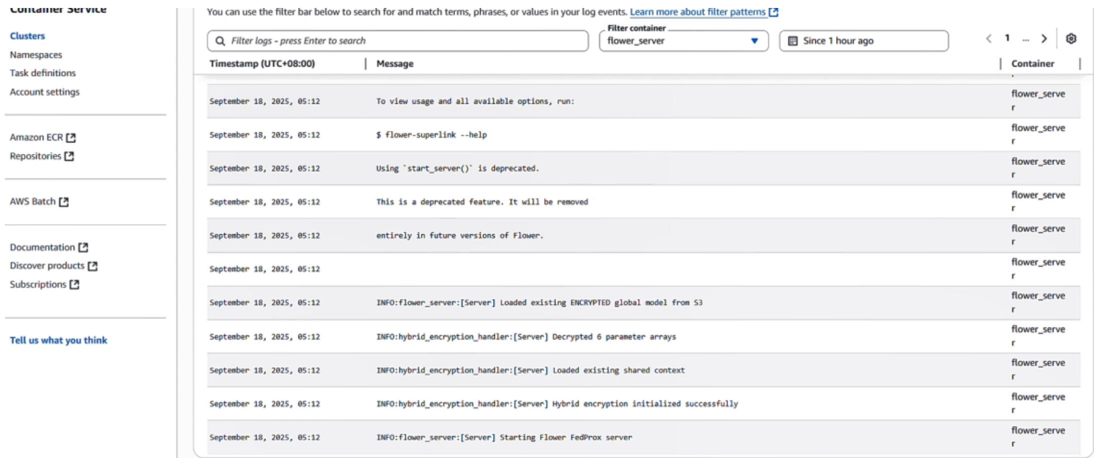


Figure 5.4.1.4: Federated Learning Server Load Context Log

The figure 5.4.1.4 shows a federated learning process with three clients. When the federated learning start, the server loads the shared context and initializes the hybrid encryption handler. It also assesses to the existing encrypted global model in s3, decrypts and load the model.

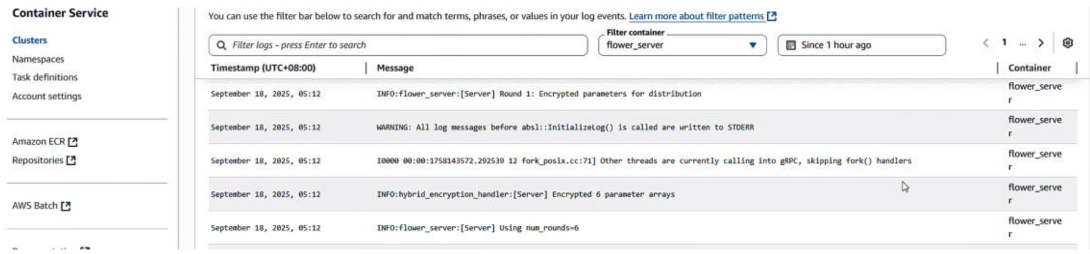


Figure 5.4.1.5: Server Encryption and Distribute Model Log

Then, the server encrypts the model parameters and send to clients. The federated learning will run 6 rounds as shown in the log.

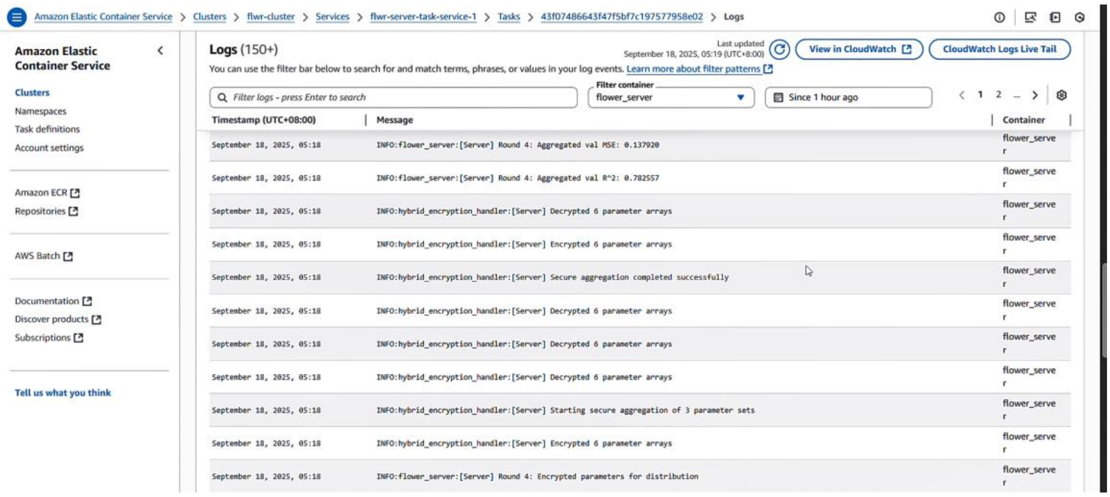


Figure 5.4.1.6: Server Decryption and Aggregation Log

After server received encrypted local updates from three clients, it decrypted each clients' encrypted parameters then perform aggregation. After aggregation, it encrypted the updated global model and sent to the three clients for evaluation. After that, it decrypted the aggregated model again to update the server's global model.

Besides, server also aggregated the metrics returned by the clients to show the federated learning performance.

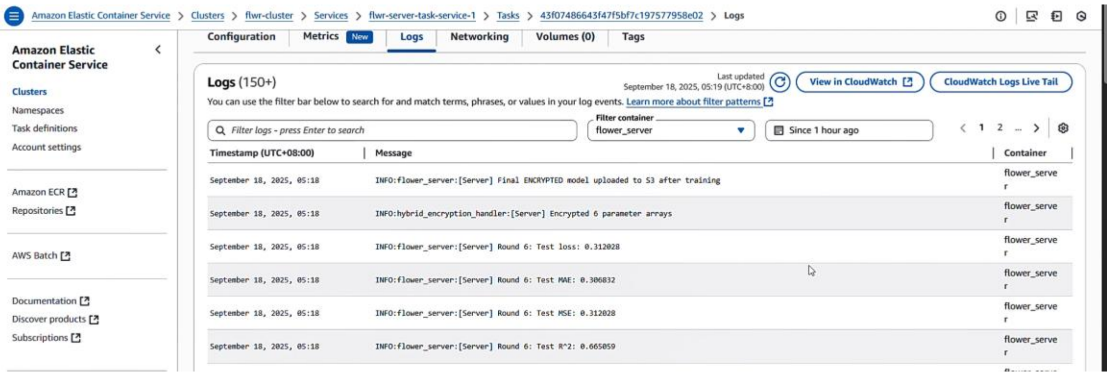


Figure 5.4.1.7: Server Upload Final Model

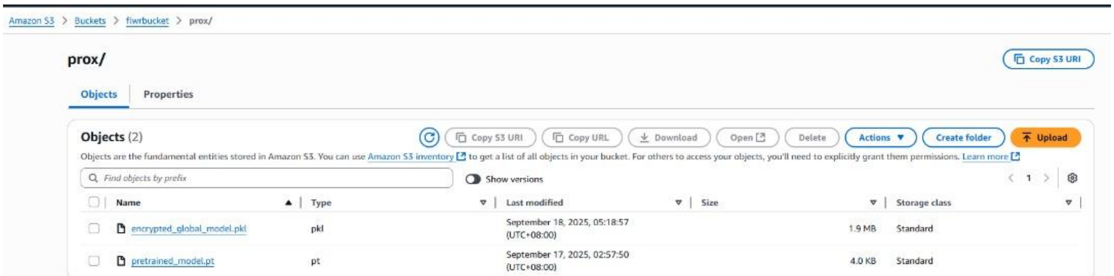


Figure 5.4.1.8: Model Successfully Upload to S3

At the end of the federated learning, server encrypted the final global model then upload to the S3 successfully.

5.4.2 Federated Learning Client

To join the federated learning, the clients either join the federated learning by invoked with the MQTT protocol or join manually.

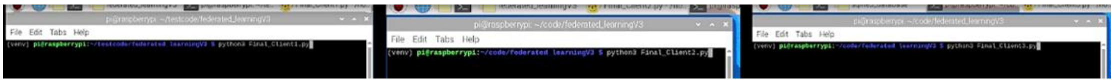


Figure 5.4.2.1: Client Join the Federated Learning Manually

Three raspberry pi manually run the client script to join the federated learning.

