

Smart and Self-Sustaining Early-Warning System for Coastal Flooding

Final Report

Prepared by:

Chen Pan¹, Principal Investigator
Mimi Xie¹, Co-Principal Investigator
Hua Zhang², Co-Principal Investigator
Wenlu Wang², Co-Principal Investigator

Contact: Chen Pan, chen.pan@utsa.edu

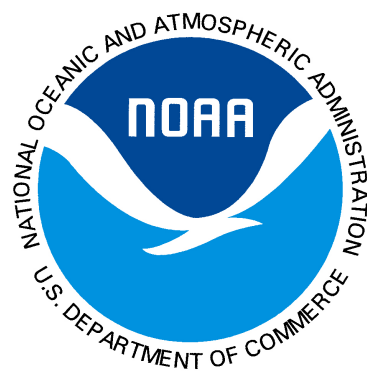
¹The University of Texas at San Antonio One UTSA Circle San Antonio, TX 78249

²Texas A&M University–Corpus Christi 6300 Ocean Drive Corpus Christi, TX 78412

Submitted to:

Texas General Land Office 1700 Congress Ave. Austin, TX 78701-1495

This report was funded in part by a Texas Coastal Management Program grant approved by the Texas Land Commissioner, providing financial assistance under the Coastal Zone Management Act of 1972, as amended, awarded by the National Oceanic and Atmospheric Administration (NOAA), Office for Coastal Management, pursuant to NOAA Award No. NA23NOS4190249. The views expressed herein are those of the author(s) and do not necessarily reflect the views of NOAA, the U.S. Department of Commerce, or any of their subagencies.



June, 2026

Contents

1	Introduction	4
1.1	Background and Motivation	4
1.2	Project Goal and Scope	5
1.3	System-Level Objectives	5
1.4	Implemented System Overview	6
1.5	Report Organization	6
2	System Architecture	8
2.1	Overall System Architecture	8
2.2	Design Requirements and System Rationale	9
2.3	Major Functional Components	10
2.4	End-to-End Data Flow	11
2.5	Energy and Power Flow	11
2.6	Local and Global Intelligence	12
2.7	Dashboard-Centered Decision-Support Workflow	12
2.8	Server Portability and Deployment Flexibility	13
2.9	Summary	13
3	Hardware Prototype	14
3.1	Field Sensing Node Hardware	14
3.2	Sensors, MSP430, and ESP32 Interface Design	16
3.3	Energy-Harvesting Circuit Implementation	17
3.4	Gateway and Edge Server Hardware	17
3.5	Prototype Physical Assembly	18
3.6	Summary	20
4	Embedded Software and Local Flood-Risk Prediction	22
4.1	Embedded Software Overview	22
4.2	Timer-Driven Sensing Cycle	23
4.3	Temperature and Humidity Decoding	24
4.4	ADC Sampling for Light and Precipitation	24
4.5	Discrete Water-Level State Reading	25
4.6	Embedded MLP-Based Local Flood-Risk Prediction	26
4.7	Power-Aware Operation and ESP32 Coordination	29
4.8	Embedded Software Pseudocode	29
4.9	Summary	31
5	Wireless Communication and Data Transmission	32
5.1	Communication Workflow Overview	32
5.2	UART Transfer and Payload Structure	33
5.3	Transmitter Operations	33
5.4	Receiver Data Forwarding and Parsing	34
5.5	Communication Reliability	34

5.6	Summary	35
6	Server, Database, and Dashboard Implementation	36
6.1	Server Implementation Overview	36
6.2	Serial Data Ingestion and Parsing	37
6.3	Local Data Storage	37
6.4	Dashboard Interface	38
6.5	Map, Sensor Plots, and Risk Display	38
6.6	Data Export and Server Portability	39
6.7	Summary	40
7	Global Warning Analysis and Flood-Path Visualization	41
7.1	Global Analysis Overview	41
7.2	Multi-Node Data Preparation and Local Risk Inputs	42
7.3	Spatiotemporal Global Model and Warning Analysis	43
7.4	Flood-Path Visualization	45
7.5	Summary	46
8	Prototype Validation and Deliverables	47
8.1	Validation Scope	47
8.2	Hardware Validation	47
8.3	Software and Data-Workflow Validation	48
8.4	Project Deliverables	50
8.5	Limitations and Future Improvements	52
8.6	Summary	53
9	Non-Technical Guidance and Long-Term Maintenance	54
9.1	Purpose of the Early-Warning System	54
9.2	Interpreting Dashboard Information	54
9.3	Understanding Risk Levels and Warning Outputs	55
9.4	Recommended Operational Workflow	55
9.5	Routine Maintenance	56
9.6	Troubleshooting Common Issues	56
9.7	Software Updates and Model Retraining	57
9.8	Long-Term Deployment Recommendations	57
9.9	Summary	57
10	Outreach, Education, and Workforce Development	59
A	Technical Operation and Software Update Notes	60
A.1	Software Components	60
A.2	Updating the MSP430 Firmware	60
A.3	Updating ESP32 LoRa Code Using Arduino IDE	60
A.4	Running the Raspberry Pi Server	61
A.5	CSV Data Backup and Dataset Management	61
A.6	Retraining the Local MLP Model	61
A.7	Updating the Global DNN Model	62

A.8	Server Migration	62
A.9	Version Control and Backup	62

1. Introduction

Coastal flooding is a recurring hazard for coastal communities and can create significant risks to public safety, transportation, infrastructure, water and soil conservation, and natural habitats [1–4]. Flooding conditions can change rapidly during severe storms, heavy rainfall, storm surge, and other coastal weather events. In many cases, the level of risk is strongly affected by local conditions, such as drainage patterns, low-lying terrain, local precipitation, water accumulation, and the spatial relationship among vulnerable sites. Therefore, effective flood early warning requires not only regional weather information, but also continuous local monitoring and timely interpretation of field-level data. Prior studies have shown that coastal flood early-warning systems can benefit from sensing networks, forecasting models, automated alerting, and decision-support interfaces [5, 6]. Recent research has also explored deep-learning-based coastal flood and inundation prediction under climate-adaptation and sparse-observation settings [7, 8].

This project developed a smart and self-sustaining early-warning system for continuous coastal flood monitoring and prediction. The system integrates distributed sensing nodes, solar energy harvesting [9–16], low-power embedded processing [17–21], Tiny Machine Learning (TinyML) [22–27], long-range wireless communication, server-side data storage, web-based visualization, and flood-risk analysis. The final prototype was designed to support local monitoring of flood-related environmental conditions, node-level flood-risk prediction, and server-side interpretation of multi-node flood patterns.

The overall system follows a field-to-server design. At the field level, sensing nodes collect local environmental and water-related measurements. Each node uses an MSP430FR5994 microcontroller for low-power sensing, data processing, and local flood-risk inference [28]. A BQ25570-based energy-harvesting power subsystem supports self-sustaining operation by harvesting solar energy, charging local energy storage, and providing a regulated 3.3 V output for the embedded electronics [29]. An ESP32/LoRa communication module transmits sensor and node information to a receiver [30–32]. At the server level, the received data are stored, visualized, and analyzed through a dashboard. The dashboard allows users to view sensor trends, inspect node status, select time windows, run warning analysis, and visualize possible flood-path direction.

1.1 Background and Motivation

Many existing flood-warning approaches rely on regional weather forecasts [33, 34], satellite-based observations [35, 36], or sparse monitoring infrastructure. These resources are valuable for large-scale awareness, but they may not provide enough local detail to identify how flood risk develops at specific roads, drainage areas, low-lying locations, or other vulnerable sites [34, 37]. Flooding may begin locally before it becomes visible at a broader regional scale. As a result, community leaders and local officials can benefit from additional local sensing systems that provide more direct information about field conditions [5, 37].

A distributed flood-monitoring system can address this need by placing sensing nodes at selected locations across a monitored area [37, 38]. Each node observes local conditions and reports measurements to a server. When multiple nodes are deployed, the system can provide both local and spatial information. Local measurements help determine the condition around each sensor node, while multi-node data can reveal how flood risk changes across the area over

time [5, 37].

Another important motivation is long-term sustainability. Field monitoring systems are difficult to maintain if they require frequent battery replacement, manual data collection, or constant physical inspection. For coastal and flood-prone areas, maintenance can become even more difficult before, during, or after severe weather events. To reduce this burden, the system developed in this project combines low-power embedded hardware with solar energy harvesting and local energy storage. This design supports longer operation in the field and reduces the need for frequent manual intervention.

The project also addresses the need for intelligent interpretation of sensor data. Raw sensor readings are useful, but local officials often need clear warning information rather than only raw measurements. Therefore, this project combines sensing with embedded machine learning and server-side analysis. The local model provides node-level risk prediction, while the server combines multi-node information to support broader warning analysis and flood-path visualization.

1.2 Project Goal and Scope

The goal of this project is to develop a smart and self-sustaining early-warning system for coastal flooding using low-power embedded sensing, energy harvesting, wireless communication, and machine-learning-based risk prediction. The project focuses on building an integrated prototype that can collect local flood-related measurements, perform node-level risk evaluation, transmit data wirelessly, store and visualize time-series records, and support server-side flood-warning analysis.

The scope of the project includes both hardware and software development. On the hardware side, the project developed a sensor-node prototype that integrates environmental and flood-related sensors, an MSP430 embedded processing unit, an ESP32/LoRa wireless communication unit, and a BQ25570-based solar energy-harvesting power subsystem. On the software side, the project developed embedded sensing and inference logic, LoRa-based data transmission, receiver-side data forwarding, server-side data ingestion, database storage, dashboard visualization, and risk-analysis functions.

The project also includes a practical decision-support component. The dashboard is designed to allow users to view current and historical data, select a time range of interest, conduct risk-warning analysis, and visualize possible flood-path direction. In addition to the technical system, this report includes non-technical guidance for local officials on how to use the system, interpret warning outputs, and maintain the system over time.

1.3 System-Level Objectives

The final system was developed around three main system-level objectives. The **first** objective was to establish a self-sustaining local sensing platform for continuous flood-related monitoring. This includes distributed sensing nodes, environmental and water-level measurements, low-power embedded operation, and solar energy harvesting with local energy storage. The **second** objective was to enable intelligent flood-risk interpretation at both the node level and the server level. Each node supports local flood-risk prediction using an embedded lightweight machine-learning model, while the server combines multi-node records to support broader warning analysis and flood-path visualization. The **third** objective was to provide a practical decision-support interface for local use. This includes wireless data transmission, server-side data stor-

age, dashboard-based visualization, time-window selection, warning-analysis execution, and guidance for long-term operation and maintenance.

These objectives reflect the final implemented structure of the project. The emphasis is not only on building individual hardware or software components, but also on integrating sensing, power management, embedded intelligence, wireless communication, server-side analysis, and user-facing visualization into a complete early-warning prototype.

1.4 Implemented System Overview

The implemented system consists of four connected layers. **1)** The field sensing layer includes the sensors deployed at each monitoring node. The node measures environmental and flood-related information, including temperature, humidity, light, precipitation, and water level. These measurements provide the local data foundation for monitoring and prediction. **2)** The embedded intelligence and power-management layer includes the MSP430 microcontroller, the local prediction model, and the BQ25570-based energy-harvesting subsystem. The MSP430 collects sensor data, maintains recent measurement history, constructs model input features, and runs local MLP-based flood-risk inference. The power subsystem harvests solar energy, charges local capacitor and LiPo storage, and provides a stable 3.3 V output for the MSP430, sensors, and ESP32/LoRa module. **3)** The communication layer connects the field node to the server. The MSP430 sends sensor records to the ESP32 through UART, and the ESP32 transmits the records using LoRa communication. A receiver collects the wireless packets and forwards them to the server. **4)** The server-side decision-support layer stores the received records, provides dashboard visualization, and supports global warning analysis. The dashboard allows users to view sensor trends, inspect node-level status, select time windows, run warning analysis, and visualize possible flood-path direction.

Together, these layers form an end-to-end early-warning prototype that connects local sensing, embedded risk prediction, wireless transmission, server-side analysis, and user-facing visualization. This structure allows the system to support both technical inspection and practical decision-making.

1.5 Report Organization

The remainder of this report is organized according to the final system structure. Chapter 2 presents the overall system architecture, including the field-to-server workflow, energy and power flow, local intelligence, server-side global analysis, dashboard-centered decision-support workflow, and system design requirements. Chapter 3 describes the hardware prototype, including the sensor-node architecture, sensing subsystem, MSP430 embedded processing unit, ESP32/LoRa communication unit, BQ25570-based energy-harvesting power subsystem, receiver/gateway hardware, physical assembly, and field-deployment considerations. Chapter 4 describes the embedded software and local flood-risk prediction method, including sensor acquisition, time aggregation, local input features, risk-level definition, MLP-based local prediction, embedded inference, and power-aware operation.

Chapter 5 describes the wireless communication and data transmission design, including the MSP430-to-ESP32 UART interface, low-level node frame format, LoRa wireless transmission, wake/sleep coordination, receiver operation, server-side ingestion, data storage format, and communication reliability. Chapter 6 describes the server, database, and dashboard implementation, including data ingestion, local time-series storage, node and time-window selec-

tion, sensor plotting, node-status monitoring, and the user interface for warning analysis and flood-path visualization. Chapter 7 presents the global warning analysis and flood-path visualization methods, including multi-node input construction, spatial relationships among nodes, spatiotemporal prediction, area-level warning summaries, and direction visualization.

Chapter 8 presents prototype testing, evaluation, and deliverables. It includes hardware validation, power-subsystem testing, communication testing, server and dashboard testing, local prediction evaluation, global analysis evaluation, end-to-end testing, completed deliverables, technical contributions, limitations, and future work. Chapter ?? provides non-technical guidance and long-term maintenance recommendations for local officials and operators, including dashboard use, warning interpretation, practical response considerations, routine inspection, sensor cleaning, calibration, server backup, troubleshooting, and recommended maintenance schedules.

2. System Architecture

This chapter presents the overall architecture of the smart and self-sustaining coastal flood early-warning system. The system was designed as an integrated field-to-server platform that connects distributed sensing nodes, local embedded processing, energy harvesting, wireless communication, server-side data storage, dashboard visualization, and flood-warning analysis. The purpose of this chapter is to explain how the major system components are connected and why the selected architecture supports continuous monitoring, low-power operation, local flood-risk prediction, and practical decision support.

The architecture follows a layered design. At the field level, sensing nodes collect local environmental and water-related measurements. Each node uses an MSP430 microcontroller for low-power sensing, local data processing, and embedded flood-risk inference. The field node also includes a solar energy-harvesting power subsystem and an ESP32/LoRa communication module. At the receiver side, LoRa packets are collected and forwarded to a Raspberry Pi 4B server. The Raspberry Pi 4B stores the received records, hosts the dashboard, and supports server-side analysis. The dashboard then allows users to review sensor trends, inspect node status, select time windows, run warning analysis, and visualize possible flood-path direction.

2.1 Overall System Architecture

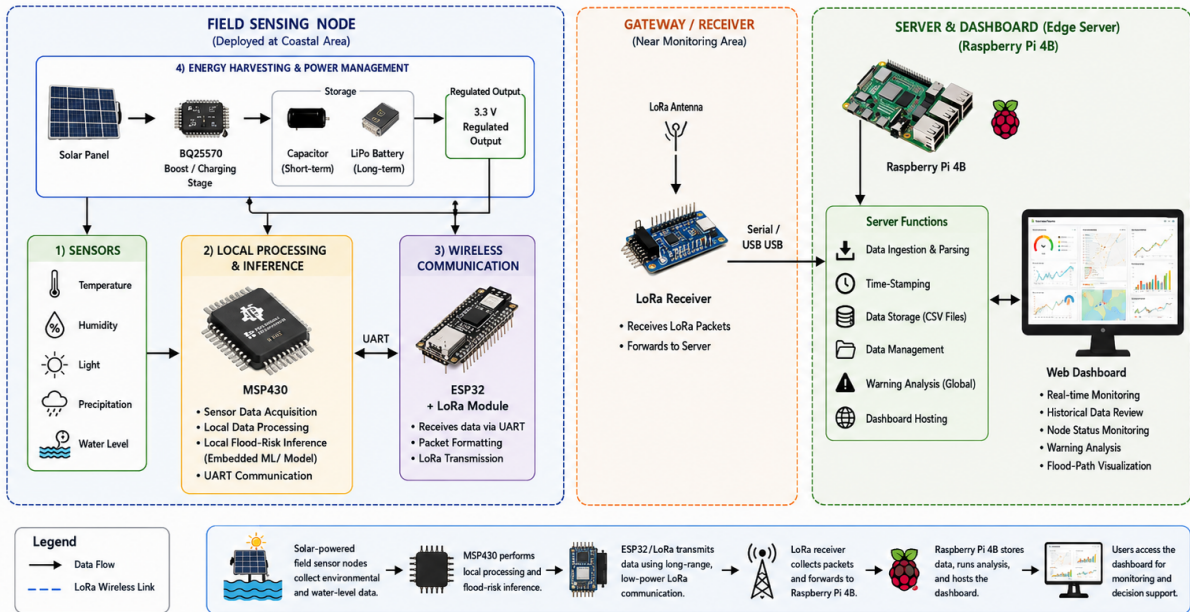


Figure 2.1: System Architecture Early-warning System.

Figure 2.1 shows the conceptual architecture of the complete system. The figure summarizes the major field-to-server workflow, including solar-powered sensing nodes, MSP430-based local sensing and inference, ESP32/LoRa wireless transmission, LoRa receiver/gateway operation, Raspberry Pi 4B server-side storage, dashboard visualization, and global warning analysis.

The architecture is modular so that each major subsystem has a clear role and interface. The sensing node is responsible for local data collection and power-aware field operation. The

MSP430 firmware manages sensor acquisition, local data processing, embedded inference, and UART output. The ESP32/LoRa module provides the wireless transmission function. The receiver and Raspberry Pi 4B server handle packet reception, data parsing, time-series storage, dashboard hosting, and server-side analysis. This modular structure improves system maintainability because individual subsystems can be updated, replaced, or expanded without redesigning the entire platform. It also supports future expansion to additional sensing nodes or migration to different server platforms.

2.2 Design Requirements and System Rationale

The architecture was guided by several technical and practical requirements. These requirements explain why the system uses distributed sensing, low-power embedded processing, solar energy harvesting, LoRa communication, local storage, dashboard visualization, and multi-node warning analysis.

1) Continuous Monitoring. The system must support continuous or periodic monitoring so that flood-related changes can be observed over time rather than only during manual inspection. This requirement motivates the use of distributed sensing nodes, time-series data storage, and dashboard-based historical review. Continuous monitoring also allows the system to support both real-time awareness and retrospective analysis of past weather or flooding events.

2) Low-Power and Self-Sustaining Operation. The field nodes must support low-power and self-sustaining operation because frequent battery replacement is not practical for long-term outdoor deployment. This requirement motivates the use of the MSP430 microcontroller, sleep/wake operation, solar energy harvesting, local capacitor and LiPo battery storage, and BQ25570-based regulated power delivery. The goal is to reduce manual maintenance while allowing the node to continue operating during periods of weak or unavailable solar input.

3) Stable Power Delivery. The embedded electronics require a stable voltage supply for reliable sensing, computation, and wireless transmission. The MSP430, sensors, and ESP32/LoRa module are powered through a stable 3.3 V regulated output from the power-management subsystem. The local storage stage, including the capacitor and LiPo battery, provides energy buffering for short-term current demands and longer operation when harvested energy is limited.

4) Local Embedded Intelligence. The system must provide local embedded intelligence so that each node can estimate its own flood-risk level using nearby sensor measurements. This requirement motivates the lightweight local MLP model and its deployment on the MSP430. Local inference allows each node to provide a direct risk interpretation for its own location instead of only transmitting raw sensor values.

5) Long-Range Low-Data-Rate Communication. The system must transmit compact sensor records from outdoor nodes to a receiver. This requirement motivates the ESP32/LoRa communication design. LoRa is suitable because the system sends small packets at periodic intervals rather than high-bandwidth data streams. The compact frame format reduces packet airtime and supports energy-aware operation.

6) Robustness to Intermittent Connectivity. Outdoor wireless links may be affected by distance, weather, antenna placement, power conditions, and obstruction. Therefore, the system must tolerate missing or delayed packets. This requirement motivates frame validation, server-side parsing checks, missing-data handling, node-status monitoring, and dashboard indicators for inactive nodes.

7) Multi-Node Scalability. The system must support multiple sensing nodes so that local flood-related measurements can be compared across the monitored area. This requirement motivates

node IDs, node-based data organization, time-window filtering, node-location management, and server-side multi-node analysis. Scalability is important because flood risk is spatial and cannot be fully represented by a single sensor node.

8) Practical Visualization and Maintainability. The system must provide practical visualization for both technical users and local officials. This requirement motivates the web-based dashboard, sensor plots, warning-analysis function, flood-path visualization, and non-technical guidance. The system must also be maintainable after deployment, which motivates clear documentation of hardware components, data format, server setup, troubleshooting steps, and long-term maintenance procedures.

2.3 Major Functional Components

The complete system consists of several functional components that work together to support the field-to-server early-warning workflow.

1) Distributed Field Sensing Nodes. Each sensing node is deployed at a selected monitoring location. The node collects local flood-related and environmental measurements, including temperature, humidity, light, precipitation, and water level. These measurements provide the local data foundation for monitoring, local prediction, and server-side analysis.