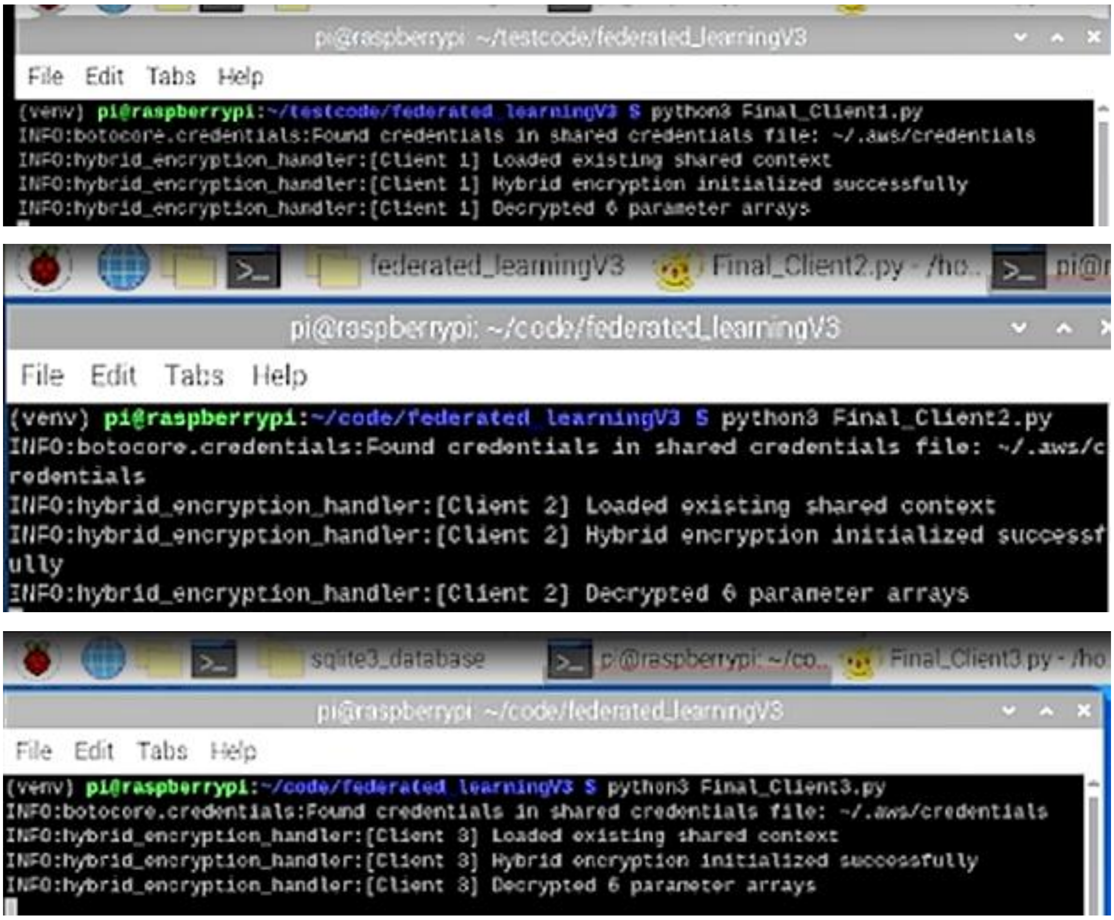


Figure 5.4.2.2: Start of the FL on Three Clients Side

At the start of the federated learning, the three clients load the existing shared context from the s3 and initialize the hybrid encryption handler. They also download the encrypted global model from s3, decrypt and load into model.

Figure 5.4.2.3: gRPC Channel Open for Communication

gRPC channel is opened for clients to join the federated learning to communicate with the server.

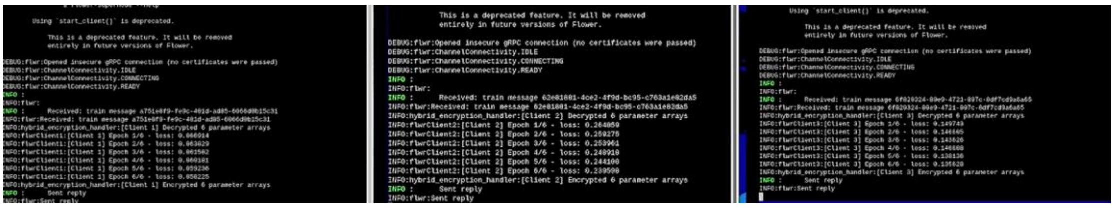


Figure 5.4.2.4: Clients' Training in Federated Learning

The three clients received the train message from server successfully, it means they also received the encrypted global parameters from the server. They started to decrypt server-sent global parameters and loads them into their local model and replacing their current weights. They then train locally with the new weights. After training, they encrypted their updated local model parameters and sent to server as shown in the figure above. Besides, they also sent the performance metrics of the training on validation set to server in plaintext.

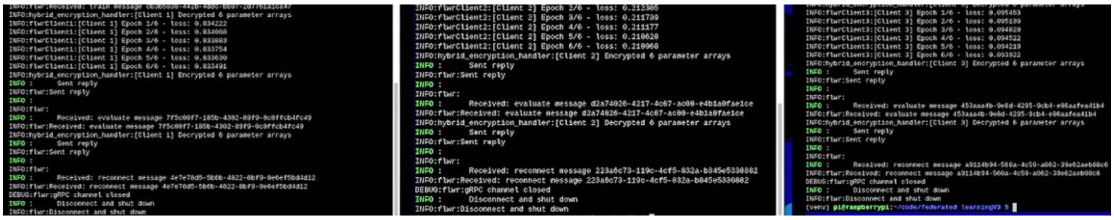


Figure 5.4.2.5: Clients' Evaluation and Disconnect After FL Completion

After aggregation from server, the clients received the evaluation message from server to evaluate the aggregated global model. They decrypted it and loaded the aggregated model and train on test set. Then, they return the test performance metrics in plaintext. This is just a review on the aggregation result not the parameters, so no encryption and decryption happen on sending back the performance metrics. But all things involve with parameters are protected by the CKKS hybrid encryption.

After all rounds of federated learning complete, the Grpc channel is closed and all clients disconnect successfully.

5.4.3 Data Synchronization

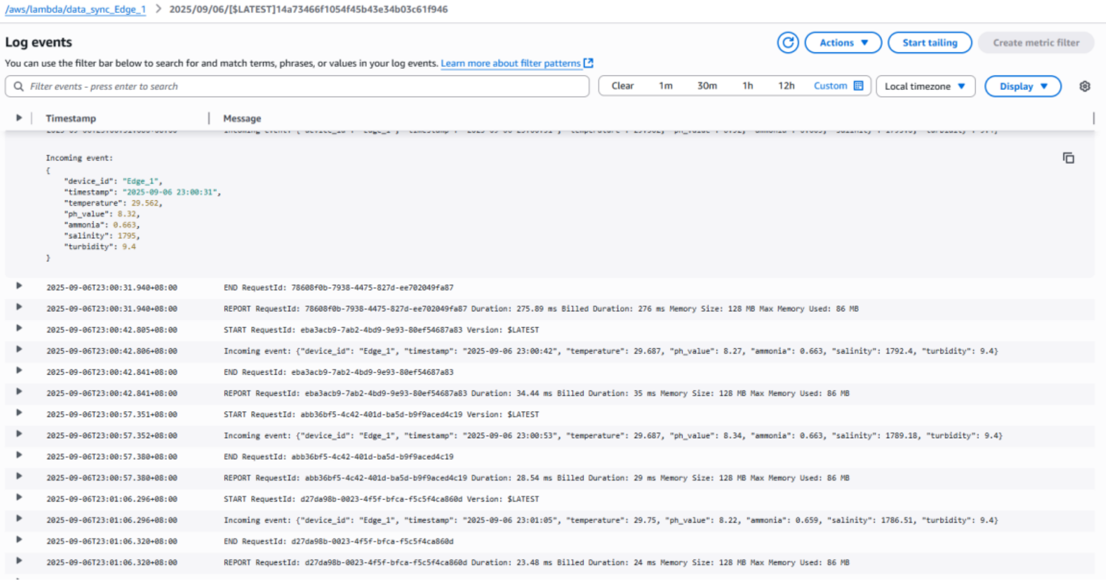


Figure 5.4.3.1: Log Event of the Lambda function of data_sync_Edge_1

When client runs synchronization, the Lambda successfully received the event in JSON format which is the client's sensor data then put them into the table in DynamoDB.

Table: Edge_1 - Items returned (20)

Scan started on September 19, 2025, 23:36:05

timestamp (String)	device_id (String)	ammonia	ph_value	salinity	temperature	turbidity
2025-09-06 22:01:58	Edge_1	0.05	7.2	3.1	26.7	1.8
2025-09-06 23:00:31	Edge_1	0.663	8.32	1795	29.562	9.4
2025-09-06 23:00:42	Edge_1	0.663	8.27	1792.4	29.687	9.4
2025-09-06 23:00:53	Edge_1	0.663	8.34	1789.18	29.687	9.4
2025-09-06 23:01:05	Edge_1	0.659	8.22	1786.51	29.75	9.4
2025-09-06 23:01:17	Edge_1	0.662	8.23	1783.44	29.625	9.4

Figure 5.4.3.2: The Synchronized Data in the Table of Edge_1 in DynamoDB

The figure shows Lambda successfully process and put the sensor data in JSON message into the table.

5.4.4 Encrypted Global Model to Predict Dissolved Oxygen

```
Mode: decrypt
1. ts=2025-09-20 17:38:00 temp=25.01 ph=6.05 nh3=0.34 sal=3.42 turb=96.51 do_pred=6.89
2. ts=2025-09-20 17:39:00 temp=25.04 ph=7.72 nh3=0.35 sal=3.49 turb=96.41 do_pred=7.42
3. ts=2025-09-20 17:40:00 temp=25.08 ph=6.17 nh3=0.39 sal=3.42 turb=96.43 do_pred=9.42
4. ts=2025-09-20 17:41:00 temp=25.02 ph=6.45 nh3=0.34 sal=3.46 turb=96.32 do_pred=9.31
5. ts=2025-09-20 17:42:00 temp=25.02 ph=6.45 nh3=0.38 sal=3.48 turb=96.60 do_pred=7.31
6. ts=2025-09-20 17:43:00 temp=25.02 ph=6.44 nh3=0.33 sal=3.46 turb=96.34 do_pred=7.57
7. ts=2025-09-20 17:44:00 temp=25.07 ph=6.60 nh3=0.39 sal=3.43 turb=96.14 do_pred=9.57
8. ts=2025-09-20 17:45:00 temp=25.09 ph=6.26 nh3=0.30 sal=3.42 turb=96.58 do_pred=8.82
9. ts=2025-09-20 17:46:00 temp=25.09 ph=6.05 nh3=0.40 sal=3.43 turb=96.46 do_pred=10.22
10. ts=2025-09-20 17:47:00 temp=25.01 ph=7.25 nh3=0.33 sal=3.43 turb=96.05 do_pred=12.22
```

Figure 5.4.4.1: Ten outputs of the Dissolved Oxygen Prediction

Client loads the final encrypted global model and runs per minutes to real time predict the dissolved oxygen based on the five parameters.

5.5 Implementation Issues and Limitations

While the project demonstrated the feasibility and effectiveness of server-side federated learning for precision aquaculture, there are several implementation issues and limitations that could be addressed in the future work.

The first implementation issue and limitation are lacking the dissolved oxygen sensor to collect the ground truth data for the training. The current implementation still uses the predicted dissolved oxygen value based on the sensor data to support local training and federated learning. Hence, the prediction may differ from the ground truth but edge will not know about it.

Besides, the second implementation issue and limitation are enhanced model complexity. As the system matures, a more sophisticated ML models could be implemented to improve the ability or function of model. For example, the model now is able processing computer vision data to analyse the fish or prawn behaviour.

Moreover, the third limitation is the encryption implementation in model. Due to the technical difficulty and the chance to degrade the model performance, full homomorphic encryption in this project is not implemented in the system. Future work may emphasize on this to solve the issue of model accuracy performance and the computation overhead.

5.6 Concluding Remark

The system managed to deploy federated learning in a federated edge cloud structure that can scale. The federated learning server was deployed and posted in the AWS Cloud, which facilitated the communication between the external clients with the server. The training can be initiated in a given schedule or manually and the server can carry out aggregation and access the Amazon S3 to store the models. Regarding scalability, the system will have flexibility in the participation as clients can seamlessly join the federated learning process based on the demand.

Besides, the system also increases security by deploying CKKS Hybrid Encryption. This secures the parameters of the model during transmission against attack possible. The server can effectively encrypt global model parameters and transmit them to clients, and the clients can decrypt, update and re-encrypt the parameters and transmit back to the server. By this way, this technique ensures secure protection of all communications.

Moreover, an efficient data synchronization pipeline was created to deal with the issues related to the intermittent network connectivity. The AWS Lambda function receives data when a client posts it to an MQTT topic, and it will be put by Lambda into DynamoDB. The data is stored in the local database in case of instability in the network and will automatically be synchronized into DynamoDB when the connection is restored.

Last, the final encrypted global model is able to used by local clients to run on edge to make real time prediction on dissolved oxygen value.

CHAPTER 6

System Evaluation And Discussion

6.1 System Testing and Performance Metrics

In this project, the system was evaluated on two complementary axes which are **predictive performance** of the dissolved-oxygen (DO) prediction model and **system-level properties** such as communication reliability, encryption correctness, scalability and deployment behaviour.

Predictive performance is assessed using mae, mse and R^2. By having these metrics together, they give a robust view of DO prediction quality. To support the metrics, the project also apply 8:1:1 testing technique to provide validation metrics and testing metrics that provide more comprehensive explanation for the prediction.

System-level tests focused on communication path between client and server, the CKKS-based encryption and decryption lifecycle, and scheduling and synchronization behaviour. Representative logs and snapshots are provided in Chapter 4 and the preliminary results section.

6.2 Testing Setup and Result

6.2.1 Testbed and Resources

The evaluation was performed on the deployed system introduced in Chapter 4 with real cloud server and Raspberry Pi clients. A server running on AWS ECS hosted the server component with models stored in S3 and synchronization handled by MQTT/Lambda/DynamoDB, with clients deployed as Raspberry Pis running an AWS IoT Greengrass client and a Flower client. These resources served as a basis for realistic evaluation of performance, encryption overhead and federated learning behaviour.

A summary of the testbed hardware and software resources is presented in Table 6.2.1.1.

Component	Specification	Purpose
Server	AWS ECS Fargate (vCPU: 1, Memory: 6 GB), Python 3.x, Flower, PyTorch	Hosts federated server and aggregation
Storage & Messaging	AWS S3, MQTT, DynamoDB, Lambda	Model storage and synchronization pipeline
Client	Raspberry Pi 5, AWS IoT Greengrass	Local training, encryption, synchronization
Network	WiFi broadband	Communication between clients and cloud

Table 6.2.1.1: Testbed resources used for evaluation

6.2.2 Dataset and Preprocessing

```python
def build_dataloaders():
    # Pseudo-labeled local data
    pseudo_df = generate_pseudo_labels_from_local()
    feature_columns = ["temperature", "turbidity", "ph_value", "ammonia", "salinity"]

    features = pseudo_df[feature_columns].values
    target = pseudo_df["Dissolved Oxygen(g/ml)"].values.reshape(-1, 1)
    confidence = pseudo_df["confidence"].values

    features = np.where(np.isinf(features), np.nan, features)
    target = np.where(np.isinf(target), np.nan, target)
    valid = ~np.isnan(features).any(axis=1) & ~np.isnan(target).flatten()
    features = features[valid]
    target = target[valid]
    confidence = confidence[valid]

    scaler_x = StandardScaler().fit(features)
    scaler_y = StandardScaler().fit(target)
    X = torch.tensor(scaler_x.transform(features), dtype=torch.float32)
    y = torch.tensor(scaler_y.transform(target), dtype=torch.float32)
    conf_t = torch.tensor(confidence, dtype=torch.float32)

    # Apply 8:1:1 train-test-validation split
    X_tr, X_tmp, y_tr, y_tmp, c_tr, c_tmp = train_test_split(X, y, conf_t, test_size=0.2, random_state=42, shuf
    X_va, X_te, y_va, y_te, c_va, c_te = train_test_split(X_tmp, y_tmp, c_tmp, test_size=0.5, random_state=42,

    train_ds = TensorDataset(X_tr, y_tr, c_tr)
    val_ds = TensorDataset(X_va, y_va, c_va)
    test_ds = TensorDataset(X_te, y_te, c_te)

    return (
        DataLoader(train_ds, batch_size=64, shuffle=True),
        DataLoader(val_ds, batch_size=64, shuffle=False),
        DataLoader(test_ds, batch_size=64, shuffle=False),
    )
```

Figure 6.2.2.1: Code Snippet for PseudoLabel Generation and DataLoader Creation

Due to the lack of dissolved oxygen value as ground truth, before the clients start to communicate with server in the federated learning. A produced dissolved oxygen used by a pretrained model will serve as the ground truth based on the current sensor parameters in the clients' sqlite database. Then, compute per-sample confidence based on local variance of pseudo predictions, clipped to [0.5, 0.95]. This confidence is used in training later to ensure reliable pseudo-labels influence training more and also mitigates noise from pseudo-labels. After that, the ground truth will combine with the five parameters and form a new dataloader which is prepared for federated learning.

Then, select features and pseudo target, remove any non-number value and infinity. standardize inputs and target with StandardScaler. After that, split the data into train/val/test = 80%/10%/10%. and create dataloader.

6.2.3 Test Plans for Federated Learning and Results

6.2.3.1 Federated Learning (GRPC) Test

Test Description:

This test is to validate functional FL rounds over gRPC

Actions:

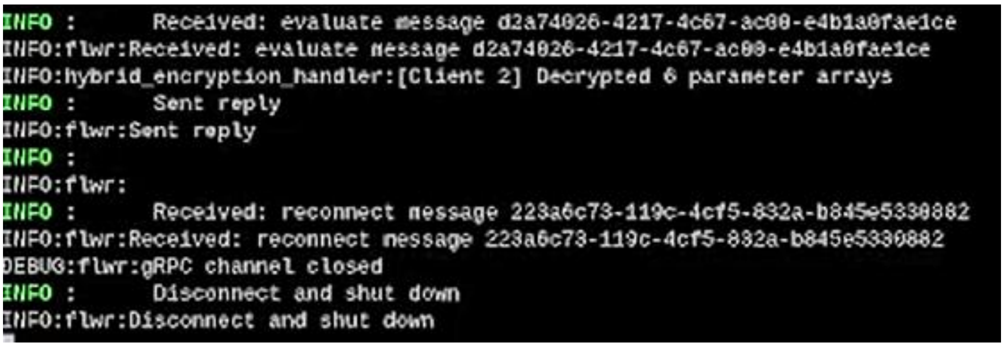
Run multiple federated rounds with distributed clients

Expected Output:

No communication errors and latencies logged.

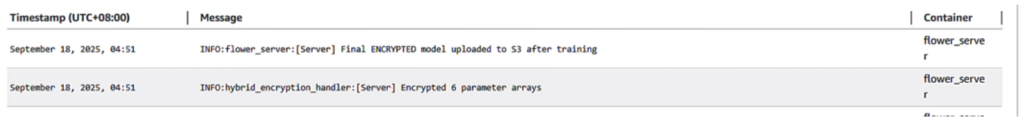
Actual Output:

Clients successfully complete federated learning with Grpc channel with no communication error



```
INFO :      Received: evaluate message d2a74026-4217-4c67-ac00-e4b1a0fae1ce
INFO:flwr:Received: evaluate message d2a74026-4217-4c67-ac00-e4b1a0fae1ce
INFO:hybrid_encryption_handler:[Client 2] Decrypted 6 parameter arrays
INFO :      Sent reply
INFO:flwr:Sent reply
INFO :
INFO:flwr:
INFO :      Received: reconnect message 223a6c73-119c-4cf5-832a-b845e5330882
INFO:flwr:Received: reconnect message 223a6c73-119c-4cf5-832a-b845e5330882
DEBUG:flwr:gRPC channel closed
INFO :      Disconnect and shut down
INFO:flwr:Disconnect and shut down
```

Server successfully complete federated learning with with Grpc channel with no communication error



Timestamp (UTC+08:00)	Message	Container
September 18, 2025, 04:51	INFO:flower_server:[Server] Final ENCRYPTED model uploaded to S3 after training	flower_server
September 18, 2025, 04:51	INFO:hybrid_encryption_handler:[Server] Encrypted 6 parameter arrays	flower_server

Pass/Fail:

Pass

6.2.3.2 Scalability Test

Test Description:

This test is to test the scalability of the federated learning across multiple machines in different locations.