2) Energy-Harvesting and Power-Management Unit. Each node includes a solar-based energy-harvesting subsystem. A BQ25570-based regulator boosts weak solar input, supports charging of the capacitor and LiPo battery storage stage around approximately 4.2 V, and provides a stable 3.3 V regulated output for the embedded electronics. This subsystem supports the self-sustaining goal of the project.

3) MSP430 Embedded Processing Unit. The MSP430 microcontroller is responsible for sensor data acquisition, local data processing, and embedded flood-risk inference. The local inference model is implemented directly on the microcontroller so that each node can estimate its own flood-risk level before transmitting data to the server.

4) ESP32/LoRa Communication Unit. The ESP32 communicates with the MSP430 through UART and transmits node records using LoRa communication. This design separates low-power sensing and inference from long-range wireless communication. The ESP32/LoRa module can be activated primarily when data transmission is needed.

5) LoRa Receiver and Gateway Interface. The LoRa receiver collects wireless packets from the field nodes. The receiver forwards the received payloads to the server-side software for parsing, timestamping, storage, visualization, and analysis. The receiver is configured to match the LoRa parameters used by the field nodes.

6) Raspberry Pi 4B Edge Server. The gateway-side server is implemented using a Raspberry Pi 4B. The Raspberry Pi 4B receives data forwarded from the LoRa receiver, stores parsed time-series records, hosts the dashboard, and executes server-side analysis functions. This local edge-server design allows the monitoring system to operate near the deployment site without requiring cloud connectivity for basic data collection and visualization.

7) Web-Based Dashboard and Decision-Support Interface. The dashboard provides the user-facing interface for viewing sensor data, checking node status, selecting time ranges, running warning analysis, and visualizing flood conditions. The dashboard supports both technical inspection and practical decision-making by local officials.

2.4 End-to-End Data Flow

As shown in Figure 2.1, the complete data workflow begins at the sensing nodes and ends at the dashboard-based decision-support interface. Each node periodically collects local sensor measurements. These measurements represent the current condition near the node and include environmental readings and water-level information. The MSP430 processes the collected readings, updates local data buffers, and generates a local node record. When sufficient recent history is available, the MSP430 can also run the local flood-risk prediction model.

After local processing, the MSP430 sends the node record to the ESP32 through UART. The ESP32 verifies the frame and transmits the payload through LoRa. The LoRa receiver collects the packet and forwards it to the Raspberry Pi 4B server. The server parses the packet, reconstructs the sensor values, attaches a server-side entry time, and stores the record in local time-series storage. The dashboard then retrieves the stored records based on user-selected node and time-window settings.

The same stored records are also used for analysis. When the user selects a time range and runs warning analysis, the server retrieves the corresponding multi-node data and passes them to the analysis module. The output may include node-level warning results, area-level risk interpretation, and flood-path visualization. In this design, the server-side analysis does not replace the local embedded model. Instead, it uses local measurements and local risk outputs as part of a broader multi-node interpretation.

2.5 Energy and Power Flow

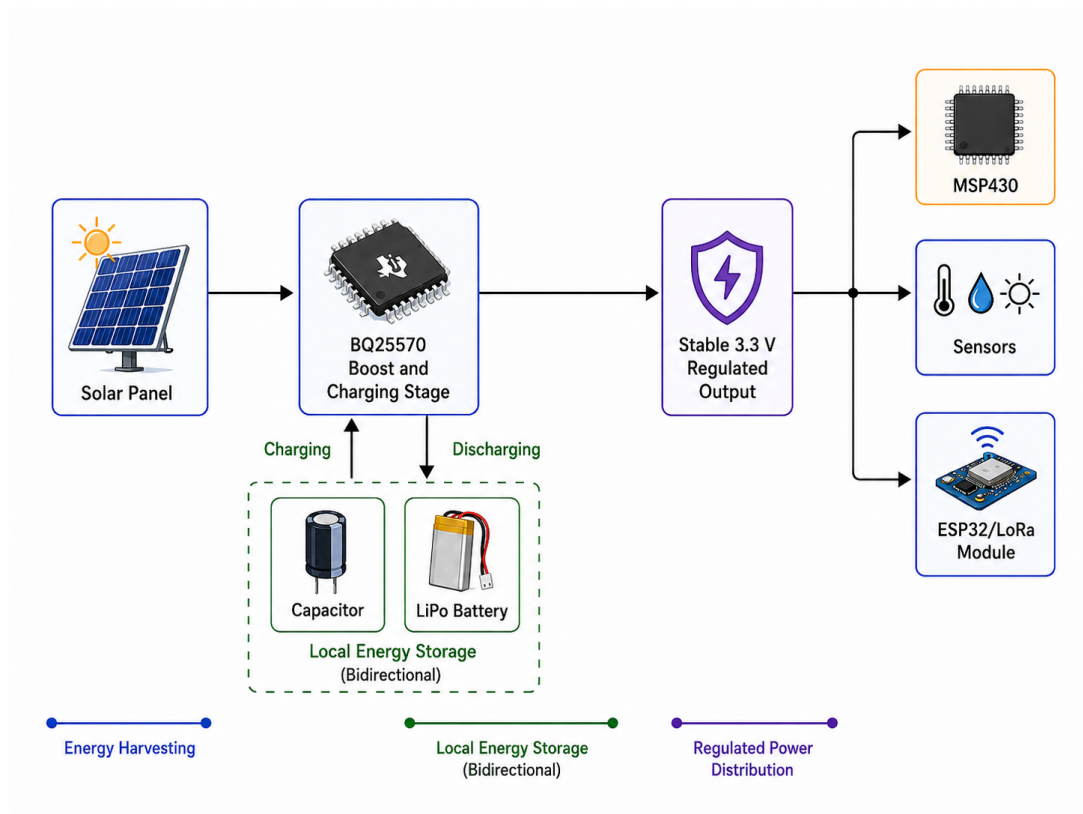


Figure 2.2: Energy-harvesting & Regulated Power-flow Architecture of the Field Sensing Node.

Figure 2.2 shows the simplified power path of each field sensing node. The power subsystem is designed to support long-term outdoor operation with reduced manual maintenance. Solar energy is harvested from the local environment and processed by the BQ25570-based regulator. When the solar input is weak, the regulator boosts the input to a usable level and supports charging of the local storage stage. The capacitor provides short-term energy buffering, while the LiPo battery supports longer operation during periods of weak or unavailable sunlight.

This power-management framework is important because the wireless communication module may require higher instantaneous current during transmission, while the MSP430 and sensors are intended to remain in low-power operation for most of the time. By combining energy harvesting, local storage, voltage regulation, and sleep/wake behavior, the system supports the self-sustaining requirement of the project.

2.6 Local and Global Intelligence

The project uses a two-level intelligence structure: local embedded inference and server-side global analysis. At the local level, each node runs an embedded flood-risk prediction model. This model is designed to be lightweight enough for execution on the MSP430 microcontroller. The purpose of the local model is to provide fast, node-level risk estimation using recent local sensor measurements. This is important because flood conditions may develop differently at different locations, and each node should be able to evaluate its own local condition.

At the server level, the system performs global warning analysis by combining data from multiple nodes. While local prediction focuses on the condition around one node, global analysis focuses on the monitored area as a whole. The server can compare risk levels across nodes, examine temporal trends, and use node locations to infer possible flood movement. For example, if several nodes show increasing risk in a time sequence, the system can use that sequence and the spatial arrangement of nodes to estimate a possible flood propagation direction.

This two-level design provides several benefits. Local inference gives each node an immediate risk indicator, while global analysis identifies broader spatial patterns that are not visible from a single node. The dashboard can then present both node-level risk and area-level interpretation to support practical decision-making.

2.7 Dashboard-Centered Decision-Support Workflow

The dashboard is designed as the main interface between the technical system and the end user. It provides access to raw monitoring data, processed trends, node status, warning results, and flood-path visualization. The dashboard supports both real-time monitoring and retrospective analysis. Real-time monitoring helps users understand current field conditions, while retrospective analysis allows users to review previous events, inspect abnormal readings, and evaluate how risk changed over time.

A typical user workflow is as follows. **1)** The user opens the dashboard hosted by the Raspberry Pi 4B server. **2)** The user checks whether the sensor nodes are online and reporting data. **3)** The user selects one or more nodes or views the full monitoring area. **4)** The user selects a time range for analysis. **5)** The dashboard displays sensor time-series plots for the selected period. **6)** The user runs the flood-risk warning analysis. **7)** The server uses the selected data window to generate local and global risk information. **8)** The dashboard displays risk results and, when applicable, flood-path direction or propagation indicators.

This dashboard-centered workflow is important because it connects technical data collection

with practical interpretation. It allows users to analyze the same system under both current and historical time windows, which is useful for monitoring, testing, validation, and event review.

2.8 Server Portability and Deployment Flexibility

Although the prototype server was implemented on a Raspberry Pi 4B, the server-side software is not limited to this hardware platform. The data-ingestion program, local storage workflow, dashboard, and analysis scripts can be transferred to a desktop computer, workstation, or other computing platform with a compatible Ubuntu or Linux environment. This portability is useful for development, debugging, long-term operation, and future system migration.

2.9 Summary

This chapter presented the overall architecture of the smart and self-sustaining coastal flood early-warning system. The system integrates distributed sensing nodes, solar energy harvesting, BQ25570-based power management, MSP430-based local sensing and inference, ESP32/LoRa communication, LoRa receiver operation, Raspberry Pi 4B server-side data storage, dashboard visualization, and global warning analysis. The architecture was guided by requirements for continuous monitoring, low-power operation, stable power delivery, local intelligence, long-range communication, intermittent-connectivity tolerance, multi-node scalability, practical visualization, and long-term maintainability. The following chapters describe the hardware prototype, embedded software, wireless communication, server implementation, warning analysis, testing results, and non-technical guidance in more detail.

3. Hardware Prototype

This chapter describes the hardware prototype developed for the smart and self-sustaining coastal flood early-warning system. While Chapter 2 presented the overall system architecture, this chapter focuses on the physical implementation of the field sensing node, sensor interfaces, embedded hardware connections, power-management subsystem, communication hardware, receiver, and Raspberry Pi 4B edge server. The goal of the hardware prototype was to provide a practical platform for continuous local sensing, low-power embedded processing, solar-powered operation, and wireless data delivery to the server.

The prototype hardware was designed around a modular structure. The field sensing node integrates environmental and flood-related sensors, an MSP430FR5994 microcontroller, an ESP32/LoRa communication module, and a BQ25570-based energy-harvesting power subsystem. The gateway side includes a LoRa receiver and a Raspberry Pi 4B server. The field node is responsible for data collection and local embedded operation, while the gateway and server are responsible for receiving, storing, displaying, and analyzing the transmitted data.

3.1 Field Sensing Node Hardware

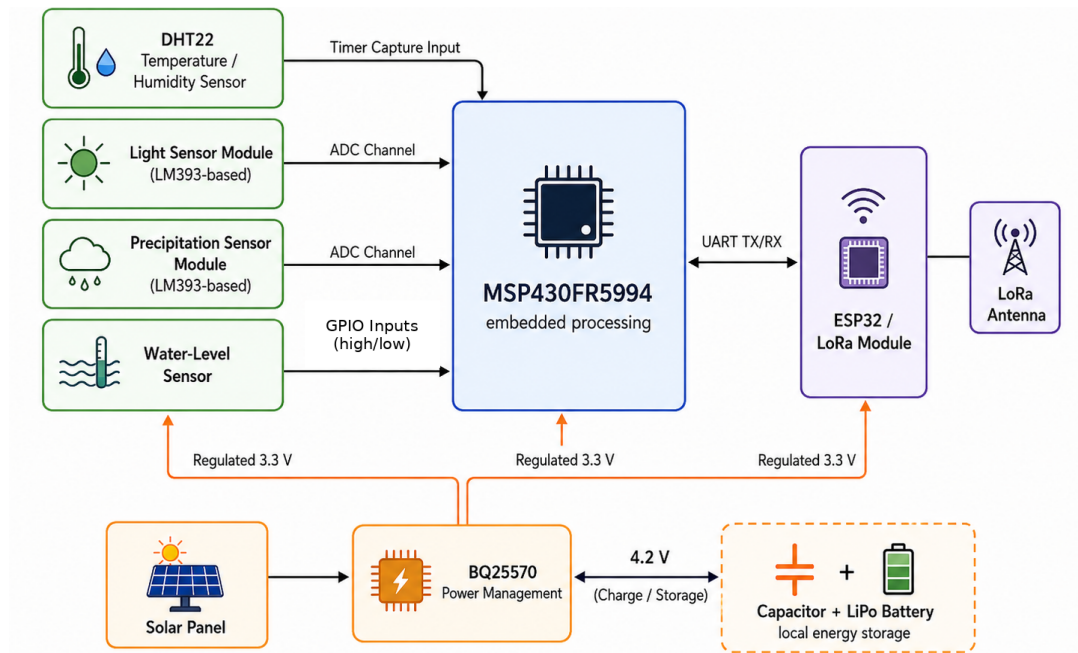


Figure 3.1: Field Sensing Node Hardware Interfaces.

The field sensing node is the primary hardware unit deployed at the monitoring location. It integrates the sensing interfaces, MSP430FR5994 embedded processing unit, ESP32/LoRa communication module, and BQ25570-based energy-harvesting power subsystem. The node is designed to collect local environmental and flood-related measurements, process the sensor data locally, and transmit compact node records to the LoRa receiver.

Figure 3.1 shows the hardware interfaces of the field sensing node. The DHT22 temperature and humidity sensor is connected to a timer-capable MSP430 input for timer capture-based

pulse measurement. The LM393-based light and precipitation sensor modules are connected to MSP430 ADC channels for analog measurement. The water-level sensor is connected to GPIO interrupt inputs configured for falling-edge detection. The MSP430 communicates with the ESP32/LoRa module through UART, and the ESP32/LoRa module provides wireless transmission to the receiver. The BQ25570-based power subsystem receives solar input, manages the local capacitor and LiPo battery storage stage around 4.2 V, and provides regulated 3.3 V power to the MSP430, sensors, and ESP32/LoRa module.

The major hardware components of the field sensing node are summarized as follows. **1)** The MSP430FR5994 serves as the low-power embedded processor for sensing, local data processing, inference, timer capture, ADC sampling, GPIO input reading, and UART communication. **2)** The DHT22/AM2302 sensor provides temperature and humidity measurements through timer-capture-based pulse measurement. **3)** The LM393-based light sensor provides an analog light-related signal to MSP430 ADC12 channel 2. **4)** The LM393-based precipitation sensor provides an analog rain-related signal to MSP430 ADC12 channel 3. **5)** Four FS-IR02B optical water sensors are installed at different height levels to provide discrete water-level detection through MSP430 Port 3 pins 0–3, which are read as GPIO high/low states. **6)** The ESP32/LoRa module receives UART records from the MSP430 and transmits LoRa packets to the receiver. **7)** The BQ25570-based power subsystem harvests solar energy, manages local capacitor and LiPo battery storage around the 4.2 V charging/storage stage, and provides a regulated 3.3 V output to the field-node electronics.

Figure 3.2 shows the sensor modules used in the field sensing node. The node uses a light sensor module for local light measurement, a precipitation sensor module for rain-related sensing, a DHT22/AM2302 sensor for temperature and humidity measurement, and FS-IR02B optical water sensors for discrete water-level detection. These sensors provide the environmental and flood-related inputs used by the MSP430 for local data processing and node-level flood-risk analysis.

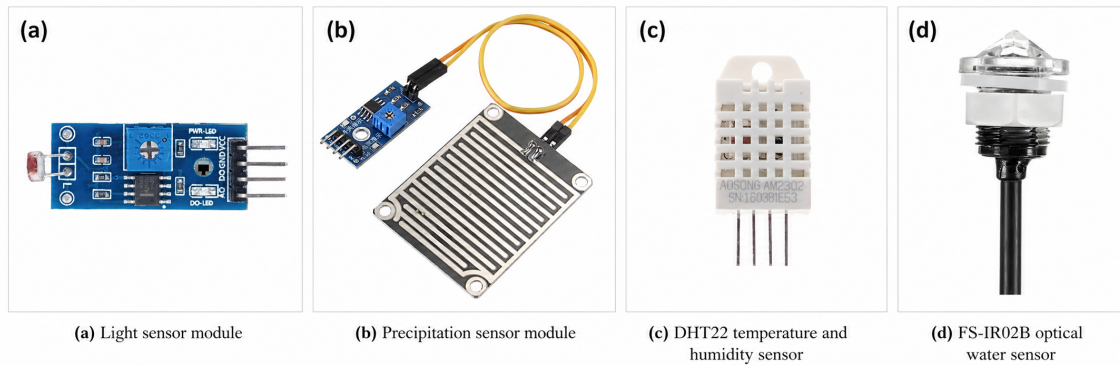


Figure 3.2: Sensor modules used in the field sensing node: light, precipitation, temperature/humidity, and optical water-level sensing.

This node-level design separates sensing, embedded processing, wireless transmission, and power management into clear hardware blocks. This separation makes the prototype easier to assemble, debug, maintain, and extend.

3.2 Sensors, MSP430, and ESP32 Interface Design

This section describes the hardware interfaces between the sensors, MSP430FR5994 microcontroller, and ESP32/LoRa module. The sensing node uses different MSP430 peripheral interfaces according to the signal type of each sensor. The DHT22 temperature and humidity sensor uses a Timer B capture input for pulse-width measurement. The light and precipitation sensors use ADC12 channels for analog measurement. The discrete water-level sensing function uses GPIO input pins that are read as high/low states. The MSP430 communicates with the ESP32 through UART and uses an additional GPIO control line to support ESP32 sleep/wake coordination.

DHT22 temperature and humidity sensor. The DHT22/AM2302 sensor [39] provides temperature and humidity measurements through a single-wire digital signal. Since the DHT22 encodes data using pulse-width timing, its data line is connected to an MSP430 Timer B capture input. This hardware connection allows the MSP430 firmware to measure the pulse widths required for DHT22 decoding. The detailed timing-decoding procedure will be described in Chapter 4.

Light sensor interface. The light sensor uses an LM393-based sensing board [40], where the sensing element produces an analog voltage that changes with the light condition. The LM393 circuit can provide threshold-based digital comparison on many modules, but this prototype uses the analog output path so that the MSP430 can obtain a quantitative light-related measurement. The analog output of the light sensor board is connected to MSP430 ADC12 channel 2. The MSP430 converts this signal into a 12-bit ADC value in the range from 0 to 4095, which is later processed by the embedded software before storage and transmission.

Precipitation sensor interface. The precipitation sensor also uses an LM393-based sensing board. The rain-sensing surface produces an analog response that changes with water contact or wetness, and the LM393 circuit can support threshold comparison if a binary rain/no-rain signal is desired. In this prototype, the analog output is connected to MSP430 ADC12 channel 3 so that the node can record a quantitative precipitation-related measurement instead of only a binary status. The resulting ADC value is processed by the embedded software and included in the node record.

Discrete water-level sensing interface. The water-level sensing function is implemented using four FS-IR02B optical water sensors installed at different height levels. Each FS-IR02B sensor provides a digital water-contact state. Therefore, the four sensors together provide a discrete water-level indication rather than a continuous analog water-level measurement. The four sensor outputs are connected to MSP430 Port 3 pins 0–3. In the current firmware, these pins are configured as GPIO inputs with pull-up resistors, and the MSP430 reads their high/low states during the sensing cycle. This design provides a simple and low-power way to identify which discrete water-level thresholds have been reached.

MSP430-to-ESP32 UART interface. The MSP430 sends node records to the ESP32 through UART. On the MSP430 side, Port 2 pins 5 and 6 are configured for UART operation. The UART is configured at 9600 baud using the system clock. On the ESP32 side, GPIO 39 is used as UART RX and GPIO 40 is used as UART TX. This UART link transfers the sensor record and node-level information from the MSP430 to the ESP32 before LoRa transmission.

ESP32 sleep/wake control. In addition to UART communication, the MSP430 uses Port 3 pin 7 as a control signal connected to the ESP32 wakeup pin, GPIO 4. This signal allows the low-power MSP430 sensing unit to trigger the ESP32 communication module when wireless transmission is needed. This design reduces unnecessary ESP32 active time and supports the energy-aware operation of the field node.

The resulting interface mapping is summarized as follows: 1) DHT22 temperature and hu-

midity use an MSP430 Timer B capture input; **2)** the LM393-based light sensor uses MSP430 ADC12 channel 2; **3)** the LM393-based precipitation sensor uses MSP430 ADC12 channel 3; **4)** the four FS-IR02B optical water sensors use MSP430 Port 3 pins 0–3 as GPIO inputs for discrete high/low water-level detection; **5)** the MSP430 UART uses Port 2 pins 5 and 6 at 9600 baud; **6)** the ESP32 UART uses GPIO 39 as RX and GPIO 40 as TX; and **7)** MSP430 Port 3 pin 7 is connected to ESP32 GPIO 4 for sleep/wake control.

This interface design keeps the hardware connections aligned with the electrical behavior of each sensor and communication signal. Chapter 4 explains how the firmware implements DHT22 pulse decoding, ADC sampling, discrete water-level state reading, UART frame generation, and ESP32 wake/sleep coordination.

3.3 Energy-Harvesting Circuit Implementation

The energy-harvesting circuit is one of the key hardware components of the field sensing node. Its purpose is to support long-term outdoor operation by harvesting solar energy, managing local energy storage, and providing a stable supply voltage to the embedded electronics. This design reduces the need for frequent battery replacement and supports the self-sustaining operation goal of the project.

As shown in Figure 3.1, the power subsystem uses a solar panel, a BQ25570-based power-management circuit, local capacitor and LiPo battery storage, and a regulated 3.3 V output. The solar panel provides the input energy source. When the solar input is weak, the BQ25570-based circuit boosts the input and manages energy transfer between the harvested source, the local storage stage, and the regulated output path.

The BQ25570-based subsystem has two main paths. The **first** path is the storage path connected to the capacitor and LiPo battery. This storage stage is maintained around the 4.2 V charging/storage level and supports both charging and discharging. When sufficient solar energy is available, the BQ25570-based circuit charges the local storage stage. When solar input is weak or unavailable, the stored energy can support continued node operation. The capacitor provides short-term energy buffering, while the LiPo battery provides longer-term energy storage. The **second** path is the regulated output path. The BQ25570-based circuit provides a stable 3.3 V output that powers the MSP430FR5994, sensing modules, and ESP32/LoRa module. This regulated output is important because the embedded electronics require a stable supply voltage for reliable sensing, local processing, UART communication, and wireless transmission.

This design is important because the capacitor and LiPo battery storage stage does not directly power the embedded electronics without regulation. Instead, the BQ25570-based subsystem manages the harvested and stored energy and provides the regulated 3.3 V rail used by the field-node electronics. Combined with low-power MSP430 operation and controlled ESP32/LoRa wakeup and transmission behavior, the energy-harvesting subsystem supports the long-term deployment of the field sensing node.

3.4 Gateway and Edge Server Hardware

The gateway-side hardware consists of a Heltec ESP32 V3 LoRa board used as the receiver and a Raspberry Pi 4B used as the edge server. The Heltec ESP32 V3 receiver collects LoRa packets transmitted by the field sensing nodes and forwards the received data to the Raspberry Pi 4B through a USB serial connection. This USB-based connection is convenient for field operation because the receiver can be easily connected to the Raspberry Pi 4B, or later moved

to a desktop computer, laptop, or server with a compatible Ubuntu/Linux environment.

In the prototype, the LoRa receiver is configured to match the field-node LoRa transmission settings. The system operates in the 915 MHz band with 125 kHz bandwidth, spreading factor 7, coding rate 4/5, and preamble length 8. These settings support low-data-rate transmission of compact sensor records while maintaining a practical communication range for the prototype deployment.

The Raspberry Pi 4B runs an Ubuntu/Linux environment and serves as the local edge server. It receives data from the Heltec ESP32 V3 receiver through USB serial, stores parsed records locally, and hosts the web-based dashboard service. The dashboard can be opened through a browser on the same Ubuntu system. The Raspberry Pi 4B also provides the computing platform for server-side data management and warning-analysis scripts. More detailed server-side software design is described in Chapter 6.

Figure 3.3 shows the gateway and edge-server hardware setup. The figure highlights the receiver-side data path from LoRa packets, to the Heltec ESP32 V3 LoRa receiver, to the Raspberry Pi 4B through USB serial, and finally to the browser-based dashboard running on the Ubuntu/Linux system. It also shows that the USB receiver connection supports migration to another Ubuntu/Linux desktop, laptop, or server when needed.

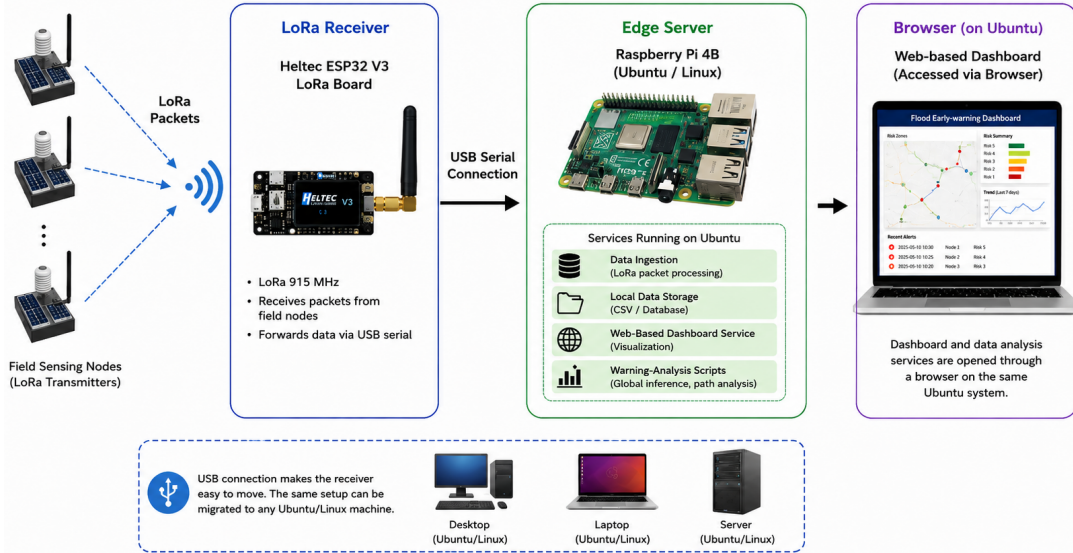


Figure 3.3: Gateway and edge-server hardware setup.

3.5 Prototype Physical Assembly

The hardware prototype includes two physical parts: the outdoor field sensing node and the gateway/edge-server unit. The field sensing node is designed for outdoor-oriented sensing and local communication, while the gateway/edge-server unit is designed for LoRa packet reception, USB serial forwarding, server-side data ingestion, dashboard hosting, and analysis support.

The field sensing node was assembled inside a weatherproof enclosure to protect the MSP430FR5994 microcontroller, ESP32/LoRa module, BQ25570-based power-management circuit, local energy-storage components, wiring, and internal sensor connections. The enclosure provides physical protection while still allowing the node to interface with external sensing and power components. A solar panel is mounted above the enclosure to support energy harvesting, and the LoRa antenna is mounted externally to improve wireless communication with the receiver.

The precipitation sensor is externally mounted so that the rain-sensing surface is exposed to rainfall. The four FS-IR02B optical water-level sensors are installed at different height levels along the vertical PVC support structure to provide discrete water-level detection. The DHT22 temperature/humidity sensor and light sensor are placed inside or near the protected sensing area, where they can measure local environmental conditions while remaining protected from direct physical damage. The externally mounted precipitation and water-level sensors are supported using 3D-printed holders and sealed wiring paths to improve robustness under wet conditions.

Several practical considerations are important for the field-node assembly. The solar panel should be placed where it can receive sufficient sunlight. The LoRa antenna should be positioned to reduce obstruction and improve communication range. The precipitation sensor should remain exposed to rainfall while being protected from mechanical damage. The optical water-level sensors should be mounted at known height levels so that their digital outputs correspond to meaningful discrete water-level states. The enclosure openings and cable entry points should be sealed or protected to reduce water ingress. Cables should be strain-relieved to reduce failure due to wind, handling, or vibration. The node should also be mounted securely while still allowing maintenance access.

Figure 3.4 shows the assembled field sensing node prototype. The figure documents the physical integration of the weatherproof enclosure, solar panel, LoRa antenna, externally mounted precipitation sensor, internal electronics, and discrete water-level sensing structure.

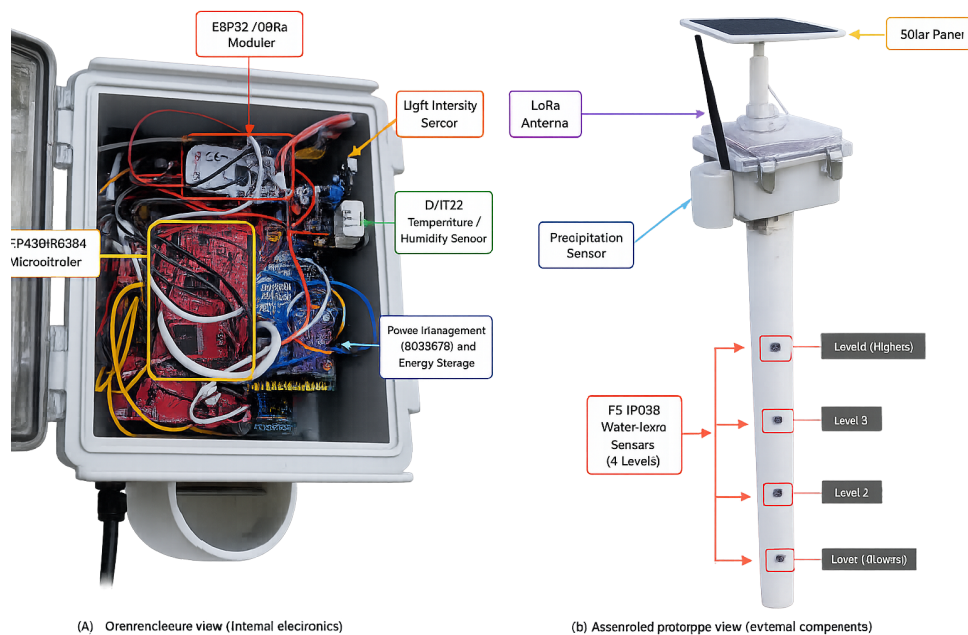


Figure 3.4: Assembled field sensing node prototype.

The gateway/edge-server hardware was assembled using a Raspberry Pi 4B and a Heltec ESP32 V3 LoRa receiver. The Heltec receiver receives LoRa packets from the field sensing node and forwards the received data to the Raspberry Pi 4B through USB serial. The Raspberry Pi 4B runs the Ubuntu/Linux server environment for serial data ingestion, packet parsing, CSV storage, dashboard hosting, and warning-analysis support. The USB-based connection also makes the gateway flexible, because the receiver can be moved from the Raspberry Pi 4B to another desktop, laptop, or server if needed.

Figure 3.5 shows the assembled gateway and edge-server hardware prototype. This figure complements the gateway hardware diagram presented earlier in this chapter by showing the actual physical implementation of the LoRa receiver, USB serial connection, antenna, and Raspberry Pi 4B server platform.

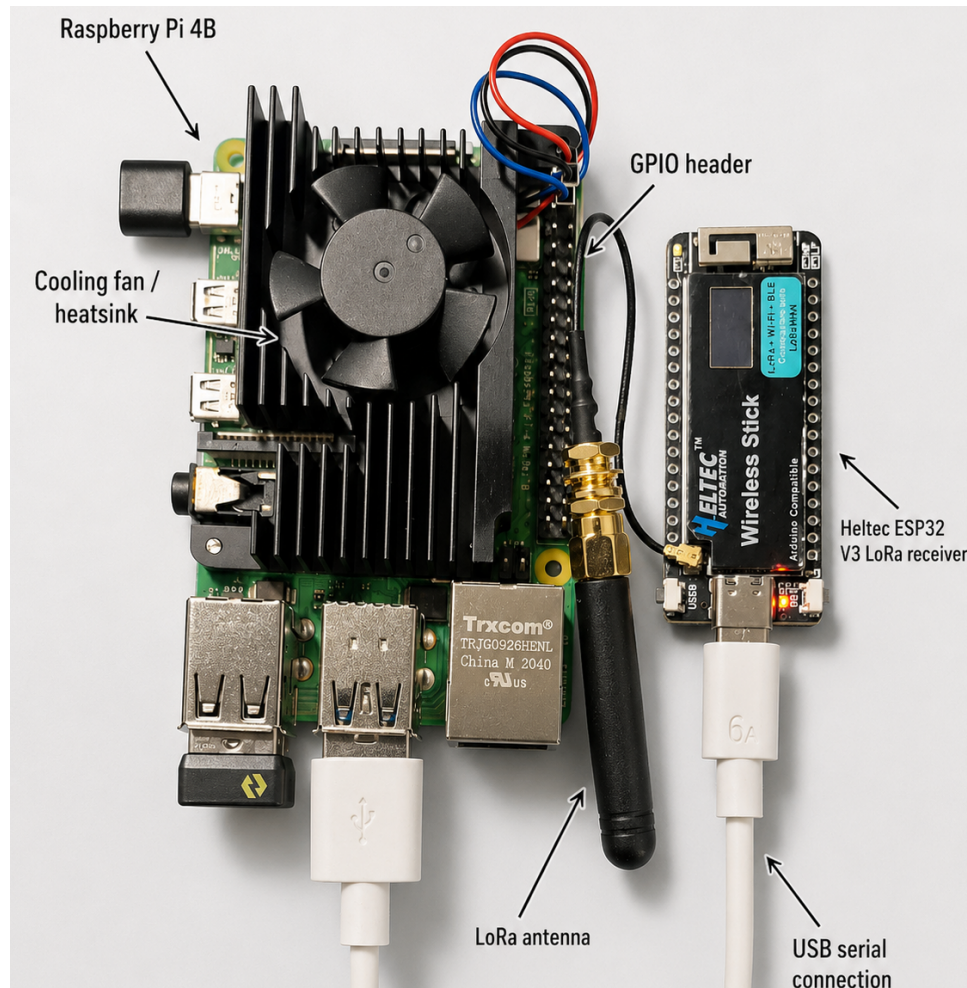


Figure 3.5: Assembled gateway and edge-server hardware prototype with Raspberry Pi 4B and Heltec ESP32 V3 LoRa receiver connected through USB serial.

3.6 Summary

This chapter described the hardware prototype of the smart and self-sustaining coastal flood early-warning system. The field sensing node integrates environmental and flood-related sensors, an MSP430FR5994 microcontroller, an ESP32/LoRa wireless communication module, and a BQ25570-based solar energy-harvesting power subsystem. The DHT22/AM2302 temperature and humidity sensor is connected to an MSP430 Timer B capture input for pulse-width-based measurement. The LM393-based light and precipitation sensor modules are connected to MSP430 ADC12 channels for analog measurement. Four FS-IR02B optical water sensors are installed at different height levels and connected to MSP430 Port 3 GPIO inputs for discrete high/low water-level detection.

The MSP430 communicates with the ESP32/LoRa module through UART, using MSP430

Port 2 pins 5 and 6 on the MSP430 side and ESP32 GPIO 39 and GPIO 40 on the ESP32 side. MSP430 Port 3 pin 7 is connected to ESP32 GPIO 4 to support sleep/wake coordination for energy-aware wireless communication. The power subsystem harvests solar energy, manages capacitor and LiPo storage around the 4.2 V charging/storage level, and provides a stable 3.3 V regulated output for the embedded electronics. The gateway-side hardware includes a Heltec ESP32 V3 LoRa receiver and a Raspberry Pi 4B edge server connected through USB serial for packet reception, local data storage, dashboard hosting, and server-side analysis.

4. Embedded Software and Local Flood-Risk Prediction

This chapter describes the embedded software implemented on the MSP430FR5994 field sensing node. While Chapter 3 described the hardware prototype and sensor interfaces, this chapter focuses on the firmware workflow that converts raw sensor signals into local flood-risk prediction outputs. The embedded software performs timer-driven sensing, DHT22 temperature and humidity decoding, ADC sampling for light and precipitation, discrete water-level state reading, sensor conversion, configurable aggregation-window processing, local MLP-based flood-risk inference, and ESP32 coordination for wireless transmission.

The local prediction model is executed directly on the MSP430. This design allows the field node to produce a local risk estimate before transmitting the node record to the receiver. The local prediction result can then be used by the server and dashboard as part of the broader flood-warning workflow.

4.1 Embedded Software Overview

The MSP430 firmware is organized around a repeated sensing and processing cycle. After initialization, the MSP430 configures the clock system, UART interface, ADC module, timer modules, water-level GPIO inputs, and ESP32 wakeup control pin. The firmware then enters a low-power operating state and wakes periodically through a timer-driven event. During each sensing cycle, the MSP430 activates the DHT22 sensor, captures pulse timing for temperature and humidity decoding, samples the light and precipitation ADC channels, reads the discrete water-level GPIO states, updates the local prediction module, and prepares a node record for transmission to the ESP32/LoRa module.

Figure 4.1 shows the MSP430 embedded software workflow. The workflow is organized into three functional layers: firmware control, sensing and processing, and inference and communication. The control layer initializes the clocks, timers, ADC, UART, GPIO inputs, and ESP32 wakeup pin, then places the MSP430 in a low-power waiting state until the next timer-triggered sensing cycle. The sensing and processing layer acquires one raw sensing packet from the DHT22 temperature/humidity sensor, light sensor, precipitation sensor, and discrete water-level inputs, then converts the raw measurements into the training-aligned representation. The inference and communication layer groups raw packets according to the configurable aggregation count, updates the history buffer, constructs the 16-feature input vector, executes the embedded MLP, updates the local risk prediction, and coordinates ESP32 wakeup and UART transmission.

The workflow in Figure 4.1 separates the MSP430 firmware into sensing, local prediction, and communication-preparation functions. Sensor acquisition produces the raw environmental and discrete water-level measurements. The local prediction path converts these measurements into model-aligned features and generates a five-class flood-risk output. The communication-preparation path attaches the prediction result to the node record and coordinates the ESP32 wakeup signal before UART transmission. The detailed UART frame and LoRa transmission workflow are described in Chapter 5.

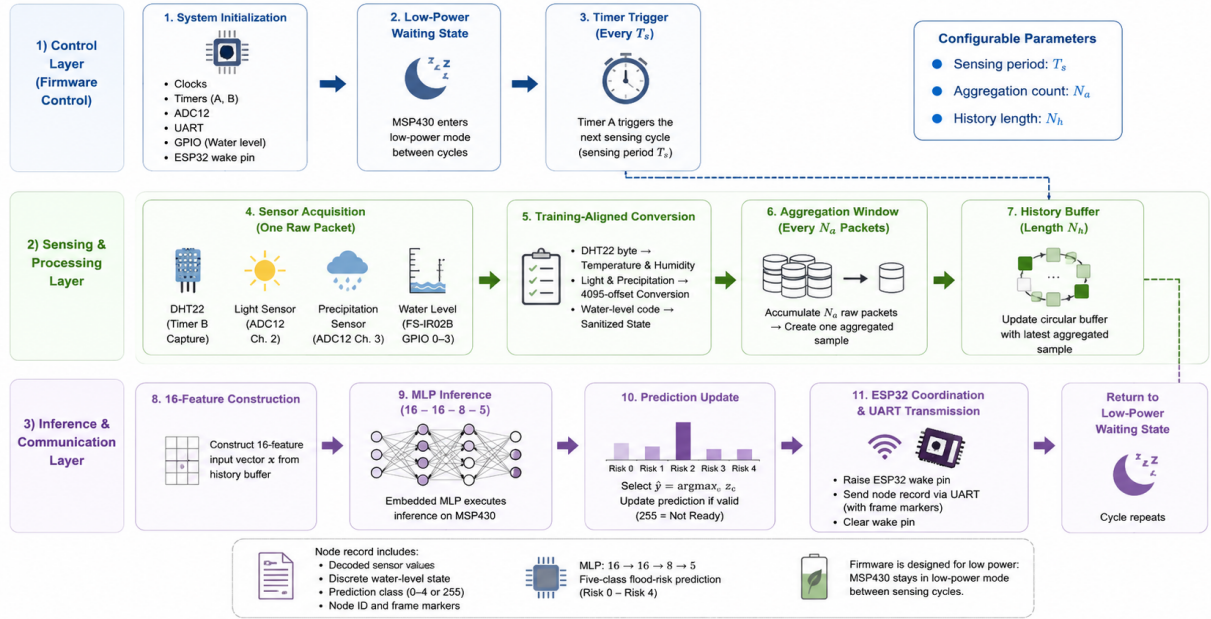


Figure 4.1: MSP430 embedded software workflow.

4.2 Timer-Driven Sensing Cycle

The sensing node uses a timer-driven execution structure. In the implemented firmware, Timer A generates periodic timing events. A timer counter is used to determine when the full sensing cycle should be triggered. Let (T_s) denote the sensing period. The value of (T_s) is configurable in the firmware and determines how frequently the node performs a complete sensing operation. In the current prototype configuration, (T_s) is set to approximately 15 seconds.

The timer-driven structure is important for low-power operation. Instead of keeping the MSP430 active continuously, the firmware allows the microcontroller to remain in a low-power state between sensing events. When the timer condition is met, the firmware starts the DHT22 capture procedure and then performs the remaining sensor acquisition tasks. After sensing, local processing, prediction update, and transmission preparation are complete, the system returns to the waiting state until the next sensing cycle.

The aggregation interval is also configurable. Let (N_a) denote the number of raw sensing packets grouped into one aggregated sample. The aggregation duration is therefore

$$T_a = N_a T_s, \quad (4.1)$$

where (T_a) is the aggregation-window duration. In the current prototype, $(T_s \approx 15)$ seconds and $(N_a = 4)$, so (T_a) is approximately 60 seconds. This setting matches the current model input configuration. If (T_s) , (N_a) , or the feature windows are changed, the model training configuration should be updated accordingly to ensure the embedded features remain aligned with the trained model.

The high-level sensing cycle can be summarized as follows. The timer event starts the sensing cycle. The firmware activates the DHT22 sensor and captures timing pulses for temperature and humidity. After the DHT22 data are reconstructed and verified, the firmware samples the light and precipitation ADC channels, reads the discrete water-level GPIO inputs, updates the local prediction helper, and prepares the node record for UART transmission to the ESP32. The following sections describe these sensor-specific operations.