Actions:

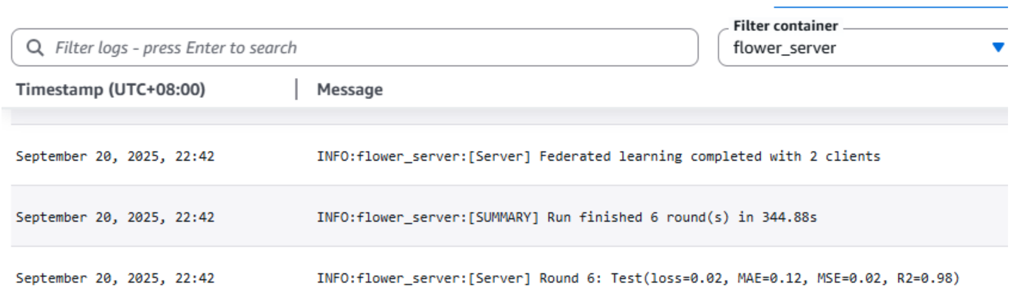
1. Run Federated Learning with Two Clients: modify environment variables at task definition at server side:
    a. MIN_AVAILABLE_CLIENTS=2
    b. MIN_FIT_CLIENTS=2
    c. MIN_EVALUATE_CLIENTS=2
2. Run Federated Learning with Three Clients modify environment variables at task definition at server side:
    a. MIN_AVAILABLE_CLIENTS=3
    b. MIN_FIT_CLIENTS=3
    c. MIN_EVALUATE_CLIENTS=3

Expected Output:

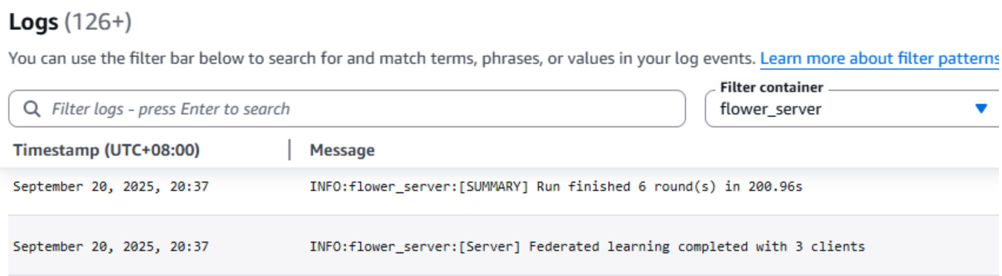
1. Able to complete federated learning with two clients
2. Able to complete federated learning with three clients

Actual Output:

Server successfully complete federated learning with two clients



Server successfully complete federated learning with three clients



Pass/Fail:

Pass

6.2.3.3 Start Federated Learning Server (Auto/Manual)

Test Description:

This test is to validate boot behaviour of the server

Actions:

Manual: Change desired task in service to 1

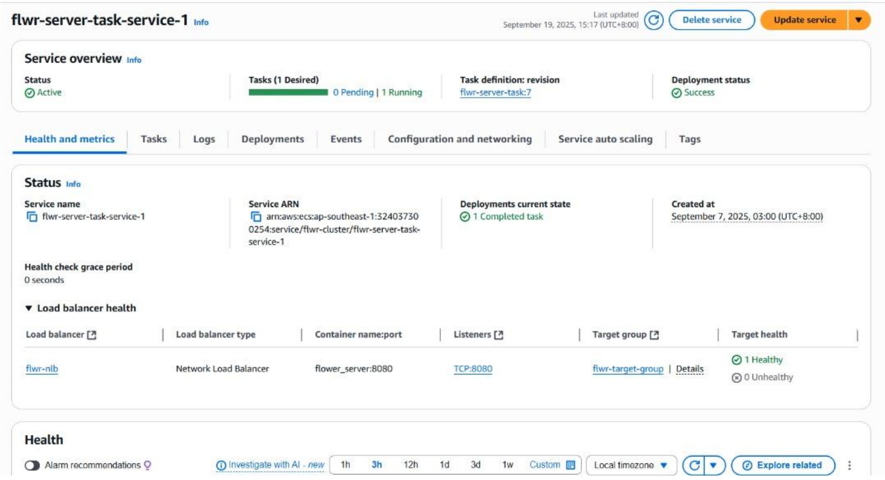
Auto: EventBridge Scheduler schedule to start

Expected Output:

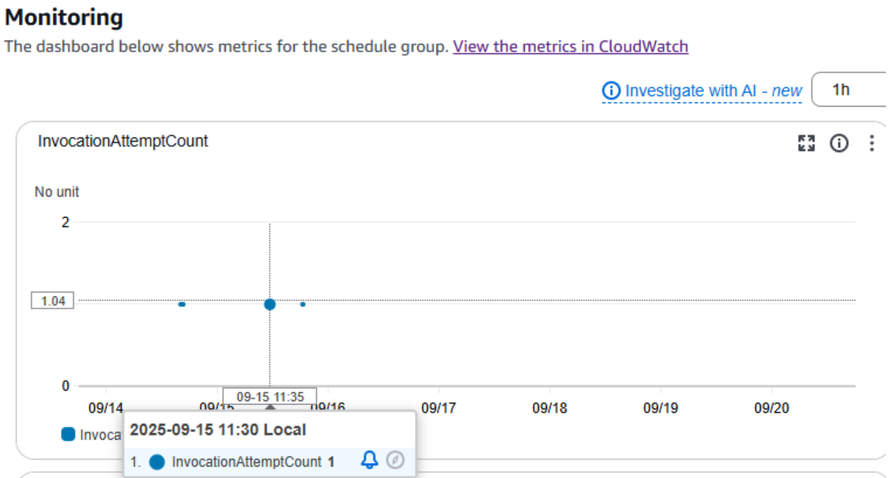
Able to start the server by manual or by schedule

Actual Output:

Manual: Server start running task when desired task is manually set to 1.



Auto: EventBridge Schedule invoke the flwr-biweekly-start schedule to start the server automatically.



Pass/Fail:

Pass

6.2.3.4 Stop Server (Auto/Manual)

Test Description:

This test is to test whether the server can shutdown in schedule

Actions:

Manual: Change desired task in service to 0

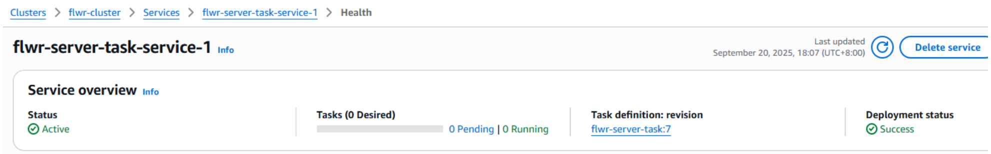
Auto: EventBridge Scheduler schedule to stop

Expected Output:

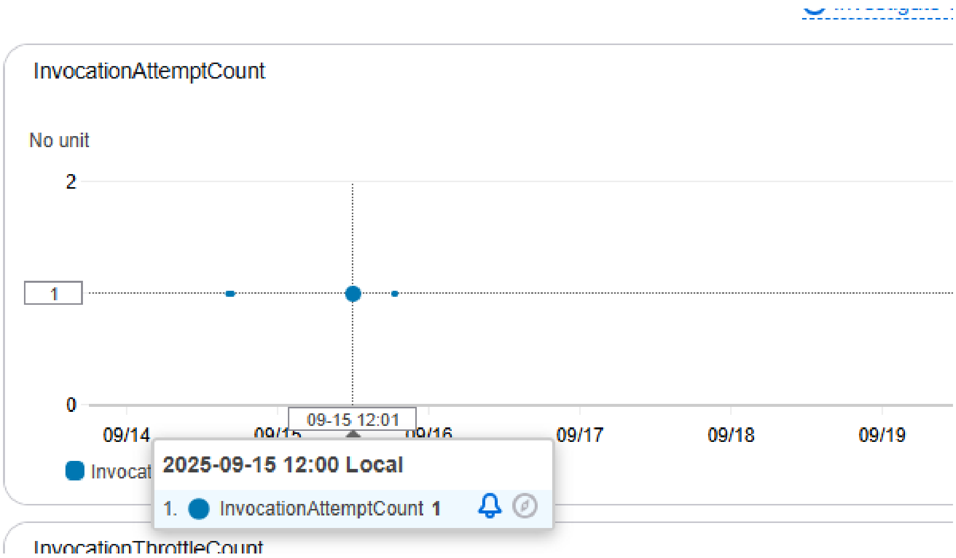
Able to stop the server by manual or by schedule

Actual Output:

Manual: Server stop running task when desired count of task is manually set to 0.



Auto: EventBridge Schedule invoke the flwr-biweekly-stop schedule to stop the server automatically.



Pass/Fail:

Pass

6.2.3.5 Shared Context Generation

Test Description:

To Validate CKKS shared-context generation and distribution

Actions:

Delete the current shared context in s3

Expected Output:

When no shared context in s3, server generates shared context, uploads to S3, clients download and load.

Actual Output:

Server successfully generated new shared context and uploaded it to S3.

September 20, 2025, 22:36	INFO:hybrid_encryption_handler:Uploaded new shared context to S3
September 20, 2025, 22:36	INFO:hybrid_encryption_handler:[Server] Hybrid encryption initialized successfully
September 20, 2025, 22:36	INFO:hybrid_encryption_handler:[Server] Generating new shared context
September 20, 2025, 22:36	INFO:flower_server:[Server] Starting Flower FedProx server

Pass/Fail:

Pass

6.2.3.6 Global Model Initialization

Test Description:

This is to test whether server able to initialize global model when no encrypted global model in s3

Actions:

Delete the current encrypted global model in s3

Expected Output:

When no encrypted global model in s3, server generates new global model

Actual Output:

Server successfully initialized a new model

September 20, 2025, 22:36	INFO:flower_server:[Server] Using num_rounds=6
September 20, 2025, 22:36	INFO:flower_server:[Server] No existing encrypted model in S3; starting fresh
September 20, 2025, 22:36	INFO:hybrid_encryption_handler:Uploaded new shared context to S3

Pass/Fail:

Pass

6.2.3.7 Server Perform Aggregation Test

Test Description:

This is to test whether server able to perform aggregation in the federated learning

Actions:

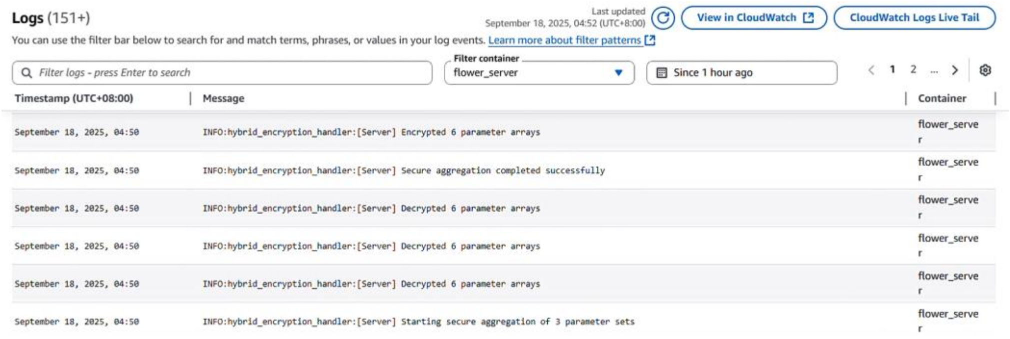
Make sure more than one client involved in the federated learning