4.3 Temperature and Humidity Decoding

The DHT22/AM2302 sensor provides temperature and humidity data using a single-wire timing-based digital protocol. The MSP430 must first generate a start signal before the sensor returns its data. In the implemented firmware, the MSP430 briefly waits to ensure timing stability, configures the DHT22 data pin as an output, pulls the line low to issue the start pulse, drives the line high, waits for a short release period, and then switches the pin back to input mode. The pin is then connected to the Timer B capture function so that the MSP430 can measure the timing of the incoming DHT22 response pulses.

After the start signal, the DHT22 responds by sending a sequence of pulses. The data frame contains 40 data bits, which are organized into five bytes: humidity high byte, humidity low byte, temperature high byte, temperature low byte, and checksum. The firmware uses Timer B capture to record the timing of falling edges. The difference between consecutive captured timer values is stored as a pulse-width measurement. A timing threshold is then used to determine whether each bit corresponds to a binary 0 or binary 1.

The firmware reconstructs the five received bytes from the measured pulse widths. The first 8 bits are assigned to the humidity high byte, the next 8 bits to the humidity low byte, the next 8 bits to the temperature high byte, the next 8 bits to the temperature low byte, and the final 8 bits to the checksum byte. The checksum is verified by adding the first four bytes and comparing the result with the received checksum. If the checksum does not match, the firmware clears the temperature and humidity bytes to avoid using invalid DHT22 data.

The DHT22 humidity value is reconstructed as

$$H = \frac{256h_h + h_l}{10}, \quad (4.2)$$

where h_h and h_l are the humidity high and low bytes. The temperature value is reconstructed as

$$T = \frac{256t_h + t_l}{10}, \quad (4.3)$$

where t_h and t_l are the temperature high and low bytes. When the sign bit is set in the temperature field, the firmware removes the sign bit and treats the decoded temperature as negative. These decoded temperature and humidity values are later used by the local MLP feature-construction process.

4.4 ADC Sampling for Light and Precipitation

The light and precipitation sensors are sampled through the MSP430 ADC12 module. Both sensors use LM393-based sensing boards, but the firmware uses the analog output path rather than the threshold-based digital output path. This allows the node to obtain quantitative 12-bit measurements instead of only binary status values.

The light sensor analog output is connected to MSP430 ADC12 channel 2. During the sensing cycle, the firmware selects this ADC channel, starts the conversion, reads the 12-bit result, and stores the raw light value. The precipitation sensor analog output is connected to MSP430 ADC12 channel 3. The firmware then selects the precipitation channel, starts the conversion, and stores the raw precipitation value.

The ADC output range is

$$x_{\text{raw}} \in [0, 4095], \quad (4.4)$$

where x_{raw} represents a raw 12-bit ADC reading. To match the training dataset representation, the embedded inference helper converts the raw readings using

$$L = 4095 - L_{\text{raw}}, \quad (4.5)$$

and

$$P = 4095 - P_{\text{raw}}, \quad (4.6)$$

where L is the training-aligned light value, P is the training-aligned precipitation value, L_{raw} is the raw light ADC reading, and P_{raw} is the raw precipitation ADC reading. This conversion keeps the embedded inference input consistent with the data representation used during model training.

4.5 Discrete Water-Level State Reading

The water-level sensing function is implemented using four FS-IR02B optical water sensors installed at different height levels. Each sensor provides a digital high/low state. Together, the four sensors provide a discrete water-level representation. This is different from a continuous analog water-level measurement. The firmware reads the four water-level signals from MSP430 Port 3 pins 0–3.

In the current implementation, these pins are configured as GPIO inputs with pull-up resistors. During each sensing cycle, the firmware reads the four-bit Port 3 state and masks the relevant bits. The resulting water-level code is stored as a compact discrete state:

$$w = b_0 + 2b_1 + 4b_2 + 8b_3, \quad (4.7)$$

where $b_0, b_1, b_2, b_3 \in \{0, 1\}$ are the digital states of the four FS-IR02B sensors. In the implemented risk representation, the valid water-level codes are

$$w \in \{15, 14, 12, 8, 0\}. \quad (4.8)$$

These codes correspond to increasing water-level conditions as more sensor levels are reached. A lower code indicates that more of the discrete water-level sensors have changed state, which corresponds to a higher detected water level.

The firmware sanitizes the water-level code before using it in the prediction pipeline. The sanitized value is mapped to a local water-risk class as

$$r_w = \begin{cases} 0, & w = 15, 1, \\ w = 14, 2, & w = 12, 3, \\ w = 8, 4, & w = 0. \end{cases} \quad (4.9)$$

The normalized water-risk input is then

$$r_{w,n} = \frac{r_w}{4}. \quad (4.10)$$

This normalized value is included in the local MLP input features.

4.6 Embedded MLP-Based Local Flood-Risk Prediction

The local flood-risk prediction model is implemented as a lightweight multilayer perceptron on the MSP430. The embedded inference helper receives the decoded DHT22 values, training-aligned light and precipitation values, and discrete water-level code. It then performs aggregation-window processing, constructs the 16-input feature vector, and runs the MLP inference function.

Model training and deployment. The MLP is trained offline using training data constructed with the same sensor representation used by the embedded firmware. The training pipeline uses converted temperature, humidity, light, precipitation, and water-risk values, together with recent history-window statistics, to learn a five-class flood-risk classifier. After training, the learned weights and biases are exported into a C header file and compiled into the MSP430 firmware. Therefore, the MSP430 performs inference only; it does not train or update the neural-network parameters in the field. This design keeps the embedded computation lightweight while allowing the model to be retrained offline when additional field data becomes available.

Aggregation-window processing. The embedded inference helper groups multiple raw sensing packets into one aggregated sample before feature construction. Let N_a denote the number of raw sensing packets used in one aggregation window. The aggregated value for a sensor variable x is

$$\bar{x}_a = \frac{1}{N_a} \sum_{k=1}^{N_a} x_k, \quad (4.11)$$

where N_a is configurable. In the current prototype, $N_a = 4$. With the current sensing period, this produces an approximately 60-second aggregated sample. This averaging is applied to temperature, humidity, light, and precipitation. For the discrete water-level code, the firmware keeps the minimum code observed within the aggregation window:

$$w_a = \min(w_1, w_2, \dots, w_{N_a}). \quad (4.12)$$

This rule is used because a lower water-level code corresponds to a higher detected water level.

History buffer. The local inference helper maintains a circular history buffer of aggregated samples. Let N_h denote the number of aggregated samples stored in the history buffer. The current deployed configuration uses $N_h = 60$, but this is a configurable model-design parameter rather than a fixed hardware limitation. The history buffer is used to compute current values, short-term averages, longer-window averages, maximum precipitation values, maximum water-risk values, and precipitation change information. If the aggregation interval or history length is changed, the model training configuration should be updated consistently so that the embedded features remain aligned with the trained model.

Feature construction. The local model uses a 16-dimensional feature vector. To avoid a long one-line expression, the feature vector is written as the concatenation of a current-condition vector and a history-statistics vector:

$$\mathbf{x} = [\mathbf{x} * \text{cur } \mathbf{x} * \text{hist}] \in \mathbb{R}^{16}. \quad (4.13)$$

The current-condition vector is

$$\mathbf{x}_{\text{cur}} = [T_n \ H_n \ L_n \ P_n \ r_{w,n}]^T, \quad (4.14)$$

where T_n , H_n , L_n , P_n , and $r_{w,n}$ are the normalized current temperature, humidity, light, precip-

itation, and water-risk values. The history-statistics vector is

$$\mathbf{x}_{\text{hist}} = \begin{bmatrix} \bar{P}_{W_5} \\ \bar{P}_{W_{15}} \\ \bar{P}_{W_{60}} \\ P_{W_{15}}^{\max} \\ P_{W_{60}}^{\max} \\ \bar{H}_{W_{15}} \\ \bar{T}_{W_{15}} \\ \bar{L}_{W_{15}} \\ r_{w,W_{15}}^{\max} \\ r_{w,W_{60}}^{\max} \\ P_n - P_{n-W_{15}} \end{bmatrix}. \quad (4.15)$$

Here, W_5 , W_{15} , and W_{60} denote feature windows measured in numbers of aggregated samples. In the current deployed model, these windows are set to 5, 15, and 60 aggregated samples, respectively. When the aggregation interval is approximately 60 seconds, these windows correspond approximately to 5, 15, and 60 minutes. If a different aggregation interval is selected, the physical duration represented by each window changes accordingly, and the model should be retrained or reconfigured to match the new setting. The main normalization operations are

$$T_n = \frac{T - 20}{25}, \quad (4.16)$$

$$H_n = \frac{H}{100}, \quad (4.17)$$

$$L_n = \frac{L}{4095}, \quad P_n = \frac{P}{4095}, \quad (4.18)$$

and

$$r_{w,n} = \frac{r_w}{4}. \quad (4.19)$$

The firmware also clips values to reasonable ranges to reduce the effect of abnormal sensor readings before model inference.

MLP architecture. The embedded MLP uses a 16-input, two-hidden-layer, five-output structure. The architecture is

$$16 \rightarrow 16 \rightarrow 8 \rightarrow 5. \quad (4.20)$$

The two hidden layers use ReLU activation. The output layer produces five logits corresponding to Risk0, Risk1, Risk2, Risk3, and Risk4. Figure 4.2 shows the local MLP architecture.

The MLP computation is

$$\mathbf{a}_1 = \text{ReLU}(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1), \quad (4.21)$$

$$\mathbf{a}_2 = \text{ReLU}(\mathbf{a}_1\mathbf{W}_2 + \mathbf{b}_2), \quad (4.22)$$

and

$$\mathbf{z} = \mathbf{a}_2\mathbf{W}_3 + \mathbf{b}_3, \quad (4.23)$$

where $\mathbf{x}$ is the 16-feature input vector, $\mathbf{a}_1$ and $\mathbf{a}_2$ are hidden-layer activations, and $\mathbf{z}$ is the five-dimensional output logit vector. The predicted risk class is selected as

$$\hat{y} = \arg \max_{c \in \{0,1,2,3,4\}} z_c. \quad (4.24)$$

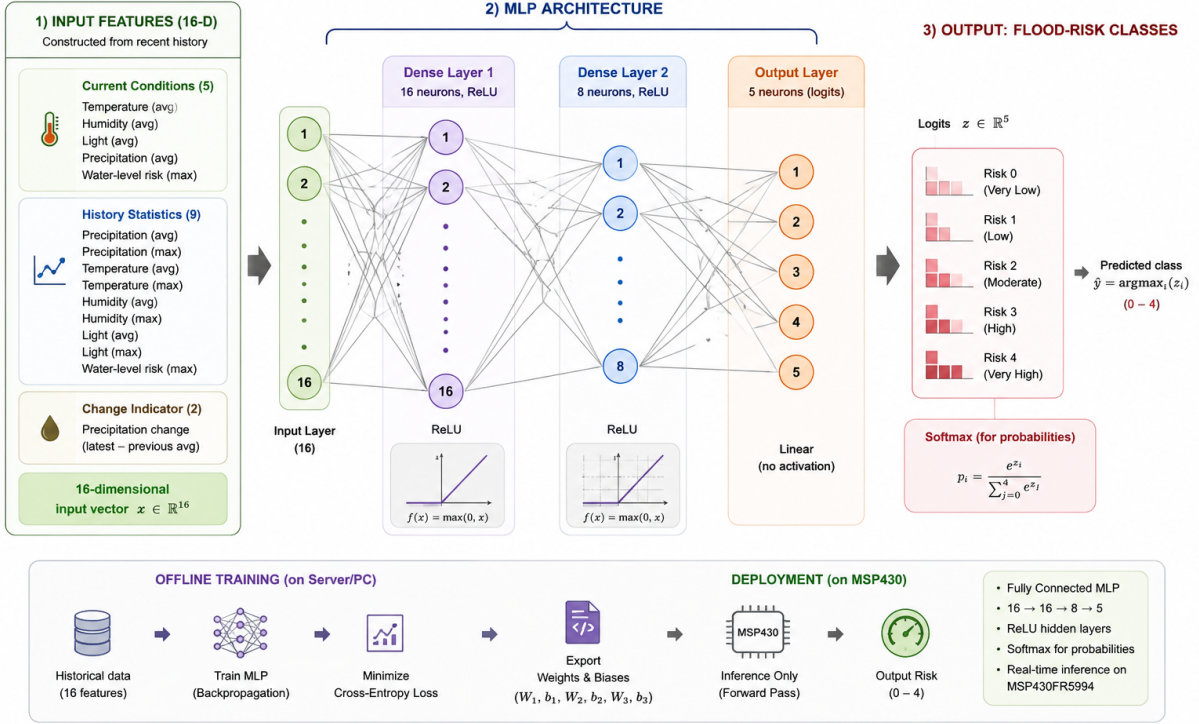


Figure 4.2: Embedded MLP architecture for local flood-risk prediction.

Feature saliency analysis. A saliency analysis can be used to interpret the local MLP behavior. The saliency map estimates the relative influence of each input feature on the model output. This provides a useful diagnostic view because it indicates whether the embedded model is relying on physically meaningful inputs, such as precipitation history, water-level state, humidity, temperature, and temporal changes in precipitation.

Figure 4.3 shows the saliency analysis of the local MLP input features for Node 1. The result shows that **current water risk** is the most influential feature by a clear margin, with a normalized saliency value much larger than all other inputs. This indicates that the local MLP strongly relies on the current discrete water-level condition when producing the flood-risk prediction. The second most important feature is **maximum water risk over the recent 15-minute window**, which suggests that recent water-level persistence or short-term water-level history also contributes to the prediction. The next group of influential features includes **average precipitation over the longer window**, **current temperature**, **average humidity**, and **average temperature**, showing that the model also uses environmental context rather than relying only on the instantaneous water-level state.

The lower-ranked features include current precipitation, light intensity, short-window precipitation averages, and short-window precipitation maxima. This does not mean that these features are unimportant in all conditions; rather, for this saliency analysis, their average influence over the selected valid samples is smaller than the water-risk and longer-window environmental features. Overall, the saliency map is consistent with the intended model behavior: direct water-level evidence dominates the local warning decision, while recent water-level history and environmental trends provide additional contextual information for flood-risk prediction.

Prediction output. The local model outputs one of five risk classes, from Risk0 to Risk4. In the embedded implementation, the class IDs are represented by integer values from 0 to 4. A special value of 255 is used to indicate that prediction is not ready. When a new valid prediction is produced, the firmware updates the stored prediction field. If a new prediction has not yet

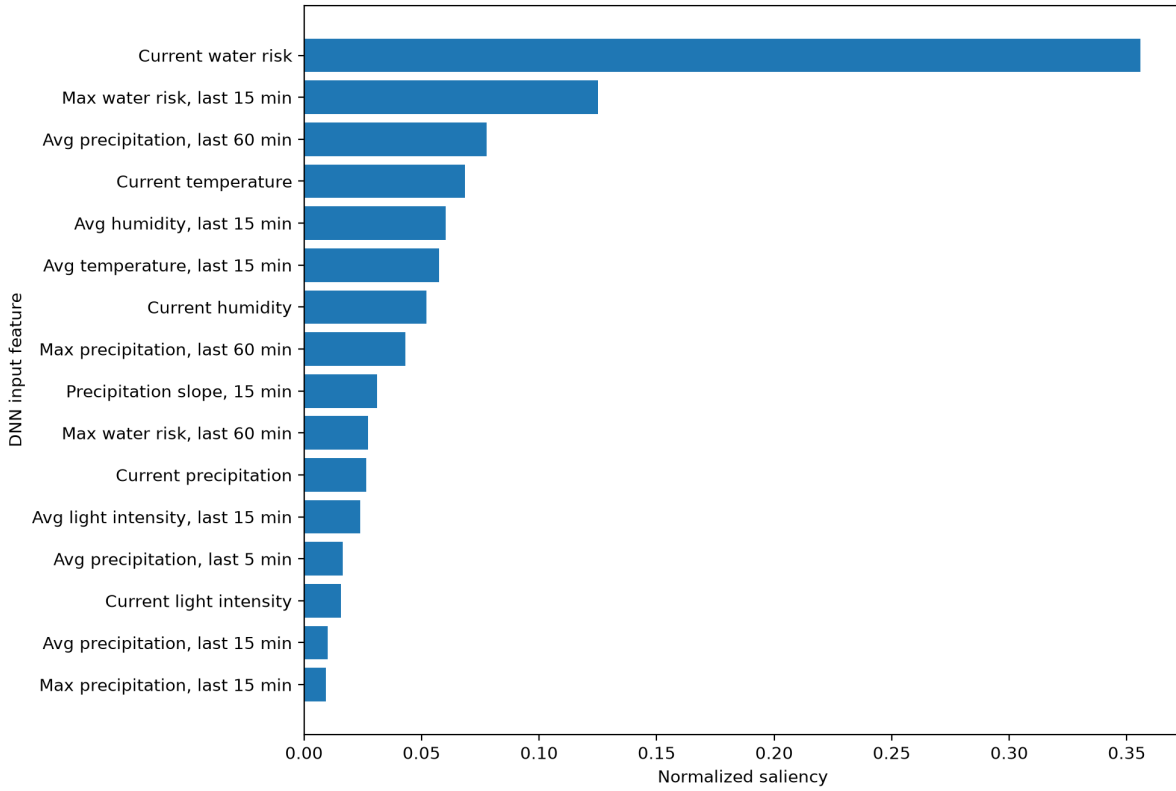


Figure 4.3: Saliency map of the local MLP input features.

been produced, the firmware keeps the previous valid prediction or keeps the not-ready state during startup. The detailed UART frame format used to transmit this node record is described in Chapter 5.

4.7 Power-Aware Operation and ESP32 Coordination

The ESP32/LoRa module consumes more power during wireless communication than the MSP430 consumes during sensing and local processing. Therefore, the firmware coordinates the ESP32 through a dedicated wakeup control line. The MSP430 uses Port 3 pin 7 as the control output connected to ESP32 GPIO 4. Before sending a node record, the MSP430 sets the wakeup control line high, waits briefly, and then sends the record through UART. After transmission, the MSP430 clears the control line so that the ESP32 can return to its low-power state. This approach allows the MSP430 to remain the primary low-power controller of the field node. The MSP430 performs sensing, aggregation-window processing, and local inference, while the ESP32 is primarily activated for wireless communication. This separation is consistent with the self-sustaining hardware design, where the BQ25570-based energy-harvesting subsystem provides the regulated 3.3 V output and the firmware reduces unnecessary active time of the higher-power communication module.

4.8 Embedded Software Pseudocode

Algorithm 1 summarizes the MSP430-side embedded workflow. The detailed ESP32 LoRa transmission and receiver-side parsing are described in Chapter 5. This pseudocode shows the relationship among sensing, local processing, prediction, and communication preparation. The

Algorithm 1 MSP430 embedded sensing and local prediction workflow

Inputs: Sensing period T_s , aggregation count N_a , history length N_h , and configured node ID

Inputs: Trained MLP weights $\{\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \mathbf{W}_3, \mathbf{b}_3\}$

Outputs: Node record containing decoded sensor values, discrete water-level state, prediction class, node ID, and frame markers

- 1: Initialize clocks, UART, ADC12, Timer A, Timer B capture, water-level GPIO inputs, and ESP32 wakeup control pin
- 2: Initialize prediction field as 255 to indicate not ready
- 3: Enter low-power waiting state
- 4: **while** system is active **do**
- 5: **if** Timer A indicates the next sensing cycle **then**
- 6: Generate DHT22 start pulse and release the data line
- 7: Enable Timer B capture for DHT22 pulse timing
- 8: **end if**
- 9: **if** 40 DHT22 timing pulses have been captured **then**
- 10: Decode humidity bytes, temperature bytes, and checksum byte
- 11: **if** checksum fails **then**
- 12: Clear DHT22 data bytes
- 13: **end if**
- 14: Sample light sensor using ADC12 channel 2
- 15: Sample precipitation sensor using ADC12 channel 3
- 16: Read Port 3 pins 0–3 for discrete water-level state
- 17: Convert DHT22 bytes to temperature and humidity
- 18: Convert raw light and precipitation values using the 4095 offset
- 19: Sanitize the water-level code
- 20: Accumulate the raw packet into the aggregation buffer
- 21: **if** N_a raw packets have been accumulated **then**
- 22: Create one aggregated sample
- 23: Update the circular history buffer of length N_h
- 24: Construct the 16-feature input vector $\mathbf{x}$
- 25: Run the DNN inference
- 26: Update prediction class if a new prediction is valid
- 27: **end if**
- 28: Set record head marker, tail marker, node ID, and prediction field
- 29: Raise ESP32 wakeup control signal
- 30: Send node record to ESP32 through UART
- 31: Clear ESP32 wakeup control signal
- 32: Return to low-power waiting state
- 33: **end if**
- 34: **end while**

MSP430 firmware not only collects raw values. It also prepares model-aligned input features and produces a local flood-risk prediction before the node record is transmitted.

4.9 Summary

This chapter described the embedded software and local flood-risk prediction implementation on the MSP430FR5994 sensing node. The firmware uses a timer-driven sensing cycle to periodically acquire temperature, humidity, light, precipitation, and discrete water-level measurements. The DHT22/AM2302 sensor is decoded using a start pulse, release delay, Timer B capture, pulse-width reconstruction, and checksum verification. The light and precipitation sensors are sampled using ADC12 channels, and the four FS-IR02B optical water sensors are read as GPIO high/low states.