Expected Output:

Server able to perform aggregation

Actual Output:

Server able to perform aggregation



Pass/Fail:

Pass

6.2.3.8 S3 Upload Test

Test Description:

To validate whether server able to upload shared context after generation and upload encrypted global model after federated learning

Actions:

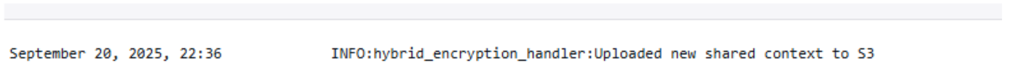
Run federated Learning

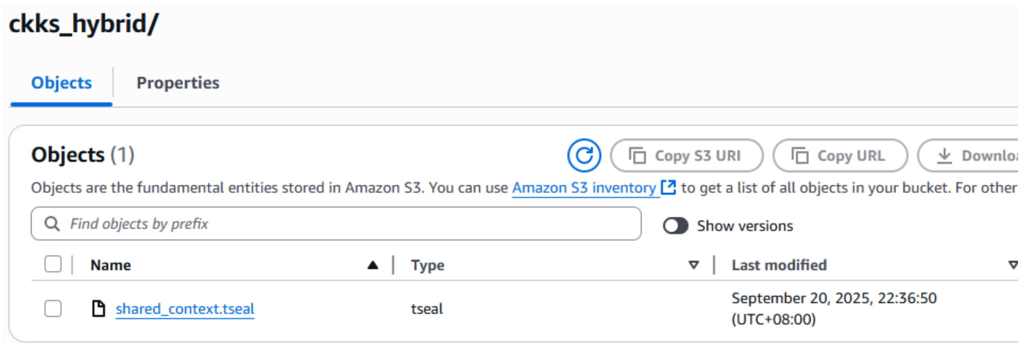
Expected Output:

Server able to upload shared context after generation and upload encrypted global model after federated learning

Actual Output:

1. Server able to upload shared context to S3 after key generation





2. Server able to upload encrypted global model after federated learning

Pass/Fail:

Pass

6.2.3.9 Encryption/Decryption (Server and Client)

Test Description:

This test is to test the ability of clients and server to handle the encryption and decryption in the federated learning.

Actions:

Make sure has shared context in the S3

Expected Output:

1. At the start of the federated learning, when the current shared context matches with the encrypted global model in the S3.

Client: Able to decrypt existing global model and load into local model

Server: Able to decrypt existing global model and load into global model

2. When federated learning round,

Client: Able to decrypt received latest encrypted global model and update local model and encrypt back after local training

Server: Able to decrypt received encrypted local model from clients and encrypt for distribution

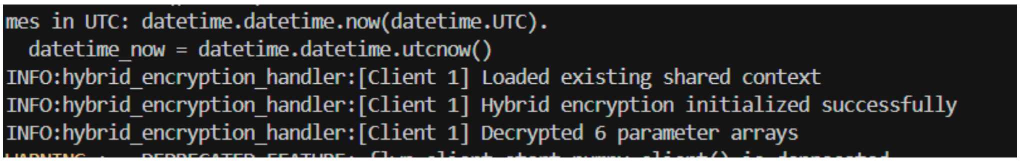
3. **At the start of the federated learning, when the current shared context is not matched with the encrypted global model in the S3.**

Server: Unable to decrypt received encrypted local model form clients and encrypt for distribution and task stop automatically.

Actual Output:

1. **At the start of the federated learning, when the current shared context matches with the encrypted global model in the S3.**

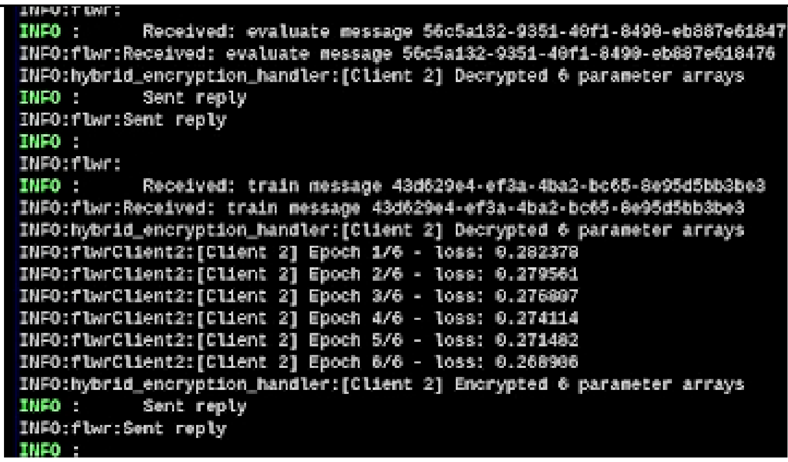
Client: Able to decrypt existing encrypted global model and load into local model



Server: Able to decrypt existing encrypted global model and load into global model

2. **When federated learning,**

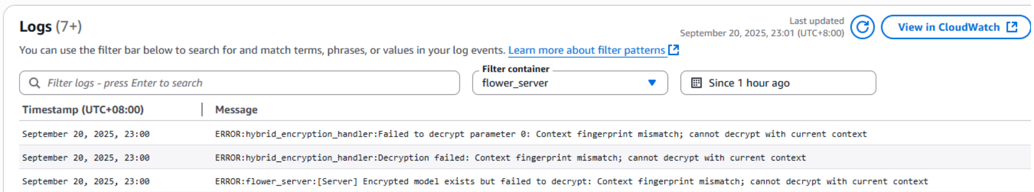
Client: Able to decrypt received latest encrypted global model and update local model and encrypt back after local training



Server: Able to decrypt received encrypted local model from clients and encrypt for distribution

3. **At the start of the federated learning, when the current shared context is not matched with the encrypted global model in the S3.**

Server: Unable to decrypt received encrypted local model form clients and encrypt for distribution and task stop automatically.



Pass/Fail:

Pass

6.2.3.10 Data Synchronization (Cloud side) Pipeline Test

Test Description:

This test is to test whether the pipeline is able to process the sync data from client and put it into DynamoDB

Actions:

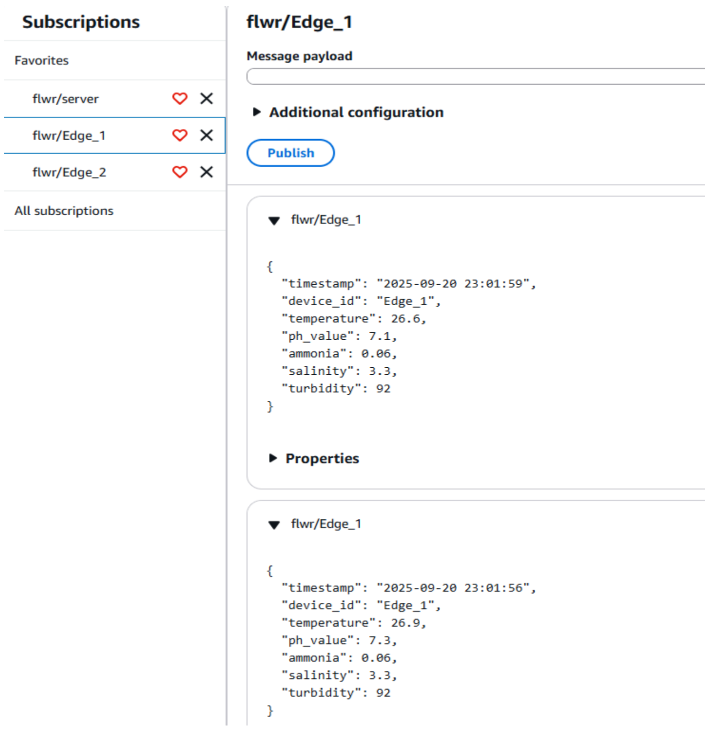
Publish data in JSON format to the MQTT Topic

Expected Output:

Cloud is able to process the sync data from clients and put into DynamoDB

Actual Output:

1. Cloud subscribed to the MQTT topic successfully received the client sensor data and received published data from edge client 1



2. The sensor data is then directed into the DynamoDB automatically.

Pass/Fail:

Pass

6.2.3.11 Improvement in Learning Performance Test

Test Description:

This test is to test whether the performance improves in the federated learning like low latency, high convergence and federated learning speed and low train loss across different ponds with heterogenous data.

Actions:

Perform full federated learning with three raspberry pi as clients and run for 6 rounds and 6 local epochs

Expected Output:

No latency issue and federated learning happen fast and training and testing loss drops effectively round after round

Actual Output:

1. The convergence speed is fast as shown in table 6.2.3.12.1. The training loss and testing loss drop a lot at the start.

2. Fast Training. The duration of full federated learning with three real raspberry pi clients is only 1 minute.

- The start time count from server start to decrypt the parameters from clients which is 4.50am

September 18, 2025, 04:50	INFO:flower_server:[Server] Round 2: Aggregated val MSE: 0.195246	flower_serve r
September 18, 2025, 04:50	INFO:flower_server:[Server] Round 2: Aggregated val R^2: 0.747389	flower_serve r
September 18, 2025, 04:50	INFO:hybrid_encryption_handler:[Server] Decrypted 6 parameter arrays	flower_serve r
September 18, 2025, 04:50	INFO:hybrid_encryption_handler:[Server] Encrypted 6 parameter arrays	flower_serve r
September 18, 2025, 04:50	INFO:hybrid_encryption_handler:[Server] Secure aggregation completed successfully	flower_serve r
September 18, 2025, 04:50	INFO:hybrid_encryption_handler:[Server] Decrypted 6 parameter arrays	flower_serve r
September 18, 2025, 04:50	INFO:hybrid_encryption_handler:[Server] Decrypted 6 parameter arrays	flower_serve r
September 18, 2025, 04:50	INFO:hybrid_encryption_handler:[Server] Decrypted 6 parameter arrays	flower_serve r

- The end time is the time when server uploaded the final encrypted global model to the global which is 4.51am

Timestamp (UTC+08:00)	Message	Container
September 18, 2025, 04:51	INFO:flower_server:[Server] Final ENCRYPTED model uploaded to S3 after training	flower_serve r
September 18, 2025, 04:51	INFO:hybrid_encryption_handler:[Server] Encrypted 6 parameter arrays	flower_serve r

Pass/Fail:

Pass

6.2.3.12 Improvement in Predictive Performance Test

Test Description: This test is to test whether the predictive performance improves in the federated learning even across different ponds with heterogenous data.
Actions: Perform full federated learning with three raspberry pi as clients and run for 6 rounds and 6 local epochs
Expected Output: The validation and test R^2 value increases round after round
Actual Output: The R^2 value increases round after round as shown in the table and increases from 0.58 to 0.64 only after 6 rounds of federated learning and 6 local epochs running on clients. And the gap between the validation R^2 and test R^2 shows that it is not overfitting and will grow in the future federated learning round.
Pass/Fail: Pass

Rounds	Performance Metrics
1	INFO:flower_server:[Server] Round 1: Aggregated train loss: 0.146051 INFO:flower_server:[Server] Round 1: Aggregated val loss: 0.218050 INFO:flower_server:[Server] Round 1: Aggregated val MAE: 0.363305 INFO:flower_server:[Server] Round 1: Aggregated val MSE: 0.218050 INFO:flower_server:[Server] Round 1: Aggregated val R^2: 0.722558 INFO:hybrid_encryption_handler:[Server] Encrypted 6 parameter arrays INFO:flower_server:[Server] Round 1: Encrypted parameters for evaluation INFO:flower_server:[Server] Round 1: Test loss: 0.424766 INFO:flower_server:[Server] Round 1: Test MAE: 0.549223 INFO:flower_server:[Server] Round 1: Test MSE: 0.424766 INFO:flower_server:[Server] Round 1: Test R^2: 0.586158
2	INFO:flower_server:[Server] Round 2: Aggregated train loss: 0.123672 INFO:flower_server:[Server] Round 2: Aggregated val loss: 0.195246 INFO:flower_server:[Server] Round 2: Aggregated val MAE: 0.339170 INFO:flower_server:[Server] Round 2: Aggregated val MSE: 0.195246 INFO:flower_server:[Server] Round 2: Aggregated val R^2: 0.747389 INFO:hybrid_encryption_handler:[Server] Encrypted 6 parameter arrays INFO:flower_server:[Server] Round 2: Encrypted parameters for evaluation INFO:flower_server:[Server] Round 2: Test loss: 0.386480 INFO:flower_server:[Server] Round 2: Test MAE: 0.515892 INFO:flower_server:[Server] Round 2: Test MSE: 0.386480 INFO:flower_server:[Server] Round 2: Test R^2: 0.619039
3	INFO:flower_server:[Server] Round 3: Aggregated train loss: 0.109024 INFO:flower_server:[Server] Round 3: Aggregated val loss: 0.185807 INFO:flower_server:[Server] Round 3: Aggregated val MAE: 0.337624 INFO:flower_server:[Server] Round 3: Aggregated val MSE: 0.185807 INFO:flower_server:[Server] Round 3: Aggregated val R^2: 0.758678 INFO:hybrid_encryption_handler:[Server] Encrypted 6 parameter arrays INFO:flower_server:[Server] Round 3: Encrypted parameters for evaluation INFO:flower_server:[Server] Round 3: Test loss: 0.372464 INFO:flower_server:[Server] Round 3: Test MAE: 0.504142 INFO:flower_server:[Server] Round 3: Test MSE: 0.372464 INFO:flower_server:[Server] Round 3: Test R^2: 0.630882

4	INFO:flower_server:[Server] Round 4: Aggregated train loss: 0.100470 INFO:flower_server:[Server] Round 4: Aggregated val loss: 0.175817 INFO:flower_server:[Server] Round 4: Aggregated val MAE: 0.315038 INFO:flower_server:[Server] Round 4: Aggregated val MSE: 0.175817 INFO:flower_server:[Server] Round 4: Aggregated val R^2: 0.770040 INFO:hybrid_encryption_handler:[Server] Encrypted 6 parameter arrays INFO:flower_server:[Server] Round 4: Encrypted parameters for evaluation INFO:flower_server:[Server] Round 4: Test loss: 0.359458 INFO:flower_server:[Server] Round 4: Test MAE: 0.485200 INFO:flower_server:[Server] Round 4: Test MSE: 0.359458 INFO:flower_server:[Server] Round 4: Test R^2: 0.641008
5	INFO:flower_server:[Server] Round 5: Aggregated train loss: 0.094586 INFO:flower_server:[Server] Round 5: Aggregated val loss: 0.174604 INFO:flower_server:[Server] Round 5: Aggregated val MAE: 0.317268 INFO:flower_server:[Server] Round 5: Aggregated val MSE: 0.174604 INFO:flower_server:[Server] Round 5: Aggregated val R^2: 0.771596 INFO:hybrid_encryption_handler:[Server] Encrypted 6 parameter arrays INFO:flower_server:[Server] Round 5: Encrypted parameters for evaluation INFO:flower_server:[Server] Round 5: Test loss: 0.358213 INFO:flower_server:[Server] Round 5: Test MAE: 0.483188 INFO:flower_server:[Server] Round 5: Test MSE: 0.358213 INFO:flower_server:[Server] Round 5: Test R^2: 0.641832
6	INFO:flower_server:[Server] Round 6: Aggregated train loss: 0.092853 INFO:flower_server:[Server] Round 6: Aggregated val loss: 0.172058 INFO:flower_server:[Server] Round 6: Aggregated val MAE: 0.314566 INFO:flower_server:[Server] Round 6: Aggregated val MSE: 0.172058 INFO:flower_server:[Server] Round 6: Aggregated val R^2: 0.774445 INFO:hybrid_encryption_handler:[Server] Encrypted 6 parameter arrays INFO:flower_server:[Server] Round 6: Encrypted parameters for evaluation INFO:flower_server:[Server] Round 6: Test loss: 0.356801 INFO:flower_server:[Server] Round 6: Test MAE: 0.480784 INFO:flower_server:[Server] Round 6: Test MSE: 0.356801 INFO:flower_server:[Server] Round 6: Test R^2: 0.642814

Table 6.2.3.12.1 Performance Metrics in the Federated Learning Test

6.3 Project Challenges

This project manages to illustrate a functioning server-side federated learning (FL) prototype in an edge-cloud setup, and a number of practical challenges were faced in the process of implementation and evaluation.

Firstly, unavailability of ground-truth dissolved oxygen sensor data. The system took the pseudo-labels which is the output of the pretrained models as the DO ground truth on the clients, thus restricting the absolute accuracy and fidelity of the learned predictor. This is clearly mentioned as the main limitation, and it has direct impact on validation of models.

Besides, trade-offs of encryption and assurances of trust is the second challenge. Shared-context CKKS hybrid scheme was adopted in order to encrypt parameter exchanges and still be computationally viable on Raspberry Pi platforms. This makes less overhead than multi-key HE, but has a trusted-server assumption and open risks in case the server is compromised. Multi-key or full homomorphic HE was not applied because it was too complex and too expensive to perform and it is not PyTorch compatibility.

Third, periodic connectivity. Although a lightweight deep neural network and encryption scheme was deployed to lower the size of model transmission, it is difficult to ensure timely round attendance during FL particularly in the case of many distributed farms (geographically) far apart. In addition, periodic connectivity will also influence the data synchronization from edge to cloud even having the robust data synchronization technique.

Lastly, sensors variance and data quality. When there is noise in sensors, sensor drift, and heterogeneity between hardware could also adversely affect the performance of the model and add biases during federated updates.

6.4 Objective Evaluation

The project objective in section 1.2 is evaluated against the implementation and test results.

For the first objective: Enhance and deploy server-side federated learning in an edge-cloud framework for broader applicability, it is achieved. The containerized flower server hosted in the AWS Fargate was successfully to coordinate federated learning rounds with two Raspberry pi clients across different locations then increases to three clients for testing. It shows that learning is scalable as long as having enough devices. Besides, the results show predictive improvement as shown in R^2 metrics in short runs have obviously improved after six rounds of federated learning, which indicates a good convergence and learning across heterogeneous ponds by the FedProx algorithm and implemented deep learning neural network.

For the second objective: refine and implement a flexible, robust data synchronization mechanism for intermittent networks, it is achieved functionally. The implemented pipeline that sync data through MQTT Protocol, Lambda to DynamoDB for online and local sqlite database for offline storage, allows continued storage and upload data in the intermittent network without data loss. Other than that, the light weight machine learning model and the encryption technique also assist in data synchronization in the intermittent network as it reduces the size of model transmission. The transmission will only need less bandwidth and low latency even in intermittent network.

For the third objective: improve and implement a secure, efficient, privacy-preserving edge-cloud framework, it is achieved. The federated learning is conducted without exposing the clients' raw sensor data and only involve model parameter exchange. To further protect the federated learning, shared-context CKKS hybrid encryption is adopted to protect the FL model parameters from being exposed or attacked in the transmission to and from server and clients. It also performs round-trip test to verify each sent and received encrypted model parameters. So, the federated learning is running securely and efficiently. However, the solution sacrifices some trust (shared key to trusted server) to be deployed in a resource-constrained client.

Lastly, for the fourth objective: maintain cost-effectiveness and accessibility for small-scale farmers, is achieved. The architecture uses cost-efficient AWS building blocks to build the server which and run on demand and lightweight Shared Context CKKS Hybrid Encryption.

6.5 Concluding Remark

This project provided a practical and privacy conscious server-side federated learning prototype of precision aquaculture which incorporates FedProx aggregation, shared-context CKKS hybrid encryption scheme, and MQTT to Lambda to DynamoDB synchronization pipeline. The system shows that the encrypted federated learning across raspberry pi edge devices and server hosted in AWS Cloud is feasible as federated learning test completed successfully with high performance and integrating encryption approach.

However, to move to production, the important things to take note is a real dissolved oxygen must be considered to purchase to produce the real dissolved oxygen. This can improve the model reliability. Second, also consider to enhance the encryption technique to apply homomorphic encryption but ensure the clients have high processing power to handle the large computation.