The firmware converts raw sensor readings into the training-aligned representation and uses configurable aggregation-window processing. The current prototype uses a configurable sensing period and aggregation count, with $T_s \approx 15$ seconds and $N_a = 4$ as an example setting that produces an approximately 60-second aggregated sample. The firmware then maintains a history buffer of aggregated samples, constructs a 16-feature input vector, and executes a lightweight 16–16–8–5 MLP for five-class local flood-risk prediction. The MLP is trained offline, exported as embedded C weights, and executed on the MSP430 for inference only. The predicted class is stored in the node record and transmitted to the ESP32/LoRa module under MSP430-controlled wake/sleep coordination. The next chapter describes the UART frame, ESP32/LoRa communication, receiver operation, and server-side data transmission workflow.

5. Wireless Communication and Data Transmission

This chapter describes the wireless communication and data-transmission workflow of the coastal flood early-warning prototype. While Chapter 4 described the MSP430-side sensing, processing, and local prediction workflow, this chapter focuses on how the node record is transferred from the MSP430 to the ESP32/LoRa module, transmitted through the LoRa wireless link, received by the gateway-side Heltec ESP32 V3 LoRa board, and forwarded to the Raspberry Pi 4B edge server through USB serial.

The communication design follows a simple and modular path. The MSP430 prepares a compact node record after each valid sensing cycle. The record is transferred to the ESP32 through UART. The ESP32 then sends the record through LoRa. On the receiver side, another Heltec ESP32 V3 LoRa board receives the packet and forwards the received data through USB serial to the Raspberry Pi 4B. This design separates local sensing and prediction from wireless communication and server-side data management.

5.1 Communication Workflow Overview

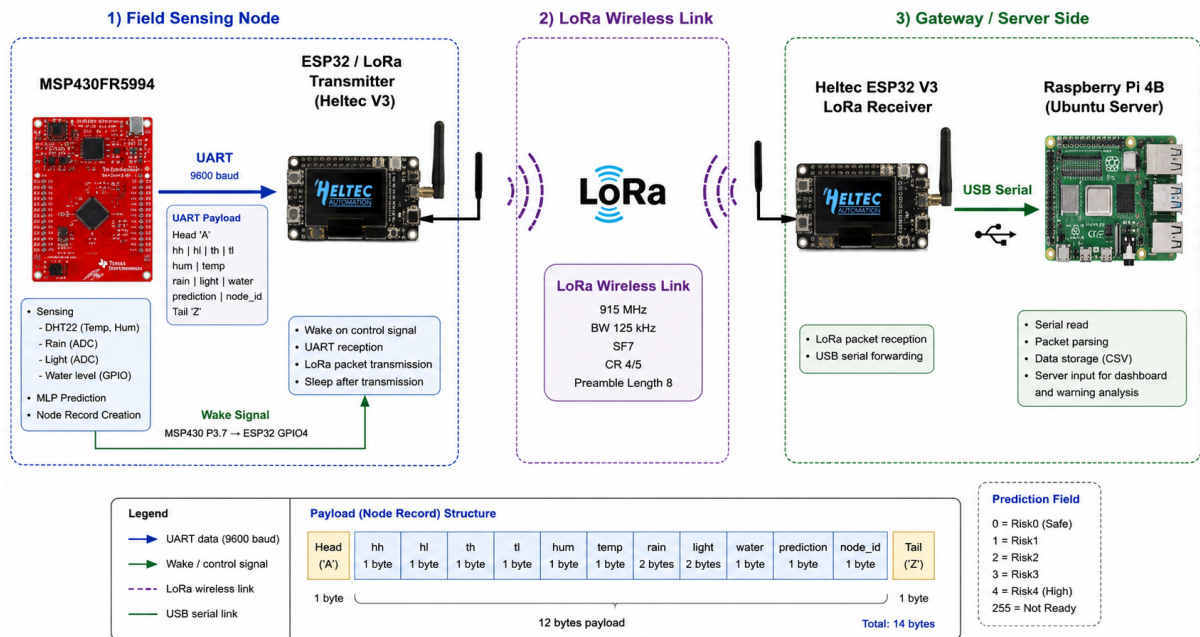


Figure 5.1: Wireless communication and data transmission workflow.

The workflow in Figure 5.1 is intentionally different from the overall system architecture in Chapter 2. The purpose here is not to show all system functions, but to focus on the communication chain that moves one node record from the embedded sensing node to the edge server. The end-to-end communication workflow starts from the field sensing node. The MSP430 collects sensor values, updates the local MLP prediction, and stores the current result in a compact node record. Before UART transmission, the MSP430 raises the ESP32 wakeup control signal. The ESP32 receives the UART record, transmits it through the LoRa radio, and then returns to a lower-power state after the communication event.

On the gateway side, the Heltec ESP32 V3 LoRa receiver listens for incoming LoRa packets using the same radio settings as the transmitter. After a packet is received, the receiver forwards the data to the Raspberry Pi 4B through USB serial. The Raspberry Pi 4B runs the Ubuntu/Linux-based server environment and receives the serial data stream for parsing, local storage, dashboard display, and warning analysis.

Figure 5.1 shows the wireless communication and data-transmission workflow. The figure emphasizes the data path from the MSP430 node record, to the ESP32/LoRa transmitter, to the LoRa receiver, and finally to the Raspberry Pi 4B server through USB serial.

5.2 UART Transfer and Payload Structure

The MSP430-to-ESP32 data transfer uses UART. In the current prototype, the MSP430 UART is configured at 9600 baud. The UART connection carries a compact binary node record rather than a long text string. This reduces communication overhead and allows the ESP32 to forward the received record through LoRa with a small packet size.

The current MSP430 payload is based on a packed sensor-data structure. The packed format avoids padding bytes between fields and makes the transmitted record compact. The payload contains a start marker, DHT22 raw bytes, light and precipitation ADC values, discrete water-level state, local prediction result, node ID, and an end marker. The field order is:

```
Head | hh | hl | th | tl | hum | temp | rain | light | water | prediction |
                                     node_id | Tail
```

The Head field is set to 'A', and the Tail field is set to 'Z'. These two markers provide a simple boundary check for the receiver and parser. The fields hh, hl, th, and tl are the DHT22 humidity and temperature bytes. The fields rain and light are the raw ADC readings from the precipitation and light sensors. The water field stores the four-bit discrete water-level state read from the FS-IR02B optical water sensors. The prediction field stores the local MLP output, where values 0–4 correspond to Risk0–Risk4 and 255 indicates that the prediction is not ready. The node_id field identifies the sensing node.

The current payload still includes one-byte hum and temp fields as part of the sensor-data structure. The main DHT22 temperature and humidity information, however, is reconstructed from the raw DHT22 byte fields hh, hl, th, and tl. Keeping the binary structure fixed helps maintain compatibility between the MSP430 firmware, ESP32 forwarding code, and server-side parser. This payload structure is different from a server-side CSV record. The MSP430 does not send a text field such as DATA at the beginning of the UART payload. Instead, the UART payload is a compact binary record bounded by 'A' and 'Z'. After the receiver and server parse the incoming record, the server can convert the decoded values into structured storage fields for CSV files, plots, and dashboard display.

5.3 Transmitter Operations

The ESP32/LoRa module serves as the wireless transmitter for the field sensing node. Since the ESP32 consumes more power during wireless operation than the MSP430 does during sensing and local inference, the MSP430 controls when the ESP32 should be active for communication. The MSP430 uses the wakeup control line to trigger the ESP32 before sending the UART record.

The transmitter operation follows a short communication sequence. **1)** After the MSP430 finishes sensing and updates the prediction field, it sets the start and end markers of the node

record. **2)** The MSP430 raises the ESP32 wakeup control signal. **3)** After a short delay, the MSP430 sends the packed sensor record through UART. **4)** The ESP32 receives the UART record and transmits it using the LoRa radio. **5)** The MSP430 clears the wakeup control signal so the ESP32 can return to its lower-power state.

The LoRa transmitter uses the 915 MHz band in the prototype. The radio configuration uses 125 kHz bandwidth, spreading factor 7, coding rate 4/5, and preamble length 8. These settings are suitable for compact sensor records and provide a practical balance between data rate, communication range, and packet airtime for the prototype deployment.

The transmitter design keeps the MSP430 as the primary low-power controller. The MSP430 performs sensing, local aggregation, MLP inference, prediction update, and UART payload preparation. The ESP32 is mainly responsible for receiving the UART record and performing the wireless LoRa transmission. This division reduces unnecessary ESP32 active time and supports the self-sustaining goal of the energy-harvesting field node.

5.4 Receiver Data Forwarding and Parsing

The gateway-side receiver is implemented using another Heltec ESP32 V3 LoRa board. The receiver is configured with the same LoRa parameters as the transmitter so that it can receive packets from the field sensing nodes. After receiving a LoRa packet, the receiver forwards the received data to the Raspberry Pi 4B through a USB serial connection.

The USB serial connection is important for practical deployment and maintenance. It allows the receiver board to be connected directly to the Raspberry Pi 4B during normal operation. It also makes the receiver portable, because the same receiver can be moved to a desktop computer, laptop, or other Ubuntu/Linux server for debugging, migration, or higher-capacity operation. This design reduces dependence on a fixed gateway hardware configuration.

On the server side, the incoming serial stream is parsed into structured node records. The parser checks the start and end markers, extracts the node ID, decodes the DHT22 byte fields, reads the ADC-based light and precipitation values, reads the discrete water-level state, and obtains the local prediction field. After parsing, the server can store the decoded information and make it available to the dashboard and warning-analysis functions. The server-side storage, dashboard, and analysis implementation are described in Chapter 6.

The receiver and parser together form the bridge between the wireless field node and the edge-server software. The receiver handles the radio-to-USB forwarding function, while the server converts the forwarded binary record into usable data fields for monitoring and analysis.

5.5 Communication Reliability

Communication reliability depends on both radio configuration and packet parsing. For the LoRa link, the transmitter and receiver must use matching radio settings, including frequency, bandwidth, spreading factor, coding rate, and preamble length. In the prototype, both sides use the 915 MHz band with 125 kHz bandwidth, spreading factor 7, coding rate 4/5, and preamble length 8. Antenna placement is also important. The field-node antenna and receiver antenna should be positioned to reduce obstruction and improve the probability of successful packet reception.

The compact payload structure also supports communication reliability. A smaller packet reduces airtime and lowers the communication burden on the ESP32/LoRa module. The start and end markers provide a simple way to identify valid records and reject incomplete or mis-

aligned data. The node ID allows the receiver and server to distinguish data from different field nodes when the system is extended to multiple sensing nodes.

The current design provides a practical reliability foundation for the prototype. Future extensions can further improve robustness by adding packet sequence numbers, timestamps, explicit checksums, acknowledgments, or retransmission policies. These additions would allow the server to detect missing packets, identify stale readings, and provide stronger end-to-end validation. For the current prototype, the compact packet format, matching LoRa parameters, USB serial forwarding, and marker-based parsing provide a simple and effective communication mechanism.

5.6 Summary

This chapter described the wireless communication and data-transmission workflow of the flood early-warning prototype. The MSP430 prepares a packed binary node record containing DHT22 byte fields, light and precipitation ADC readings, discrete water-level state, local prediction output, node ID, and start/end markers. The MSP430 wakes the ESP32/LoRa module, sends the record through UART, and then releases the wakeup signal. The ESP32 transmits the record through LoRa using the prototype radio configuration. On the gateway side, a Heltec ESP32 V3 LoRa receiver receives the packet and forwards it to the Raspberry Pi 4B through USB serial. The server parses the incoming record and prepares the decoded data for storage, dashboard display, and warning analysis. The next chapter describes the Raspberry Pi 4B server, data storage, and dashboard implementation.

6. Server, Database, and Dashboard Implementation

This chapter describes the Raspberry Pi 4B edge-server implementation, local data storage, and web-based dashboard used in the coastal flood early-warning prototype. Chapter 5 described how the node record is transmitted from the MSP430 to the ESP32/LoRa transmitter, received by the Heltec ESP32 V3 LoRa receiver, and forwarded to the server through USB serial. This chapter focuses on what happens after the data arrive at the server: serial data ingestion, packet parsing, local storage, dashboard visualization, data export, and server portability.

The server is implemented on a Raspberry Pi 4B running an Ubuntu/Linux environment. This choice provides a practical edge-computing platform that can receive data from the LoRa gateway, store decoded records locally, host the web-based dashboard, and support additional analysis functions. The same software design can also be migrated to another Ubuntu/Linux desktop, laptop, or server when more computing capacity or easier maintenance is needed.

6.1 Server Implementation Overview

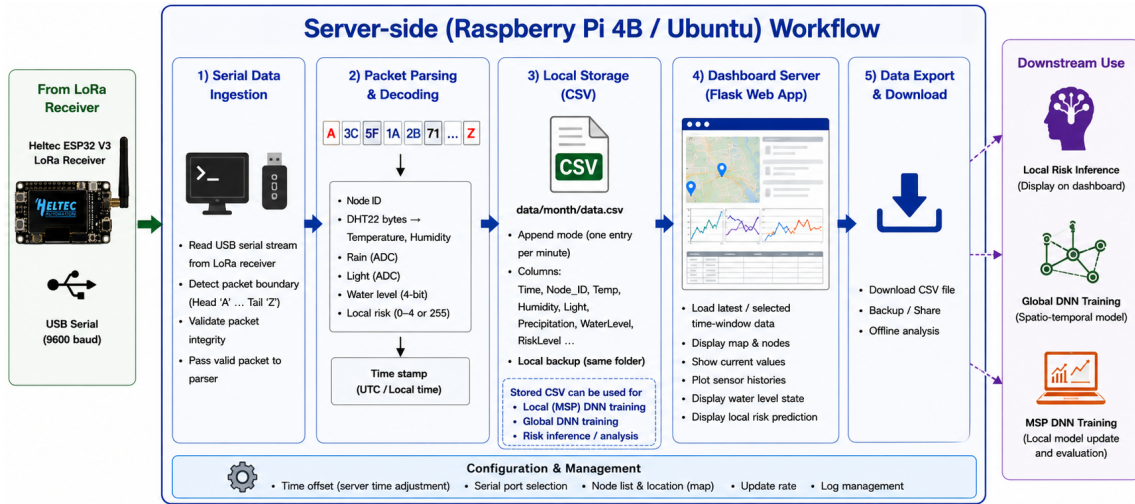


Figure 6.1: Server-side data ingestion and storage workflow.

The server-side implementation provides the bridge between the wireless sensing network and the user-facing monitoring interface. After the Heltec ESP32 V3 LoRa receiver obtains packets from the field sensing nodes, the receiver forwards the data to the Raspberry Pi 4B through a USB serial connection. The Raspberry Pi 4B runs an Ubuntu/Linux-based server environment that continuously reads the incoming serial stream, validates packet boundaries, decodes node records, stores structured data in CSV files, and provides the data source for the dashboard.

The server workflow is organized around the data path shown in Figure 6.1. **1)** The serial data-ingestion function monitors the USB serial port connected to the LoRa receiver and buffers incoming bytes. **2)** The packet-parsing and decoding function identifies valid records using the head and tail markers, extracts the node ID and sensor fields, converts raw packet values into structured measurements, and attaches a timestamp. **3)** The local storage function appends the decoded records to CSV files under the server data folder. These CSV files provide persistent local storage for monitoring and can also support later MSP DNN training, global DNN

training, and risk-inference analysis. **4)** The dashboard server loads the latest or selected time-window data from the CSV storage and presents maps, current sensor values, sensor-history plots, water-level status, and local risk prediction results. **5)** The data-export function allows users to download CSV records for reporting, backup, offline analysis, or future model updates.

Figure 6.1 focuses on the server side of the system. Unlike the communication workflow in Chapter 5, which emphasizes UART, LoRa transmission, and receiver forwarding, this figure emphasizes what the Raspberry Pi 4B server does after data arrive through USB serial. The main server output is clean, decoded, and structured CSV data that can be used by the dashboard and by later analysis modules. The global warning analysis and flood-path visualization built on these stored data are described in Chapter 7. This server-side workflow is intentionally separated from the embedded and wireless communication layers. The field node is responsible for sensing and local prediction. The ESP32/LoRa boards are responsible for wireless packet transmission and reception. The Raspberry Pi 4B server is responsible for data ingestion, validation, parsing, CSV storage, dashboard access, data export, and preparation of structured data for higher-level analysis.

6.2 Serial Data Ingestion and Parsing

The server receives data through the USB serial connection from the LoRa receiver. The receiver-side ESP32 forwards the incoming LoRa packet to the server, and the server-side process continuously monitors the corresponding serial port. Each incoming record is checked before being accepted as valid data. The start and end markers described in Chapter 5 provide a simple way to identify complete node records and reject incomplete or misaligned packets.

After a valid record is identified, the parser extracts the node ID, DHT22 humidity and temperature bytes, light and precipitation ADC readings, discrete water-level state, and local prediction field. The server-side parser converts these low-level packet fields into structured values that can be stored and displayed. For example, the DHT22 bytes are converted into temperature and humidity values, the ADC-based light and precipitation readings are converted into the same representation used by the dashboard, and the water-level code is converted into a human-readable water-level state.

The parsing step is important because the communication payload is compact and embedded-oriented, while the dashboard requires human-readable and analysis-ready values. The server therefore acts as the conversion point between the compact node record and the structured monitoring data used by the rest of the system.

6.3 Local Data Storage

The server stores decoded node records locally so that the dashboard can display both current conditions and historical sensing trends. Each stored record includes a timestamp, node ID, temperature, humidity, light intensity, precipitation, water-level state, and local prediction result when available. These records form the local data layer used by the dashboard plots, data export function, local model evaluation, and later global warning-analysis workflow.

The prototype data-collection workflow produced approximately 17 months of CSV-based sensing records. Although the actual sensing rate of the field node is higher than one sample per minute, the stored dataset is organized at a one-minute resolution. Raw readings collected within each minute are averaged to produce one representative sample per minute. This aggregation reduces short-term fluctuation, keeps the dataset size manageable, and provides a consistent

time base for visualization, model training, and warning-analysis workflows.

A file-based CSV storage approach is used in the prototype. This approach is simple, transparent, and easy to back up. It also supports field operation because records can be inspected directly without requiring a complex database server. As new records arrive, the server appends the decoded values to the local CSV storage. The resulting files can be used by the dashboard for time-window visualization and can also be exported for offline inspection, model evaluation, dataset preparation, or report generation.

This storage design separates the communication payload from the analysis-ready dataset. The MSP430 and ESP32/LoRa modules use compact records for efficient transmission, while the server converts the received data into structured CSV records that are easier to read, visualize, and reuse. This design makes the dataset portable and supports later updates to the local MLP model, server-side global model, and flood-risk analysis workflow.

6.4 Dashboard Interface

The dashboard is implemented as a web-based interface running on the Ubuntu/Linux server environment. It provides a user-facing view of the data stored by the Raspberry Pi 4B server and allows users to inspect node location, current sensing values, historical trends, water-level status, local risk prediction, and exported data records. The dashboard can be opened through a browser on the server system, which makes it convenient for prototype testing, local inspection, and field-side operation without directly viewing raw serial messages or CSV files.

Figure 6.2 shows the web-based flood monitoring dashboard. The left sidebar provides configuration and data-management functions, including node latitude and longitude settings, temperature display unit selection, CSV source-unit selection, and data export. The main panel displays an interactive map showing the selected node location, followed by the plot-control area where the user can select the time range to visualize. Below the time-window selector, the dashboard presents a current-status summary, including temperature, humidity, light intensity, precipitation, water-level state, and local risk prediction. The lower part of the interface shows the selected sensor plot over time, allowing users to inspect recent sensor trends.

The dashboard is intended to support both technical inspection and practical monitoring. For technical users, it provides a quick way to confirm whether serial ingestion, parsing, storage, plotting, and prediction display are working correctly. For non-technical users, it provides a simplified monitoring interface that combines map-based node location, current sensor readings, water-level status, and local risk output in one view. This design reduces the need for users to interact directly with low-level packet data or server-side storage files.

6.5 Map, Sensor Plots, and Risk Display

The dashboard combines spatial, numerical, and temporal information in one interface. The map component displays the selected sensing node location using the stored latitude and longitude values. This helps users associate the incoming measurements with a physical location. The map also supports interactive inspection, such as zooming and panning, so users can view the node position in relation to nearby roads, terrain, or drainage-related features.