CHAPTER 7
Conclusion and Recommendation

7.1 Conclusion

To conclude, this project successfully developed and implemented a server-side federated learning framework with high scalability and security using an edge–cloud architecture which directly addresses the major challenges faced by small-scale farmers or pond owners seeking to adopt precision aquaculture technologies. These challenges include high costs, technical barriers, unreliable internet connectivity and data privacy concerns.

The design for the system architecture integrates numerous elements that run as a framework with Raspberry Pi devices at the edge utilizing AWS IoT Greengrass, a server in the cloud managed by AWS ECS with Fargate, and Amazon S3 for model and key management. To locally manage the differences in data distribution among the various fishponds, the FedProx algorithm was used, while the CKKS-based hybrid encryption was implemented to keep the original model parameters at edges with only encrypted model updates being shared. Besides that, a strong data synchronization pipeline was set up to resolve occasionally interrupted network connections in the rural farm and model training was planned on a two-weekly basis so as to balance the freshness of the model, communication costs, and energy consumption. The very first findings gave the green light to the system's practicality, revealing efficient gRPC-based communication, the achievement of the model with gradually declining loss metrics, as well as the production and handling of the encryption key.

In addition to such technical achievements, the project has its major implications in the aquaculture industry. It promotes the economic growth of the entire Malaysia as it allows advanced technologies and privacy-protecting models that are affordable to small-scale farmers. The privacy protection strategy also enhances the eagerness of farmers to embrace digital solutions, even though the scalable edge cloud system is a blueprint on such deployments in other resource-limiting aquaculture or agriculture environments.

Altogether, the project not only proves that federated learning within an edge - cloud framework is feasible but also indicates that these systems can break major obstacles to the adoption of precision aquaculture. The system enables the small-scale farmers to enjoy modern predictive and monitoring features and protect control over sensitive farm data by solving cost, connectivity, privacy issues, and technical complexity challenges.

7.2 Recommendation

The following recommendations are proposed to outline the direction for future research and development of this framework, under the pillars of Strengths, Weaknesses, Opportunities, and Threats (SWOT).

In terms of strengths, this system significantly enhanced the privacy and security using the lightweight Shared-Context CKKS Hybrid Encryption scheme, the future development is suggested to continuously benchmark the CKKS scheme to maintain optimal performance and privacy for resource-constrained edge devices. Moreover, the system achieved high stability through the implementation of FredProx algorithm, future research can explore adaptive strategies for the FredProx algorithm regularisation parameter, allowing the system to dynamically optimise convergence across diverse farm environments. The proposed scalable architecture relying on cost-effective AWS services is recommended to be promote as a blueprint for rapid deployment in other small-scale IoT agricultural applications due to the high resilience and network stability.

On the other hand, the current weaknesses impacting model accuracy and system functionality shall also be resolved in the future development. For instance, the current model comprehensiveness, it is recommended to implement more sophisticated Machine Learning models capable of processing advanced data to enable behavioural analysis of fish or prawns and enhance the overall model performance. Furthermore, the current system suffers from the absence of a dissolved oxygen sensor for collecting ground truth data. The recommendation is to prioritise the integration of specialised dissolved oxygen sensors to ensure the predictive model is trained on reliable labels and significantly improving overall model efficacy. Additionally, due to the technical difficulty and the chance to degrade the model performance, full homomorphic encryption is not implemented in the system. Future work may emphasise on this to solve the issue of model accuracy performance and the computation overhead.

In terms of opportunities, there are several areas for expansion and technological growth aligning with the industry needs. First, it is recommended to expand the model scope to develop advanced predictive capabilities utilising computer vision data for real-time disease detection and growth monitoring, it helps to provide a more comprehensive understanding of the aquacultural conditions. This might require developing specific federated learning workflows focusing on maximising information extraction while minimising communication overhead. In addition, this proposed architecture can facilitate broader FL deployment across different agricultural sectors that are facing the similar data privacy and connectivity issues, it is recommended that the whole system should be packaged into standardised and easily deployable modules to facilitate the adoption in other small farming initiatives.

To ensure long-term feasibility of the system, potential risks should be addressed proactively. For example, the issue of sensor variability and drift can impact the global model's stability, it is recommended to be mitigated by sensor calibration algorithms at the edge level to adjust local models and improve the global model's resilience to data quality variances. Lastly, it is strongly suggested to be contend with the evolving regulatory landscape for data privacy. It is important to follow regulatory changes and integrate Secure Aggregation techniques together with the encryption to enhance security guarantees against potential malicious server compromises, thereby addressing the current "Trusted Server" limitation of the Shared-Context CKKS approach.

REFERENCES

[1] Jumatli, A., & Ismail, M. S. (2021). Promotion of sustainable aquaculture in Malaysia. In Proceedings of the International Workshop on the Promotion of Sustainable Aquaculture, Aquatic Animal Health, and Resource Enhancement in Southeast Asia (pp. 31-40). Aquaculture Department, Southeast Asian Fisheries Development Center.

[2] S. Sah, R. K. Yadav, S. W. Shah, B. Vijayan, and Naheem, "IoT-Based Smart Fish Farming Aquaculture Monitoring System." https://www.semanticscholar.org/paper/IOT-BASED-SMART-FISH-FARMING-AQUACULTURE-MONITORING-Sah-Yadav/f27dba7532b5a670f463245e4b95c70f218ecfe1

[3] M.-C. Chiu, W.-M. Yan, S. A. Bhat, and N.-F. Huang, "Development of smart aquaculture farm management system using IoT and AI-based surrogate models," Journal of Agriculture and Food Research, vol. 9, p. 100357, Sep. 2022, doi: 10.1016/j.jafr.2022.100357.

[4] W. K. Cheng, J. C. Khor, W. Z. Liew, K. T. Bea, and Y. L. Chen, "Integration of federated learning and Edge-Cloud platform for precision aquaculture," IEEE Access, p. 1, Jan. 2024, doi: 10.1109/access.2024.3454057.

[5] Godwin Idoje, Tasos Dagiuklas, and Iqbal Muddesar, "Federated Learning: Crop classification in a smart farm decentralised network," Smart agricultural technology, vol. 5, pp. 100277–100277, Oct. 2023, doi: https://doi.org/10.1016/j.atech.2023.100277.

[6] X. Yao, T. Huang, R.-X. Zhang, R. Li, and L. Sun, "Federated Learning with Unbiased Gradient Aggregation and Controllable Meta Updating," *arXiv.org*, 2019. https://arxiv.org/abs/1910.08234 (accessed Apr. 28, 2025).

[7] K. Wang, Z. Ding, D. K. C. So, and Z. Ding, "Energy Efficient Federated Learning with Age-Weighted FedSGD," *2024 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 457–462, Jun. 2024, doi: https://doi.org/10.1109/iccworkshops59551.2024.10615715.

[8] L. Collins, H. Hassani, A. Mokhtari, and S. Shakkottai, “FedAvg with Fine Tuning: Local Updates Lead to Representation Learning,” *arXiv:2205.13692 [cs]*, May 2022, Accessed: Dec. 02, 2022. [Online]. Available: https://arxiv.org/abs/2205.13692

[9] T.-T. Ho, K.-D. Tran, and Y. Huang, “FedSGDCOVID: Federated SGD COVID-19 Detection under Local Differential Privacy Using Chest X-ray Images and Symptom Information,” *Sensors*, vol. 22, no. 10, pp. 3728–3728, May 2022, doi: https://doi.org/10.3390/s22103728.

[10] X.-T. Yuan and P. Li, “On Convergence of FedProx: Local Dissimilarity Invariant Bounds, Non-smoothness and Beyond,” *arXiv.org*, 2022. https://arxiv.org/abs/2206.05187 (accessed Apr. 28, 2025).

[11] S. Truex *et al.*, “A Hybrid Approach to Privacy-Preserving Federated Learning,” *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security - AISec’19*, 2019, doi: https://doi.org/10.1145/3338501.3357370.

[12] R. C. Geyer, T. Klein, and M. Nabi, “Differentially Private Federated Learning: A Client Level Perspective,” *arXiv:1712.07557 [cs, stat]*, Mar. 2018, Available: https://arxiv.org/abs/1712.07557

[13] K. Bonawitz *et al.*, “Practical Secure Aggregation for Federated Learning on User-Held Data,” *arXiv.org*, 2016. https://arxiv.org/abs/1611.04482 (accessed Apr. 28, 2025).

[14] T. Ryffel *et al.*, “A generic framework for privacy preserving deep learning,” *arXiv:1811.04017 [cs, stat]*, Nov. 2018, Accessed: Mar. 10, 2022. [Online]. Available: https://arxiv.org/abs/1811.04017

[15] X. Yin, Y. Zhu, and J. Hu, “A Comprehensive Survey of Privacy-preserving Federated Learning,” *ACM Computing Surveys*, vol. 54, no. 6, pp. 1–36, Jul. 2021, doi: https://doi.org/10.1145/3460427.

[16] Y. Chen, L. Wang, and X. Zhou, "Federated Learning for Smart Aquaculture: Opportunities and Challenges," in *IEEE Internet of Things Journal*, vol. 9, no. 16, pp. 14321–14332, Aug. 2022, doi: 10.1109/JIOT.2022.3164567.

[17] H. Zhao, J. Liu, and M. Sun, "Efficient Federated Learning for Environmental Monitoring," in *IEEE Transactions on Green Communications and Networking*, vol. 7, no. 1, pp. 119–130, Mar. 2023, doi: 10.1109/TGCN.2023.3245678.

[18] R. Singh, A. Verma, and P. Sharma, "Federated Edge Learning in Aquaculture IoT: A Case Study," in *Proceedings of the 2024 IEEE International Conference on Communications (ICC)*, Denver, CO, USA, Jun. 2024, pp. 4560–4565, doi: 10.1109/ICC42651.2024.9865432.

[19] "Overview of Amazon Web Services," 2017. Accessed: Apr. 30, 2025. [Online]. Available: https://d1.awsstatic.com/whitepapers/aws-overview.ac197f9e2dee426af6d88f85e6bae2f7c66a6b7f.pdf

[20] D. Zhao, "Skefl: Single-Key Homomorphic Encryption for Secure Federated Learning," *arXiv (Cornell University)*, Jan. 2022, doi: https://doi.org/10.48550/arxiv.2212.11394.

[21] T. Thesing, C. Feldmann, and M. Burchardt, "Agile versus Waterfall Project management: Decision Model for Selecting the Appropriate Approach to a Project," *Procedia Computer Science*, vol. 181, no. 1, pp. 746–756, 2021, doi: https://doi.org/10.1016/j.procs.2021.01.227.

[22] A. Benaïssa, B. Retiât, B. Cebere, and A. E. Belfedhal, "TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption," *arXiv preprint arXiv:2104.03152*, 2021.

[23] A. Kumar, "MSE vs RMSE vs MAE vs MAPE vs R-Squared: When to Use?," *Analytics Yogi*, Aug. 18, 2024. https://vitalflux.com/mse-vs-rmse-vs-mae-vs-mape-vs-r-squared-when-to-use/

[24] Z. Hao, X. Zhang, Y. Liu, H. Xu, and J. Liu, "A dissolved oxygen prediction model based on GRU–N-Beats," *Frontiers in Marine Science*, vol. 11, p. 1350300, May 2024. doi: 10.3389/fmars.2024.1350300

[25] why, "Mathematics Stack Exchange," *Mathematics Stack Exchange*, 2017. https://math.stackexchange.com/questions/2398194/why-is-r-square-not-well-defined-for-a-regression-without-a-constant-term

APPENDIX

Appendix A

	A	B	C	D	E	F	G	H	I
1	created_at	entry_id	temperature	ph_value	ammonia	salinity	turbidity	Dissolved Oxygen(mg/L)	Nitrate(g/ml)
2	2021-06-19 00:00:05 CET	1889	24.875	8.43365	0.45842	4.270505	100	4.505	193
3	2021-06-19 00:01:02 CET	1890	24.9375	8.43818	0.45842	3.82807	100	6.601	194
4	2021-06-19 00:01:22 CET	1891	24.875	8.42457	0.45842	3.986776	100	15.797	192
5	2021-06-19 00:01:44 CET	1892	24.9375	8.43365	0.45842	4.479721	100	5.046	193
6	2021-06-19 00:02:07 CET	1893	24.9375	8.40641	0.45842	2.381299	100	38.407	192
7	2021-06-19 00:02:27 CET	1894	24.9375	8.42003	0.45842	3.85126	100	3.862	193
8	2021-06-19 00:02:47 CET	1895	24.875	8.43818	0.45842	4.279328	100	2.831	194
9	2021-06-19 00:03:07 CET	1896	24.9375	8.42911	0.45842	4.103936	100	5.012	193
10	2021-06-19 00:03:27 CET	1897	24.9375	8.42911	0.45842	4.271254	100	2.916	192
11	2021-06-19 00:03:47 CET	1898	24.875	8.43365	0.45842	3.46107	100	17.005	192
12	2021-06-19 00:04:31 CET	1900	24.875	8.48358	0.45842	4.352004	100	6.964	191
13	2021-06-19 00:05:11 CET	1902	24.9375	8.42911	0.45842	4.489336	100	3.465	187
14	2021-06-19 00:05:59 CET	1903	24.9375	8.42911	0.45842	4.244237	100	4.319	191
15	2021-06-19 00:06:18 CET	1904	24.875	8.43365	0.45842	3.786726	100	24.266	190
16	2021-06-19 00:06:39 CET	1905	24.9375	8.42911	0.45842	3.592592	100	25.204	188
17	2021-06-19 00:06:59 CET	1906	24.875	8.42911	0.45842	3.25169	100	22.618	191
18	2021-06-19 00:07:19 CET	1907	24.875	8.42911	0.45842	4.364429	100	2.484	191
19	2021-06-19 00:07:39 CET	1908	24.875	8.43365	0.45842	2.878745	100	31.898	191
20	2021-06-19 00:07:58 CET	1909	24.875	8.43365	0.45842	3.724584	100	23.928	191
21	2021-06-19 00:08:18 CET	1910	24.875	8.44726	0.45842	4.178197	100	4.124	192
22	2021-06-19 00:08:38 CET	1911	24.875	8.42003	0.45842	2.861857	100	38.652	192
23	2021-06-19 00:08:57 CET	1912	24.875	8.42003	0.45842	4.01385	100	3.465	204
24	2021-06-19 00:09:17 CET	1913	24.875	8.43818	0.45842	4.468699	100	6.592	192
25	2021-06-19 00:09:37 CET	1914	24.875	8.43365	0.45842	2.897241	100	37.925	191
26	2021-06-19 00:09:56 CET	1915	24.875	8.42911	0.45842	4.009667	100	5.612	191
27	2021-06-19 00:10:35 CET	1917	24.9375	8.42911	0.45842	4.106625	100	4.648	189
28	2021-06-19 00:11:32 CET	1918	24.875	8.44726	0.45842	3.990095	100	13.937	189
29	2021-06-19 00:11:51 CET	1919	24.875	8.43818	0.45842	4.265322	100	5.138	189

Processed Sample Collected IoT Pond 1 Data for Pretrained Model (83127 rows)

1	created_at	entry_id	temperature	ph_value	ammonia	salinity	turbidity	Dissolved Oxyg	Nitrate(g/r
143479	2021-10-10 12:06:36 CET	240601	25.125	-0.1414	49.81033	4.134895	100	3.2	1036
143480	2021-10-10 12:10:02 CET	240602	25.0625	-0.1414	37.6209	4.447047	100	3.2	1058
143481	2021-10-10 12:10:21 CET	240603	25.125	-0.12325	37.92929	4.294928	100	3.2	1057
143482	2021-10-10 12:10:41 CET	240604	25.1875	-0.15048	39.51263	3.652199	100	3.2	1054
143483	2021-10-10 12:11:01 CET	240605	25.125	-0.13686	40.16579	4.64252	100	3.2	1050
143484	2021-10-10 12:11:20 CET	240606	25.1875	-0.1414	26.39664	4.592255	100	3.2	987
143485	2021-10-10 12:11:40 CET	240607	25.125	-0.15956	42.5456	4.491138	100	3.2	1050
143486	2021-10-10 12:12:36 CET	240608	25.125	-0.1641	44.89249	3.814151	100	3.2	976
143487	2021-10-10 12:14:11 CET	240609	25.125	-0.15048	33.30557	4.196517	100	3.2	1031
143488	2021-10-10 12:14:31 CET	240610	25.125	-0.15502	53.71302	4.395842	100	3.2	1030
143489	2021-10-10 12:14:50 CET	240611	25.125	-0.15956	53.71302	5.25749	100	3.2	1029
143490	2021-10-10 12:15:10 CET	240612	25.125	-0.13686	34.54133	4.23462	100	3.2	1023
143491	2021-10-10 12:16:07 CET	240613	25.1875	-0.15502	53.9394	4.028022	100	3.2	1021
143492	2021-10-10 12:16:27 CET	240614	25.125	-0.1414	53.71302	4.590651	100	3.2	1019
143493	2021-10-10 12:16:46 CET	240615	25.125	-0.15956	54.85534	4.502939	100	3.2	1022
143494	2021-10-10 12:17:06 CET	240616	25.125	-0.13686	53.9394	4.203235	100	3.2	1015
143495	2021-10-10 12:17:27 CET	240617	25.125	-0.1414	54.39524	4.44377	100	3.2	1021
143496	2021-10-10 12:18:23 CET	240618	25.125	-0.12325	51.29012	4.266088	100	3.2	1023
143497	2021-10-10 12:18:43 CET	240619	25.125	-0.10509	48.37688	4.544337	100	3.2	1023
143498	2021-10-10 12:20:16 CET	240620	25.125	-0.12325	53.71302	4.339539	100	3.2	1019
143499	2021-10-10 12:20:36 CET	240621	25.1875	-0.11871	36.41419	4.574079	100	3.2	1015
143500	2021-10-10 12:22:47 CET	240622	25.125	-0.15502	60.20627	4.404085	100	3.2	1009

Processed Sample Collected IoT Pond 2 Data for Pretrain Model (172250 rows)

Source:

https://www.kaggle.com/datasets/e81da8b7666dc7af41cdc3aa5ef96c5547e4f412598a030f40d444550965e34f/data

POSTER


UTAR
UNIVERSITI TUNKU ABDUL RAHMAN
Bachelor of Computer Science (Honours)
By Bryan Ng Jing Hong
Supervisor: Ts Tan Teik Boon
Implementing Server-Side Federated Learning in an Edge-Cloud Framework for Precision Aquaculture
INTRODUCTION
Precision aquaculture uses smart technologies to monitor pond environments, but small-scale farmers face:
• ❌ High setup costs
• ❌ Poor internet connectivity
• ❌ Privacy risks
🔒 Federated Learning (FL) addresses these by enabling model training at the edge, without sharing raw data.
METHODS
Architecture Components 🔧
• Edge Devices: Raspberry Pi with IoT sensors
• Server: AWS ECS Fargate + S3 + MQTT + gRPC
• FL Algorithm: FedProx
• Encryption: CKKS-Based Hybrid Encryption
• Sync: MQTT → Lambda → DynamoDB
RESULTS
• ✅ Secure aggregation with CKKS worked.
• ✅ Offline → Online sync worked (SQLite → DynamoDB).
• ✅ Raspberry Pi clients + AWS ECS server ran successfully.
• ✅ Model Performance: 📉 MAE, MSE and 📈 R²
DISCUSSION
• ✅ FedProx handles data heterogeneity (non-IID)
• 🔐 CKKS-Based Hybrid Encryption ensures privacy - model parameters never exposed
• 🌐 Biweekly model updates strike a balance between accuracy, energy use, and bandwidth
• 🌱 Offline-capable: Clients store and sync when network returns
CONCLUSION
This project:
• ✅ Implements a secure and scalable FL framework for aquaculture
• ✅ Enables privacy-preserving machine learning for model updates
• ✅ Develops robust data synchronization technique
• ✅ Supports small-scale farmers with low-cost, resilient monitoring systems
Impact: A step toward sustainable, data-driven aquaculture in Malaysia.