The current-status summary provides a compact numerical view of the latest received data. In the shown interface, the dashboard reports sensor temperature, sensor humidity, light intensity, precipitation, water level, and local prediction risk level. This summary allows users to quickly understand the current node condition without reading the full data table or opening the

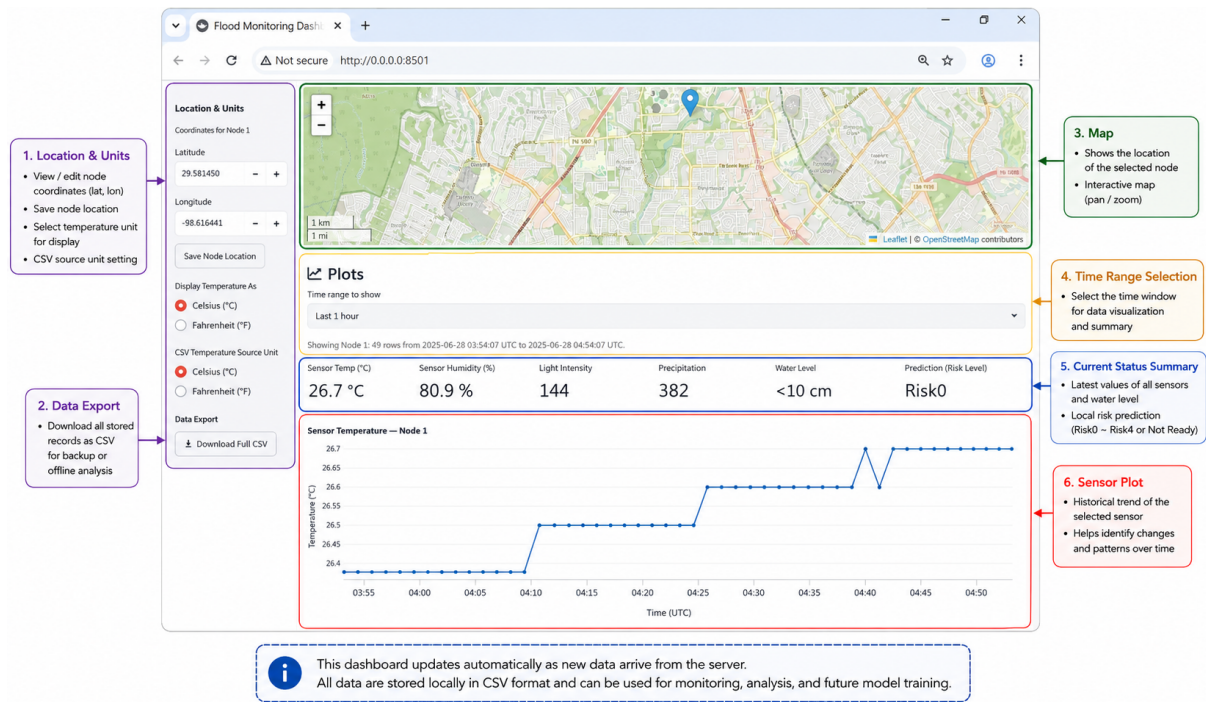


Figure 6.2: Web-based flood monitoring dashboard with annotated interface functions.

CSV file. The water-level display provides a human-readable state, while the prediction field shows the embedded local MLP output as a risk class such as Risk0–Risk4 or not ready.

The plot component allows users to review sensor trends over a selected time window. The time-range selector controls the amount of recent data shown in the plots, such as the last hour or another selected interval. This is important because flood risk is not determined only by one isolated reading. Recent changes in precipitation, water level, humidity, temperature, and other environmental variables provide useful context. By combining the selected time range with sensor-history plots, the dashboard helps users identify changes, stable periods, sudden jumps, or abnormal values in the incoming data.

The data export function allows users to download stored CSV records for backup, reporting, offline inspection, or future model development. This function is useful because the same server-side CSV data support both daily monitoring and later technical analysis. The dashboard therefore serves as the main access point between the field sensing system and the user, while the underlying server continues to maintain structured local data storage for analysis and model updates.

6.6 Data Export and Server Portability

The dashboard includes a data export function so that users can download stored records for offline analysis, reporting, debugging, or model retraining. Exported data are useful for validating sensor behavior, checking communication reliability, comparing local predictions with observed conditions, and preparing datasets for future model updates.

The server design is portable because the LoRa receiver connects through USB serial and the software runs in an Ubuntu/Linux environment. In the prototype, the Raspberry Pi 4B acts as the edge server. However, the same receiver and server software can be moved to a desktop, laptop, or higher-capacity server with a compatible Ubuntu/Linux environment. This portability

is useful for maintenance, debugging, and future expansion. For example, a Raspberry Pi 4B can support local field operation, while a desktop or server can be used later for larger-scale data processing or longer-term hosting.

The portability also reduces long-term deployment risk. If the Raspberry Pi 4B fails or more computing resources are needed, the receiver can be connected to another Ubuntu/Linux system and the server workflow can continue with limited changes. This design supports the maintainability goal of the early-warning system.

6.7 Summary

This chapter described the Raspberry Pi 4B edge-server, local storage, and dashboard implementation. The server receives forwarded LoRa packets through USB serial, parses valid node records, converts compact payload fields into structured sensing values, and stores the decoded data locally. The dashboard provides a web-based interface for viewing node location, current sensor values, time-window-based plots, water-level state, local risk prediction, and data export. The server implementation is portable because the LoRa receiver uses USB serial and the software can run on a Raspberry Pi 4B or another Ubuntu/Linux desktop, laptop, or server. The next chapter describes how the stored data and local node predictions are used for global warning analysis and flood-path visualization.

7. Global Warning Analysis and Flood-Path Visualization

This chapter describes the server-side global warning analysis and flood-path visualization framework. The local MSP430 model described in Chapter 4 produces node-level flood-risk predictions from measurements collected at one sensing node. The global analysis described in this chapter operates at the server level and uses multi-node information to estimate broader flood-risk patterns across the monitored area. Therefore, the global model does not replace the local model. Instead, it uses local risk outputs, sensor measurements, water-level states, and node locations as inputs for spatiotemporal warning analysis.

The purpose of the global analysis is to move from single-node risk prediction to network-level situational awareness. A local node can indicate whether its own location is at risk, but flooding is spatial and temporal. When multiple nodes are available, the server can compare how risk levels evolve across nearby locations and visualize possible flood progression. This chapter explains how multi-node data are prepared, how local risk outputs are used as global model inputs, how the spatiotemporal DNN performs warning analysis, and how the resulting predictions are visualized as node-level risk and flood-path information.

7.1 Global Analysis Overview

The global warning analysis is implemented on the server side using the structured data produced by the ingestion, parsing, and storage workflow described in Chapter 6. The server stores decoded node records in CSV files. These records include environmental measurements, discrete water-level states, node IDs, timestamps, and local risk predictions. For global analysis, the stored records are aligned across nodes and converted into a multi-node temporal sequence.

The global analysis has three main functions. **1)** It organizes stored sensor records from multiple nodes into time-aligned sequences. **2)** It combines local node-level risk outputs with sensor-derived features and spatial node relationships. **3)** It predicts future risk levels across the node network and visualizes possible flood-risk propagation. This design allows the dashboard to provide more than a list of individual node readings. It supports a higher-level view of how flood risk may develop across the monitored region.

The global analysis workflow is shown in Figure 7.1. The workflow begins with server-side CSV storage, where decoded node records are stored with timestamps, node IDs, sensor measurements, water-level states, local risk levels, and risk-probability values. These records are first processed by the data-alignment stage, which aligns timestamps across multiple nodes, resamples the records to a fixed interval when needed, handles missing values, and applies normalization or scaling. The aligned records are then converted into a multi-node input tensor $\mathbf{X} \in \mathbb{R}^{T \times N \times F}$, where T is the input time-window length, N is the number of sensing nodes, and F is the number of features per node.

In parallel, the node-location information is used to construct the spatial graph. The graph represents the relationship among sensing nodes through an adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$, where connected or nearby nodes are assigned adjacency values or weights. The global warning model then uses both the temporal feature tensor $\mathbf{X}$ and the spatial adjacency matrix $\mathbf{A}$ as inputs for server-side inference. The model outputs future node-level risk predictions $\hat{\mathbf{Y}} \in \mathbb{R}^{H \times N \times C}$, where H is the prediction horizon and C is the number of risk classes. These prediction results are finally sent to the dashboard to support warning display, risk-map visualization, warning tables, and flood-path visualization.

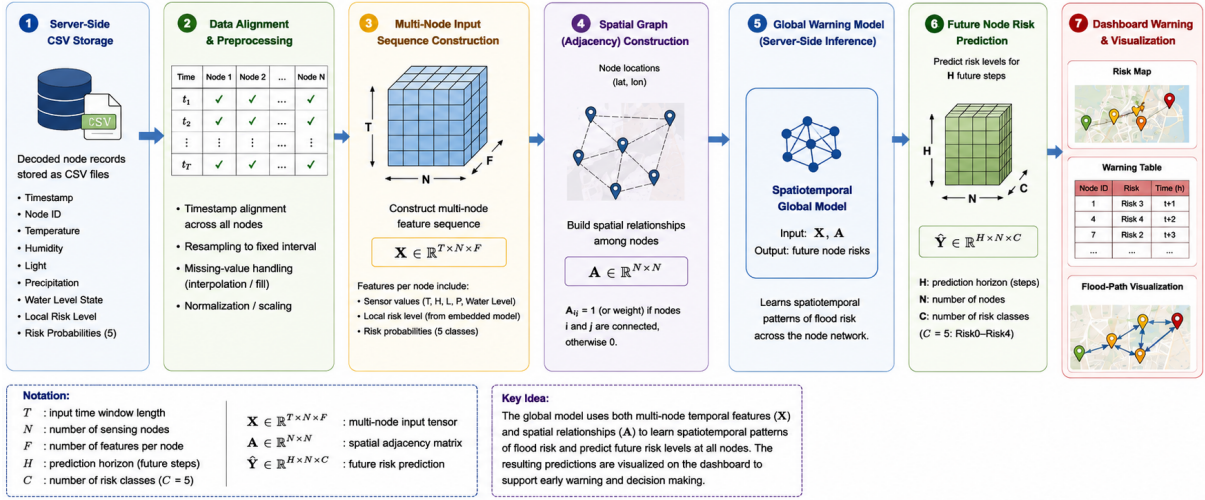


Figure 7.1: Global warning analysis workflow

This workflow highlights the analysis stage that occurs after the server has already received, parsed, and stored the node records. At this stage, the stored CSV data are no longer treated only as monitoring records. They are reorganized into time-aligned multi-node sequences and combined with node-location information so that the global model can analyze spatial and temporal risk patterns across the sensing network.

7.2 Multi-Node Data Preparation and Local Risk Inputs

The global model requires time-aligned data from multiple sensing nodes. Each node record contains measurements collected at a local sensing node, including temperature, humidity, light, precipitation, discrete water-level state, and local risk prediction. Since records may arrive at slightly different times, the server-side preparation step aligns data by timestamp and organizes the values into a consistent multi-node sequence.

Let (N) denote the number of sensing nodes, (T) denote the temporal input-window length, and (F) denote the number of features per node. The global model input can be represented as

$$\mathbf{X} \in \mathbb{R}^{T \times N \times F}, \quad (7.1)$$

where each time step contains feature vectors for all nodes. For training and batch inference, the input can be extended to

$$\mathbf{X}_B \in \mathbb{R}^{B \times T \times N \times F}, \quad (7.2)$$

where (B) is the batch size.

The feature vector for each node includes both sensor-derived measurements and local risk information. In the current design, the global input features include temperature, humidity, light, precipitation, water-level state, the local predicted risk level, and local risk-probability outputs. The local risk outputs are important because they summarize the node-level embedded inference result and allow the global model to build on the local intelligence already produced by the sensing nodes. The local model summarizes recent sensing behavior at each node, while the global model compares these local outputs across space and time.

The server-side data preparation can be summarized as

$$\mathbf{x} * t, i = \left[T * t, i, H_{t,i}, L_{t,i}, P_{t,i}, W_{t,i}, R_{t,i}, p_{t,i}^{(0)}, p_{t,i}^{(1)}, p_{t,i}^{(2)}, p_{t,i}^{(3)}, p_{t,i}^{(4)} \right], \quad (7.3)$$

where $\mathbf{x} * t, i$ is the feature vector for node i at time step t , T , H , L , P , and W represent temperature, humidity, light, precipitation, and water-level state, $R * t, i$ is the local predicted risk level, and $p_{t,i}^{(0)}$ to $p_{t,i}^{(4)}$ are the five local risk-probability values. If a deployment does not use probability outputs, the same framework can still use the local predicted risk class and sensor features as the global model input.

The multi-node preparation step also uses node locations. Each sensing node has a known latitude and longitude. These coordinates are used to define the spatial relationship among nodes and to support map-based visualization. For graph-based global modeling, the node locations are converted into an adjacency structure:

$$\mathbf{A} \in \mathbb{R}^{N \times N}, \quad (7.4)$$

where $(\mathbf{A})$ represents spatial connections among nodes. The adjacency structure can be based on geographic proximity, predefined neighbor relationships, or a distance-weighted graph. A normalized graph matrix can be written as

$$\tilde{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}(\mathbf{A} + \mathbf{I})\mathbf{D}^{-\frac{1}{2}}, \quad (7.5)$$

where $\mathbf{I}$ is the identity matrix and $\mathbf{D}$ is the degree matrix of $\mathbf{A} + \mathbf{I}$. This normalized graph representation allows the model to combine information from each node and its spatial neighbors. This preparation process converts raw stored data into model-ready input. The original CSV records remain useful for inspection, dashboard plotting, export, and future model retraining, while the processed multi-node sequences are used for global inference. If additional nodes are deployed later, the same data-preparation logic can be extended by updating the node list, node locations, and adjacency structure.

7.3 Spatiotemporal Global Model and Warning Analysis

The global prediction model is designed to learn both spatial and temporal relationships. Spatial relationships describe how risk at one node is related to risk at nearby nodes. Temporal relationships describe how recent sensor and risk patterns evolve over time. To capture both types of information, the implemented global model follows a spatiotemporal deep-learning structure that combines graph-based spatial processing, temporal sequence modeling, and a prediction head.

The current model structure can be summarized as follows. **1)** The input is a multi-node temporal feature sequence. **2)** Graph convolution layers process the node features according to the spatial adjacency structure. **3)** A temporal modeling layer, such as a GRU, captures how the spatially processed node representations change over the input window. **4)** A fully connected prediction head outputs future risk estimates for each node. This design allows the server to produce network-level predictions rather than only displaying current node readings.

Figure 7.2 shows the global DNN model structure. The figure emphasizes that the input is not a single-node record, but a multi-node sequence containing sensor features and local risk outputs. The graph convolution layers model spatial relationships among nodes, the temporal layer models recent trends, and the prediction head generates future node-level risk outputs.

For a given time step (t), the graph convolution operation can be expressed as

$$\mathbf{H}_t^{(1)} = \sigma \left(\tilde{\mathbf{A}} \mathbf{X}_t \mathbf{W}_{g1} \right), \quad (7.6)$$

$$\mathbf{H}_t^{(2)} = \sigma \left(\tilde{\mathbf{A}} \mathbf{H}_t^{(1)} \mathbf{W}_{g2} \right), \quad (7.7)$$

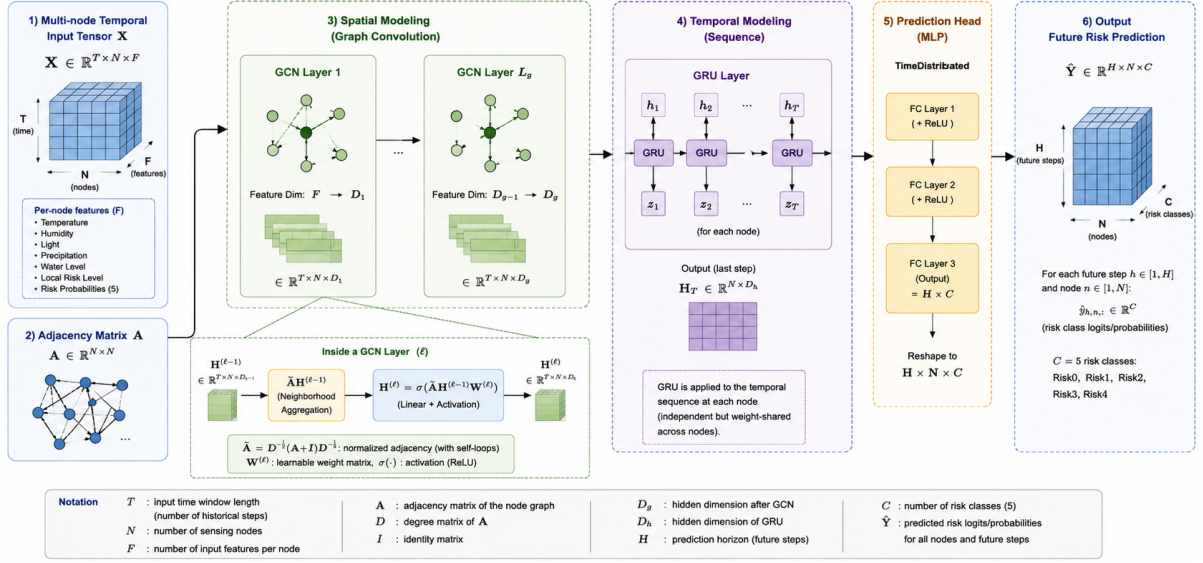


Figure 7.2: Spatiotemporal global DNN structure.

where $\mathbf{X}_t \in \mathbb{R}^{N \times F}$ is the node-feature matrix at time step t , $\tilde{\mathbf{A}}$ is the normalized adjacency matrix, $\mathbf{W}_{g1}$ and $\mathbf{W}_{g2}$ are trainable graph-convolution weights, $\sigma(\cdot)$ is a nonlinear activation function, and $\mathbf{H}_t^{(1)}$ and $\mathbf{H}_t^{(2)}$ are spatially processed node representations.

The temporal model receives the sequence of spatially processed node representations and updates a hidden state across the input window. For node (i), this can be written as

$$\mathbf{s}_{t,i} = \text{GRU} \left(\mathbf{H}_{t,i}^{(2)}, \mathbf{s}_{t-1,i} \right), \quad (7.8)$$

where $\mathbf{H}_{t,i}^{(2)}$ is the graph-processed representation of node i at time t , and $\mathbf{s}_{t,i}$ is the temporal hidden state. After the final input time step, the prediction head maps the hidden state to future risk logits:

$$\mathbf{z}_{h,i} = \text{MLP}_{\theta} (\mathbf{s}_{T,i}), \quad h = 1, \dots, H, \quad (7.9)$$

where H is the prediction horizon and $\mathbf{z}_{h,i} \in \mathbb{R}^C$ contains class logits for node i at future step h . In this project, $C = 5$ corresponds to Risk0–Risk4.

The overall global prediction function can therefore be represented as

$$\hat{\mathbf{Y}} = f_{\theta}(\mathbf{X}, \mathbf{A}), \quad (7.10)$$

where $f_{\theta}(\cdot)$ is the trained global DNN, $\mathbf{X}$ is the time-aligned multi-node input sequence, $\mathbf{A}$ is the spatial adjacency matrix, and $\hat{\mathbf{Y}}$ is the predicted future risk output. If the model predicts H future steps and C risk classes, the output can be represented as

$$\hat{\mathbf{Y}} \in \mathbb{R}^{H \times N \times C}. \quad (7.11)$$

Each node therefore receives a future risk prediction for each prediction step.

The model is trained using time-aligned historical sequences and future node-risk labels. A common training objective is multi-class cross-entropy over the prediction horizon and nodes:

$$\mathcal{L} = -\frac{1}{BHN} \sum_{b=1}^B \sum_{h=1}^H \sum_{i=1}^N \log(p_{b,h,i,y_{b,h,i}}), \quad (7.12)$$

where $p_{b,h,i,y_{b,h,i}}$ is the predicted probability assigned to the correct risk class $y_{b,h,i}$ for batch sample b , future step h , and node i . After training, the learned global model is executed on the server for warning analysis.

The dashboard-based warning analysis uses this global model as a server-side function. When the user selects a time window and triggers warning analysis, the server extracts the corresponding stored data, constructs the multi-node input sequence, runs the global model, and returns predicted risk levels. This allows the dashboard to support both current monitoring and retrospective analysis. For example, an older time window can be selected to replay the system behavior and inspect what the model would predict from that window.

The global model should be interpreted as a data-driven warning-analysis component, not as a full hydraulic simulation. It learns patterns from the available sensor and risk data and produces risk-level predictions over the node network. The predicted outputs should therefore be used as decision-support information together with local observations, official weather and flood information, and operator judgment.

7.4 Flood-Path Visualization

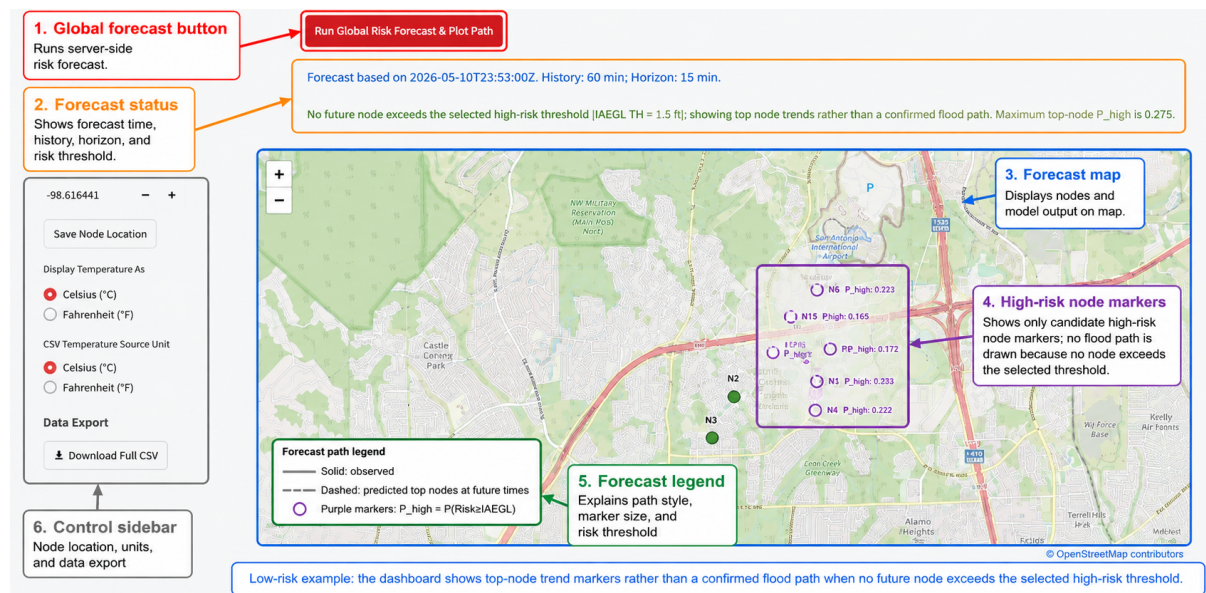


Figure 7.3: Flood-path visualization showing predicted node-level risk and possible risk-propagation direction across the sensor network.

Flood-path visualization translates the global model output into an interpretable spatial display. The goal is to help users understand not only which nodes are predicted to have higher risk, but also how risk may progress across the node network. The visualization uses node locations, predicted node risks, and spatial relationships among nodes to display risk patterns on a map or graph.

Figure 7.3 shows the flood-path visualization concept. Nodes are displayed using their geographic positions, and their colors or markers represent predicted risk levels. A path or directional indicator can be drawn between nodes to show possible risk propagation. In this context, the path should be interpreted as a risk-propagation visualization based on model prediction and node adjacency, not as a physically exact water-flow channel. This distinction is important

because the visualization summarizes predicted risk movement across discrete sensor locations rather than solving a detailed hydrodynamic flow model.

The visualization can include several elements. **1)** Node markers show the locations of the deployed sensing nodes. **2)** Node color or size indicates the predicted flood-risk level. **3)** High-risk nodes can be highlighted to draw user attention. **4)** Directional arrows or connecting lines can show possible risk propagation across adjacent or strongly related nodes. **5)** The selected time window and prediction horizon can be displayed so users understand which data were used and what future interval is being predicted.

The weighted connections in the visualization should be understood carefully. A weighted line does not necessarily mean a physical pipe, drainage path, or exact water channel. Instead, it represents a spatial or model-derived relationship between nodes. Depending on the implementation, the weight can reflect geographic distance, adjacency, predicted risk transition strength, or another risk-propagation score. For the final system, the line should connect actual node locations so that users can clearly see how the predicted path relates to the sensor network. The flood-path visualization supports practical interpretation of global warning results. Local officials or operators can quickly identify which node areas show higher predicted risk and whether the risk appears isolated or part of a broader spatial pattern. The visualization is most useful when combined with the dashboard time-series plots and current sensor readings, because the map shows spatial context while the plots show how conditions are changing over time.

7.5 Summary

This chapter described the global warning analysis and flood-path visualization framework. The server uses structured CSV records, node locations, local risk outputs, and sensor-derived features to construct multi-node temporal input sequences. The global model uses graph-based spatial modeling and temporal sequence modeling to predict future node-level risks. The dashboard warning-analysis function applies this model to selected data windows and returns predicted risks for visualization. The flood-path visualization then displays node-level risk patterns and possible risk-propagation direction across the sensor network. Together, these functions extend the system from local embedded risk prediction to network-level flood-warning support.

8. Prototype Validation and Deliverables

This chapter summarizes the prototype validation and project deliverables of the smart and self-sustaining coastal flood early-warning system. The purpose of this chapter is not to present a full large-scale field evaluation, but to document the validation activities that were completed for the available prototype. These activities include hardware power observation, data-transfer testing, LoRa communication-range testing, server/dashboard workflow validation, local MLP model evaluation, and global model development.

The validation scope reflects the current prototype stage. The field sensing node hardware, power subsystem, communication path, Raspberry Pi 4B server, CSV storage, and dashboard were tested as an integrated prototype. The local MLP model was evaluated using the prepared dataset and integrated into the MSP430 firmware for embedded inference. The global DNN model was developed and integrated into the server-side warning-analysis workflow, but full physical multi-node field validation was limited by the number of available deployed nodes. Therefore, this chapter clearly separates validated prototype functions from items that remain future work.

8.1 Validation Scope

The prototype validation was organized into two main categories: hardware validation and software/data-workflow validation. Hardware validation focused on the physical sensing node, energy-harvesting power behavior, and LoRa communication range. Software and data-workflow validation focused on data transfer, server parsing, CSV storage, dashboard display, local MLP evaluation, and global model integration.

This scope is appropriate for the current project stage. The completed tests demonstrate that the prototype can collect sensor data, sustain operation under the tested power conditions, transfer node records through the LoRa communication path, store data on the server, display the results on the dashboard, and execute local and global risk-analysis workflows. However, the validation does not claim to be a full long-term outdoor deployment study or a complete physical multi-node flood-event evaluation. Those items are identified as future work.

8.2 Hardware Validation

The hardware validation focused on whether the assembled field sensing node could support sustainable operation, provide reasonable sensor readings, tolerate outdoor wet conditions, and maintain LoRa communication over practical distances. The validation was performed at the prototype level using direct observation, comparison with commercial sensors, wet-exposure checks, and urban-range communication tests.

- **Power sustainability.** The node voltage was checked sporadically every few days during prototype operation. The capacitor voltage generally ranged from approximately 3.8 V to 4.1 V, depending on weather conditions and solar intensity. No average long-term voltage drop was observed under the tested workload. This indicates that the solar panel, BQ25570-based power-management circuit, local energy storage, and regulated stable 3.3 V supply can support the current prototype operation.

- **Temperature and humidity sensing.** The DHT22/AM2302 temperature and humidity readings were compared with commercial sensor readings. The observed difference was less than 5%, which is acceptable for the prototype-level environmental monitoring function used in this project.
- **Weatherproof enclosure.** The weatherproof box was intentionally tested using hose spraying and was also exposed to heavy rain events. No functional issue was observed after these wet-exposure conditions. This result supports the suitability of the enclosure design for protecting the main electronics during outdoor-oriented operation.
- **Externally mounted precipitation and water-level sensors.** The LM393-based precipitation sensor and the FS-IR02B optical water-level sensors were externally mounted using 3D-printed casing structures and a PVC pipe. The sensor wiring was sealed for water-exposure conditions. During the rainy season, the precipitation and optical water-level sensing components functioned properly, indicating that the external mounting and sealing approach was effective for the prototype.
- **Integrated field-node operation.** The MSP430, ESP32/LoRa module, environmental sensors, optical water-level sensors, energy-harvesting subsystem, antenna, enclosure, and external sensing structures were tested together as an integrated field sensing node. The prototype operated as a complete hardware platform for sensing, local processing, and wireless transmission.
- **LoRa communication range.** The LoRa communication range was tested in complex urban environments at Texas A&M University-Corpus Christi and The University of Texas at San Antonio. During these tests, the receiver was placed inside a building, which created a more challenging non-line-of-sight communication condition. Even under these non-ideal campus and urban conditions, the communication distance exceeded 1 km. This supports the feasibility of the selected LoRa configuration for campus-scale or neighborhood-scale monitoring, although the final range depends on antenna placement, building obstruction, terrain, vegetation, and receiver location.

8.3 Software and Data-Workflow Validation

The software and data-workflow validation focused on whether the sensing records generated by the MSP430 could be reliably transferred, stored, displayed, and used for risk prediction. Unlike isolated unit testing of each software module, the validation emphasized the complete data path from the embedded node to the server-side CSV records. This approach is appropriate because UART transfer, LoRa transmission, receiver forwarding, packet parsing, and CSV storage are closely connected in the prototype workflow.

- **UART and LoRa data-transfer workflow.** The MSP430-to-ESP32 UART baud rate was configured consistently between the embedded controller and the ESP32/LoRa module. The node power and voltage levels were also stable during operation, reducing the likelihood of communication errors caused by unstable supply conditions. During data collection, selected MSP430-side local records were compared with the corresponding records stored in the server-side CSV files after UART transfer, LoRa transmission, receiver forwarding, and server parsing. No mismatch was observed in these selective checks. This result supports that the UART transfer, LoRa link, receiver forwarding, and CSV storage path operated correctly for the tested data-collection workflow.

- **Configurable transmission setting.** The current transmission interval was selected to support dataset construction and continuous record collection. This interval is not a fixed system limitation. For deployment-time inference, the transmission interval can be adjusted according to warning latency, harvested energy availability, communication reliability, and server storage requirements. A shorter interval provides more frequent updates, while a longer interval reduces communication energy and stored data volume.
- **Server parsing and CSV storage.** The server-side software received the forwarded data stream from the LoRa receiver, parsed the incoming records, decoded the sensor fields, attached timing information, and stored the resulting values in CSV format. The stored CSV records provide the common data layer for dashboard visualization, export, long-term dataset construction, local-model improvement, global-model development, and later risk-analysis workflows.
- **Long-term dataset collection.** The server-side workflow was used to build a long-term sensing dataset over approximately 17 months. Although the actual sensing rate of the prototype was higher than one sample per minute, the raw sensing records were aggregated into one-minute samples to provide a consistent time-series dataset. Each one-minute sample represents the averaged sensor behavior within that minute, which reduces short-term noise and makes the data easier to use for dashboard visualization, local MLP training, and later global model development. The resulting dataset contains temperature, humidity, light intensity, precipitation, water-level state, and local prediction fields stored in CSV format.
- **Dashboard workflow.** The dashboard was validated as the user-facing view of the stored server data. It loaded the CSV records and displayed node location, current sensor values, selected time-window plots, water-level state, local risk output, and data export options. Since the dashboard figures have already been described in Chapter 6, the purpose here is not to repeat the interface description, but to confirm that the dashboard was connected to the server-side data workflow and could display the stored records for monitoring.
- **Local MLP evaluation and embedded deployment.** The local MLP model was evaluated using the prepared dataset before embedded deployment. The model uses a compact 16–16–8–5 architecture for five-class flood-risk prediction. After training and validation, the learned weights were exported into embedded C format and integrated into the MSP430 firmware. This confirms that the local risk inference component can be executed on the low-power sensing node rather than only on the server.

Figure 8.1 shows the validation confusion matrix for the local MLP flood-risk classifier. The rows represent the true labels and the columns represent the predicted labels. The result shows strong diagonal dominance, indicating that most validation samples were classified into the correct risk classes. The overall validation accuracy is 98.82% over 139,951 samples. The class-level recalls are 0.992 for Risk0, 0.933 for Risk1, 0.947 for Risk2, 0.949 for Risk3, and 0.993 for Risk4. These results indicate that the local MLP can distinguish the five risk levels in the prepared validation dataset, although additional real storm and flooding data are still needed for broader field validation.

- **Global model development and integration.** The global DNN model was developed to support server-side multi-node warning analysis. It uses prepared multi-node data, local risk outputs, sensor features, and spatial node relationships to predict future node-level

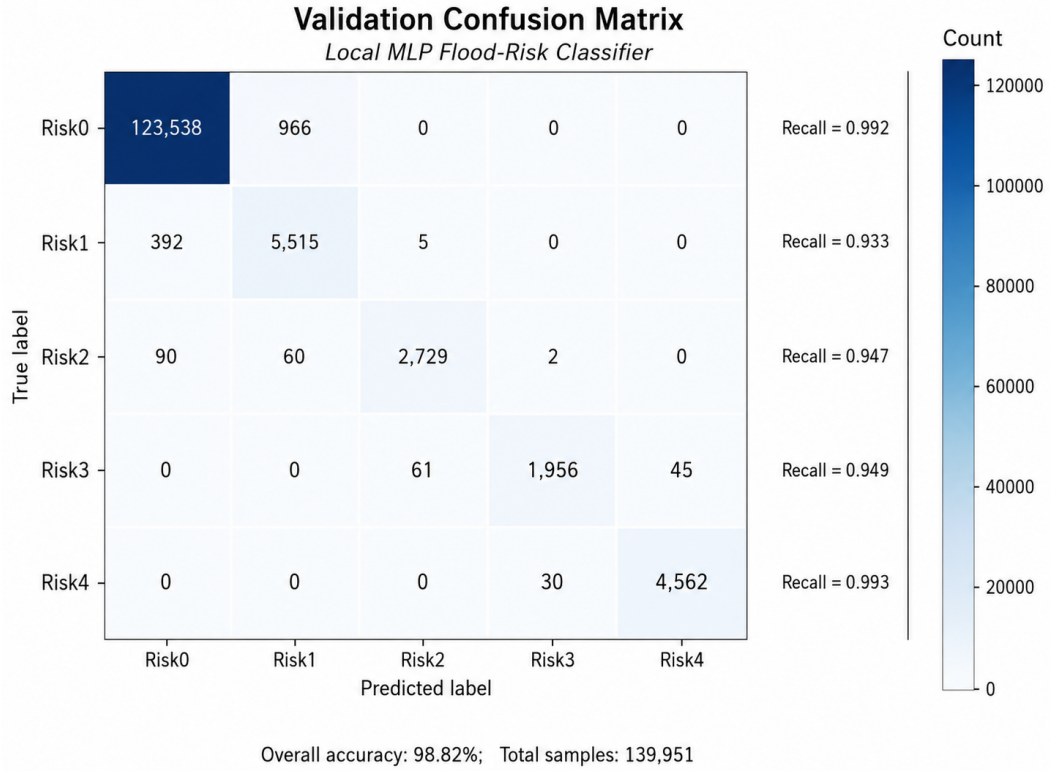


Figure 8.1: Validation confusion matrix of the local MLP flood-risk classifier.

risks. Because the available physical prototype did not include a full set of deployed field nodes, the global model was not fully validated through a real multi-node outdoor deployment. The current work demonstrates the software feasibility of the global warning-analysis workflow, while full physical multi-node validation remains future work.

8.4 Project Deliverables

The project completed the major technical deliverables required for the smart and self-sustaining coastal flood early-warning prototype. These deliverables cover the sensing-node hardware, energy-harvesting subsystem, long-range wireless communication, Raspberry Pi 4B gateway server, embedded firmware, server-side dashboard, local MLP inference, global warning-analysis workflow, and flood-path visualization. Administrative reporting and project closeout documents are not listed in this section because they belong to the project-management record rather than the technical implementation summary. Together, these deliverables demonstrate a complete prototype workflow from field sensing to server-side warning support. The field node collects environmental and water-level data, performs local risk inference, and transfers node records through LoRa. The LoRa receiver and Raspberry Pi 4B gateway/server receive, parse, store, and visualize the data. The dashboard provides local monitoring, data export, global warning analysis, and flood-path visualization. The completed prototype therefore satisfies the technical scope of sensing-module design, embedded TinyML inference, long-range wireless communication, server-side software, and AI-assisted flood-warning analysis.

Table 8.1: Summary of completed technical deliverables.

Deliverable	Completed outcome	Status
Selected sensing features	Temperature, humidity, precipitation, water level, and light intensity were integrated for flood-risk prediction.	✓
Field sensing node	MSP430FR5994 node, sensors, ESP32/LoRa module, antenna, enclosure, and external sensing structures were assembled.	✓
Energy-harvesting subsystem	Solar panel, BQ25570 power stage, local storage, and regulated 3.3 V supply were integrated.	✓
LoRa gateway and receiver	Heltec ESP32 V3 LoRa receiver and Raspberry Pi 4B gateway/server were prepared for receiving field-node data.	✓
Embedded sensing software	MSP430 firmware was completed for sensing, sensor decoding, aggregation, local inference, and ESP32 wakeup control.	✓
Local TinyML model	A compact 16–16–8–5 MLP was trained, evaluated, exported, and integrated into the MSP430 firmware.	✓
Model-rationale visualization	Saliency analysis was generated to show the importance of local MLP input features.	✓
Server and dashboard software	Serial ingestion, packet parsing, CSV storage, dashboard visualization, risk display, and data export were completed.	✓
Global warning model	Server-side global DNN workflow was developed for multi-node spatiotemporal warning analysis.	✓
Flood-path visualization	Dashboard-based global forecast and flood-path visualization functions were implemented.	✓
Technical documentation	Hardware, firmware, communication, server, dashboard, model workflow, validation, and maintenance guidance were documented.	✓

8.5 Limitations and Future Improvements

The prototype demonstrates the feasibility of an integrated flood early-warning system, but several limitations remain. These limitations are important to state clearly because the current work is a prototype-level implementation rather than a full long-term operational deployment.

Scale of physical deployment. The current system validates the sensing-node hardware, LoRa communication path, Raspberry Pi 4B gateway/server, dashboard, local MLP inference, and global warning-analysis workflow. However, the physical deployment did not include a full set of many field nodes operating simultaneously over a large flood-prone area. Therefore, the global model was developed and integrated using the prepared multi-node data workflow, but it still requires future validation with multiple physical nodes deployed at different locations, elevations, and drainage conditions.

Availability of real high-risk flood data. The collected dataset contains approximately 17 months of one-minute-resolution sensing records, which is valuable for system development and model training. However, real high-risk flooding events are naturally less frequent than normal low-risk conditions. This creates data imbalance and limits the amount of real high-risk evidence available for training and validating both local and global prediction models. Future work should continue data collection during additional rainy seasons and storm events, with particular attention to obtaining more Risk1–Risk4 samples.

Long-term outdoor robustness. The enclosure was tested using hose spraying and heavy rain exposure, and no functional issue was observed during the tested conditions. The externally mounted precipitation and water-level sensors also functioned properly during the rainy season with 3D-printed holders, PVC mounting, and sealed wiring. Nevertheless, longer-term deployment is needed to evaluate enclosure sealing, cable strain relief, connector corrosion, sensor aging, solar-panel contamination, battery health, and seasonal variation in solar-energy availability.

Transmission interval optimization. The current transmission setting was selected to support dataset construction and continuous data collection. For operational deployment, the transmission interval should be optimized based on warning latency, harvested energy availability, network reliability, and storage requirements. Adaptive transmission policies could also be explored, where the node sends data more frequently during rain or rising-water conditions and reduces communication frequency during stable low-risk periods.

Global model field validation. The server-side global DNN and flood-path visualization were implemented to support multi-node spatiotemporal warning analysis. However, because the current physical deployment was constrained by available hardware nodes, the global model has not yet been fully validated using simultaneous real measurements from many deployed nodes during flooding events. Future work should perform multi-node field tests and compare predicted risk propagation with observed flood progression.

User-centered evaluation. The dashboard provides map display, current sensor values, time-window plots, local risk prediction, data export, global warning analysis, and flood-path visualization. Future work should evaluate the interface with local officials, emergency managers, or field operators to determine whether the risk labels, plots, map markers, and warning messages are easy to interpret and useful for decision support. This feedback can guide improvements to the non-technical workflow and maintenance procedures described in Chapter 9.

8.6 Summary

This chapter summarized the prototype validation and completed technical deliverables. Hardware validation showed that the field sensing node could operate under the tested workload with stable power behavior. The capacitor voltage was checked sporadically every few days and generally ranged from 3.8 V to 4.1 V depending on weather and solar intensity, with no average long-term voltage drop observed. Sensor validation showed that the temperature and humidity readings were within 5% of commercial sensor readings. The weatherproof enclosure was tested through hose spraying and heavy rain exposure, and no functional issue was observed. The externally mounted precipitation and optical water-level sensors, supported by 3D-printed casing, PVC mounting, and sealed wiring, functioned properly during rainy conditions. LoRa range testing at TAMUCC and UTSA further showed communication distances exceeding 1 km in complex urban environments, even with the receiver placed inside a building.

Software and data-workflow validation confirmed the end-to-end data path from MSP430 sensing records to server-side CSV storage and dashboard display. During data collection, selected MSP430-side local records were compared with the corresponding records stored in the server CSV files after UART transfer, LoRa transmission, receiver forwarding, and server parsing, and no mismatch was observed. The prototype also produced approximately 17 months of CSV-based sensing records organized at one-minute resolution by averaging higher-rate raw readings within each minute. The local MLP model was evaluated using the prepared dataset and deployed for MSP430-side inference, with the validation confusion matrix showing strong five-class classification performance. The global DNN model and flood-path visualization workflow were developed and integrated into the server-side framework, while full physical multi-node validation remains future work due to hardware-node constraints. The next chapter provides non-technical guidance and long-term maintenance recommendations for local officials and system operators.

9. Non-Technical Guidance and Long-Term Maintenance

This chapter provides non-technical guidance for local officials, emergency managers, field operators, and maintenance staff who may use or maintain the coastal flood early-warning system. The goal is to explain how the system should be interpreted, how it can support decision making, and what maintenance practices are needed to keep it reliable over time.

The system should be treated as a decision-support tool. It provides local sensing information, water-level status, recent sensor trends, local flood-risk prediction, and server-side warning-analysis outputs. However, it does not replace official weather services, emergency management procedures, field inspection, or public safety decisions. Its main purpose is to improve awareness of changing local conditions and help users identify when closer inspection or additional response actions may be needed.

9.1 Purpose of the Early-Warning System

The early-warning system monitors environmental and water-related conditions near flood-prone areas. The field sensing node collects temperature, humidity, light intensity, precipitation, and water-level information. These data are transmitted wirelessly to the gateway and server, where they are stored, displayed, and analyzed.

The system is designed to provide early awareness before flooding becomes severe. Instead of relying only on visible flooding or delayed reports, users can observe changes in rainfall, water level, and predicted risk through the dashboard. The local prediction output provides a node-level risk estimate, while the global warning-analysis function provides a broader view of possible risk patterns across multiple sensing locations.

The system is most useful when used together with other information sources, such as local weather forecasts, field reports, drainage conditions, known flood-prone areas, and official emergency guidance. When the dashboard shows increasing risk, users should treat it as a signal to check conditions more carefully, not as the only basis for a public safety decision.

9.2 Interpreting Dashboard Information

The dashboard is the main user interface of the system. It allows users to view the sensing node location, current sensor values, recent sensor trends, water-level status, local risk prediction, global forecast output, flood-path visualization, and data export function without directly inspecting raw CSV files or communication messages.

The map shows where the sensing node is located, so users can connect the displayed readings to a physical monitoring site. The current-value area shows the latest temperature, humidity, light intensity, precipitation, and water-level readings. These values provide a quick check of the current local condition. The time-window plots show how sensor readings have changed over a selected period, which is important because flood risk is often related to trends rather than one isolated reading.

The water-level display provides direct local evidence from the optical water sensors. The local risk prediction shows the embedded model output for the monitored node. The global forecast and flood-path visualization, when available, provide additional server-side warning information based on multi-node data. These outputs should be interpreted as decision-support

information. A displayed path or direction indicates a predicted risk-propagation pattern across sensor locations, not an exact hydraulic water-flow route.

The data export function allows stored records to be downloaded for backup, reporting, model updates, or later analysis. This is useful after rain or flood-related events because the exported data can support review and future improvement of the system.

9.3 Understanding Risk Levels and Warning Outputs

The system uses five risk levels, from Risk0 to Risk4. Risk0 represents normal or low-risk conditions. Risk1 indicates slightly elevated concern. Risk2 indicates moderate concern and should lead users to review recent rainfall and water-level trends more carefully. Risk3 indicates high concern and should prompt site inspection or notification of responsible personnel. Risk4 indicates very high concern and should be treated as a strong warning indicator that requires immediate verification and response according to local procedures.

These risk levels should not be interpreted as absolute guarantees. A low-risk output does not guarantee that flooding cannot occur, and a high-risk output does not automatically mean that flooding has already occurred. Instead, the risk level should be used as a warning indicator that helps users decide when additional inspection, monitoring, or response is needed.

The global forecast should also be interpreted carefully. If no node exceeds the selected high-risk threshold, the dashboard may show candidate high-risk node markers without drawing a confirmed flood path. This behavior is intentional because the system should not visually overstate a flood path when the model output does not support a strong warning.

9.4 Recommended Operational Workflow

During normal conditions, users should check the dashboard periodically to confirm that recent data are being updated and that sensor readings appear reasonable. During rainfall or potential flooding conditions, users should check the dashboard more frequently and pay close attention to precipitation, water-level status, local risk prediction, and global forecast outputs.

A simple operational workflow is recommended:

1. Confirm that the dashboard is receiving recent data and that the node location is correct.
2. Review current precipitation, water-level status, and recent sensor trends.
3. Check the local risk level and, when available, the global forecast output.
4. If Risk2 or higher persists, consider field inspection or internal notification.
5. If Risk3 or Risk4 appears, compare the system output with official weather information and follow local emergency procedures.
6. After important rain or flood-related events, export and back up the relevant data.

This workflow can be adjusted based on each local agency's operating procedures, staffing, flood-risk tolerance, and emergency response responsibilities.

9.5 Routine Maintenance

Routine maintenance is needed because the system includes outdoor hardware, exposed sensors, solar power components, wireless communication equipment, and server-side software. The most important maintenance goal is to keep the sensing node powered, the sensors exposed and clean, the communication path active, and the server data storage functioning.

The solar panel should be checked regularly and after storms. Dust, leaves, bird droppings, or debris should be removed, and the panel should not be blocked by vegetation or nearby objects. The enclosure should also be inspected after heavy rain to check for water intrusion, loose covers, damaged seals, or physical damage. Cable entry points should remain sealed, and cables should be strain-relieved to reduce damage from wind, handling, or vibration.

The precipitation sensor and optical water-level sensors should be inspected regularly because they are externally exposed. The precipitation sensor surface should be kept clean and open to rainfall. The optical water-level sensors should remain aligned at their intended height levels, and their surfaces should be cleaned if dirt or debris accumulates. The PVC support and 3D-printed holders should remain fixed and undamaged.

The LoRa antenna should be checked to make sure it is connected, upright, and not blocked. The gateway and Raspberry Pi 4B server should be checked to confirm that the LoRa receiver is connected, the server is running, data are being stored, and the dashboard can be opened. CSV records should be backed up regularly, especially after important storm events.

A practical maintenance schedule is to check the dashboard and server weekly, inspect the field node and sensors monthly, and perform additional checks after heavy rain, flooding, strong wind, or physical disturbance near the node.

9.6 Troubleshooting Common Issues

If no recent data appear on the dashboard, users should first check the most likely causes: node power, LoRa receiver connection, Raspberry Pi power, and server status. If the field node has low voltage, the solar panel, power wiring, and local storage should be inspected. If the receiver is disconnected or the server has stopped, the USB connection and server application should be restarted by trained personnel.

If a sensor value looks abnormal, the sensor should be inspected before assuming the model is wrong. A dirty precipitation sensor, blocked optical water-level sensor, loose cable, or water-exposed connector can cause incorrect readings. The exposed sensor surfaces should be cleaned, and cable connections should be checked.

If the dashboard opens but plots are empty, the issue may be related to missing CSV data, a stopped parser, or a selected time window that contains no records. Users should check whether recent CSV records exist and try a different time range. If the local risk output shows “not ready,” the model may not yet have enough recent history to make a valid prediction. In that case, users should wait for more samples and confirm that data are being stored.

If the global forecast does not draw a flood path, this may simply mean that no node exceeds the selected high-risk threshold. The dashboard may still show candidate high-risk node markers or low-confidence trends. This should be interpreted as a low-risk or uncertain forecast condition rather than an automatic system failure.

When routine checks do not solve the problem, technical staff should review the firmware, receiver code, server logs, CSV files, and model configuration. Routine operators should not modify embedded firmware or model files unless they have been trained to do so.

9.7 Software Updates and Model Retraining

Normal system operators do not need to modify the MSP430 firmware, ESP32 code, server code, or machine-learning models during routine monitoring. Software updates and model retraining should be performed by trained technical staff.

As more rainy-season and storm-event data are collected, the local MLP and global DNN models should be periodically reviewed and retrained. The CSV records stored on the server provide the main data source for future model updates. After retraining, the local model can be exported and programmed into the MSP430 firmware, while the global model can be updated on the Raspberry Pi 4B or another Ubuntu/Linux server.

Model updates should be tested offline before being deployed to the field system. The technical team should keep backup copies of previous firmware, model files, server code, and CSV datasets so that the system can be restored if a new update causes unexpected behavior. Detailed software-update steps, such as updating MSP430 firmware using Code Composer Studio, updating ESP32 code, running server scripts, retraining models in VS Code, or migrating the server to another Raspberry Pi, should be maintained in a separate technical appendix or implementation note.

9.8 Long-Term Deployment Recommendations

For long-term deployment, node locations should be selected based on local flood history, drainage patterns, low-lying areas, maintenance access, and communication coverage. A single node provides useful local information, but multiple nodes are preferred because they allow the system to compare conditions across space and support global warning analysis.

The receiver should be placed where it has good communication visibility to the field nodes. A higher or less obstructed receiver location can improve reliability. If the receiver must be placed inside a building, site-specific range testing should be performed because walls and other obstructions can reduce signal strength.

Data management should also be planned for long-term operation. CSV records should be backed up regularly, and important storm-event data should be saved for future review. After major rainfall or flood-related events, operators should compare dashboard readings, model outputs, field observations, and official reports. These reviews can help determine whether sensor placement, risk thresholds, maintenance procedures, or model behavior should be adjusted. Over time, the system can become more useful as more data are collected, more nodes are deployed, and the models are updated with additional real-world observations.

9.9 Summary

This chapter provided non-technical guidance for using and maintaining the coastal flood early-warning system. The system is intended to support early awareness by showing local sensor readings, water-level status, risk predictions, recent trends, global warning outputs, and flood-path visualization. The risk levels should be treated as decision-support indicators rather than absolute guarantees.

Routine use should focus on checking whether recent data are available, reviewing precipitation and water-level trends, interpreting local and global risk outputs carefully, and following local procedures when risk increases. Long-term maintenance should include checking the solar panel, enclosure, exposed sensors, antenna, node voltage, gateway/server operation, dashboard

display, and CSV backups. Future improvements should include additional node deployment, continued data collection, periodic model retraining, and comparison with field observations and official flood information.

10. Outreach, Education, and Workforce Development

In addition to the technical development of the coastal flood early-warning prototype, the project supported student training, education integration, and future extension activities. These outcomes are important because the project combines embedded systems, wireless sensing, energy harvesting, edge artificial intelligence, server-side data processing, and decision-support visualization. The project therefore provided a practical training platform for students and created materials that can be reused in related courses and outreach activities.

A total of five graduate students and three undergraduate students were supported by the project at different stages. Student activities included hardware assembly, sensor integration, embedded firmware development, LoRa communication testing, data collection, server-side software development, dashboard implementation, dataset preparation, local machine-learning model development, and global warning-analysis exploration. Through these activities, students gained hands-on experience with field sensing systems, low-power embedded platforms, wireless communication, data-driven modeling, and flood-monitoring applications.

The project outcomes were also integrated into the PI's teaching activities. Concepts and implementation examples from the project were incorporated into courses such as Embedded System Design and AI Hardware and Programming. In the embedded-systems context, the project provided examples of microcontroller-based sensing, ADC sampling, GPIO-based water-level reading, UART communication, low-power operation, and wireless data transfer. In the AI-hardware and programming context, the project provided examples of TinyML deployment, sensor-feature construction, model evaluation, embedded inference, edge-server processing, and the relationship between local and global prediction models.

The project also provides a foundation for future outreach and extension activities. The assembled sensing node, LoRa gateway, Raspberry Pi server, dashboard, and flood-path visualization can be used as a demonstration platform for explaining how smart sensing and edge intelligence can support flood awareness. For non-technical audiences, the dashboard and maintenance guidance can help explain how local sensor readings, water-level changes, and risk predictions are used for decision support. For technical audiences, the hardware and software workflow can demonstrate the full pipeline from field sensing to embedded inference, wireless communication, server-side storage, and warning visualization.

Overall, the project contributed not only a working prototype, but also an educational and workforce-development platform. The student training and course integration activities help broaden the impact of the project and support future development of embedded AI and smart environmental monitoring systems.

A. Technical Operation and Software Update Notes

This appendix provides technical notes for maintaining and updating the prototype software. It is intended for trained technical staff, students, or future project maintainers. Routine operators do not need to follow these steps during normal system monitoring. Before changing firmware, server code, or model files, the current working version should always be backed up.

A.1 Software Components

The prototype software includes four main components. The MSP430 firmware runs on the MSP430FR5994 sensing node and controls sensor acquisition, water-level reading, aggregation-window processing, local MLP inference, and ESP32 wakeup coordination. The ESP32 transmitter code runs on the LoRa board attached to the field sensing node and sends the MSP430 node record through LoRa. The ESP32 receiver code runs on the Heltec ESP32 V3 LoRa receiver at the gateway and forwards received packets to the Raspberry Pi 4B through USB serial. The Raspberry Pi server code reads the USB serial stream, parses packets, stores decoded records in CSV format, hosts the dashboard, and supports warning-analysis functions.

These components are connected through a common data format. If the node payload format is changed in the MSP430 firmware or ESP32 transmitter code, the receiver and server parser must also be checked so that the stored CSV records remain correct.

A.2 Updating the MSP430 Firmware

The MSP430 firmware is updated using Code Composer Studio (CCS). Before making changes, the current firmware project should be backed up, including source files, header files, model-weight files, and configuration files.

In CCS, the technical staff should open the MSP430FR5994 project, confirm the target device, check the sensor pin configuration, and verify that the UART, ESP32 wakeup pin, and local MLP weight files match the current hardware. If the local MLP model has been retrained, the old embedded weight files should be replaced with the new verified files. The project should then be rebuilt and programmed to the MSP430 board.

After programming, a short validation should be performed before returning the node to normal operation. The node should read the sensors, create a valid node record, wake the ESP32, and transmit the record. Selected MSP430-side records should be compared with the final CSV records stored on the server to confirm that the sensing and communication path still works correctly.

A.3 Updating ESP32 LoRa Code Using Arduino IDE

The ESP32 transmitter and receiver programs are updated using Arduino IDE. The appropriate ESP32 board support package and Heltec ESP32 board settings should be installed before programming. The correct board, serial port, LoRa frequency, UART baud rate, bandwidth, spreading factor, coding rate, and preamble settings should be checked before uploading new code.

For the transmitter, the Arduino IDE sketch should receive the MSP430 UART record and transmit it through LoRa without changing the expected payload format. For the receiver, the Arduino IDE sketch should receive LoRa packets and forward them to the Raspberry Pi 4B through USB serial. After uploading either sketch, the technical staff should send several test records and confirm that the server stores the expected values in the CSV file. The transmitter and receiver settings must remain compatible. If the LoRa parameters or payload format are changed on one side, the other side and the server parser must also be updated.

A.4 Running the Raspberry Pi Server

The Raspberry Pi 4B acts as the gateway and edge server. The Heltec ESP32 V3 LoRa receiver is connected to the Raspberry Pi through USB. The server reads this USB serial stream and updates the CSV storage and dashboard. To start the server, open a terminal in the project folder and run:

```
./run.sh
```

If the script does not have execution permission, run:

```
chmod +x run.sh  
./run.sh
```

After the server starts, the dashboard should be opened in a browser to confirm that recent data are available. If no new records appear, the technical staff should check the USB serial port, receiver connection, server log, parser status, and recent CSV files.

A.5 CSV Data Backup and Dataset Management

The CSV files are important because they support dashboard visualization, data export, local MLP retraining, global model development, and report generation. The server data folder should be backed up regularly, especially after important rain or flood-related events.

The prototype dataset is organized at one-minute resolution. Although the field node can sense at a higher rate, raw readings within each minute are averaged to create one representative sample. If the sensing period, aggregation interval, or history length is changed later, the model-training configuration and embedded inference configuration should be updated consistently.

A.6 Retraining the Local MLP Model

The local MLP model can be retrained when additional rainy-season or storm-event data are collected. Retraining should be performed offline, for example in the Python/VS Code training environment, before any new model is deployed to the MSP430.

The new CSV data should first be checked for correct columns, time resolution, sensor representation, water-level format, and risk labels. The model should then be trained using the same feature-construction logic used by the MSP430 inference code. After training, the model should be evaluated using validation accuracy, confusion matrix, and per-class recall. This is important because flood-risk data can be imbalanced, and high overall accuracy alone may not guarantee good detection of higher-risk classes.

After the model is verified, the weights can be exported into embedded C files and integrated into the MSP430 firmware project. The updated firmware should be tested with known records before field use.

A.7 Updating the Global DNN Model

The global DNN model runs on the server and does not need to be programmed into the MSP430. It can be updated on the Raspberry Pi 4B or another Ubuntu/Linux computer. The update should use the prepared multi-node dataset, node coordinates, local risk outputs, sensor features, and adjacency structure described in Chapter 7.

Before replacing the deployed global model, the new model should be tested offline. The dashboard global forecast function should then be checked to confirm that predicted node risks, high-risk markers, and flood-path visualization still appear correctly. Because the current physical system has limited real multi-node deployment, global-model outputs should remain decision-support information until more physical multi-node field validation is completed.

A.8 Server Migration

The server can be moved to another Raspberry Pi or Ubuntu/Linux computer if the original Raspberry Pi fails or if more computing capacity is needed. The full project folder should be copied, including server code, data files, dashboard files, model files, and configuration files. The required software environment should then be recreated on the new machine.

After migration, the Heltec ESP32 V3 receiver should be connected through USB, and the USB serial device name should be checked. The run script can then be used to start the server. A successful migration should be confirmed by opening the dashboard, loading stored data, receiving a new test packet, and verifying that a new CSV record is created.

A.9 Version Control and Backup

Each working version of the firmware, ESP32 sketches, server code, local MLP model, global DNN model, and CSV dataset should be backed up with a clear date or version name. A simple version log should record what changed, why it changed, and how the new version was tested. A new model or firmware version should not overwrite a known working version until it has passed a short end-to-end test.

A practical final check after any update is to confirm that the node can sense, transmit, store, and display a new record on the dashboard. This end-to-end check helps ensure that firmware changes, ESP32 updates, server changes, and model updates have not broken the complete data path.

List of Figures

2.1	System Architecture Early-warning System.	8
2.2	Energy-harvesting & Regulated Power-flow Architecture of the Field Sensing Node.	11
3.1	Field Sensing Node Hardware Interfaces.	14
3.2	Sensor modules used in the field sensing node: light, precipitation, temperature/humidity, and optical water-level sensing.	15
3.3	Gateway and edge-server hardware setup.	18
3.4	Assembled field sensing node prototype.	19
3.5	Assembled gateway and edge-server hardware prototype with Raspberry Pi 4B and Heltec ESP32 V3 LoRa receiver connected through USB serial.	20
4.1	MSP430 embedded software workflow.	23
4.2	Embedded MLP architecture for local flood-risk prediction.	28
4.3	Saliency map of the local MLP input features.	29
5.1	Wireless communication and data transmission workflow.	32
6.1	Server-side data ingestion and storage workflow.	36
6.2	Web-based flood monitoring dashboard with annotated interface functions.	39
7.1	Global warning analysis workflow	42
7.2	Spatiotemporal global DNN structure.	44
7.3	Flood-path visualization showing predicted node-level risk and possible risk-propagation direction across the sensor network.	45
8.1	Validation confusion matrix of the local MLP flood-risk classifier.	50

REFERENCES

- [1] IPCC, *Climate Change 2022: Impacts, Adaptation and Vulnerability*. Cambridge, United Kingdom and New York, NY, USA: Cambridge University Press, 2022.
- [2] S. Hallegatte, C. Green, R. J. Nicholls, and J. Corfee-Morlot, “Future flood losses in major coastal cities,” *Nature Climate Change*, vol. 3, pp. 802–806, 2013.
- [3] J. Hinkel, D. Lincke, A. T. Vafeidis, M. Perrette, R. J. Nicholls, R. S. J. Tol, B. Marzeion, X. Fettweis, C. Ionescu, and A. Levermann, “Coastal flood damage and adaptation costs under 21st century sea-level rise,” *Proceedings of the National Academy of Sciences*, vol. 111, no. 9, pp. 3292–3297, 2014.
- [4] S. A. Kulp and B. H. Strauss, “New elevation data triple estimates of global vulnerability to sea-level rise and coastal flooding,” *Nature Communications*, vol. 10, no. 1, p. 4844, 2019.
- [5] C. T. Nguyen, S. Rhee, and J. D. Loftis, “StormSense: A blueprint for coastal flood forecast information and automated alert messaging systems,” in *Proceedings of the Third International Workshop on Science of Smart City Operations and Platforms Engineering (SCOPE)*, Porto, Portugal, 2018.
- [6] A. Chatzipavlis, D. Trogu, A. Ruju, J. Montes, A. Usai, M. Porta, G. Coco, S. De Muro, and P. Ciavola, “Development and validation of an early warning system for coastal flooding operating on a mediterranean urban beach,” *Natural Hazards and Earth System Sciences*, vol. 26, pp. 1573–1601, 2026.
- [7] B. Hassan, A. Karapetyan, A. C. H. Chow, and S. Madanat, “Climate adaptation-aware flood prediction for coastal cities using deep learning,” *Hydrology and Earth System Sciences*, vol. 30, pp. 1333–1358, 2026.
- [8] K. A. Islam, Z. Mehrab, M. Halappanavar, H. Mortveit, S. Katragadda, J. D. Loftis, and M. Marathe, “Denoising diffusion probabilistic models for coastal inundation forecasting,” 2025.
- [9] C. Pan, M. Xie, and J. Hu, “Enzyme: An energy-efficient transient computing paradigm for ultralow self-powered iot edge devices,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2440–2450, 2018.
- [10] M. Xie, C. Pan, M. Zhao, Y. Liu, C. J. Xue, and J. Hu, “Avoiding data inconsistency in energy harvesting powered embedded systems,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 23, no. 3, pp. 1–25, 2018.
- [11] C. Pan, M. Xie, and J. Hu, “Maximize energy utilization for ultra-low energy harvesting powered embedded systems,” in *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2017, pp. 1–6.
- [12] S. Priya and D. J. Inman, *Energy harvesting technologies*. Springer, 2009, vol. 21.

- [13] C. Pan, M. Xie, C. Yang, Y. Chen, and J. Hu, “Exploiting multiple write modes of non-volatile main memory in embedded systems,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 4, pp. 1–26, 2017.
- [14] M. Xie, M. Zhao, C. Pan, J. Hu, Y. Liu, and C. J. Xue, “Fixing the broken time machine: Consistency-aware checkpointing for energy harvesting powered non-volatile processor,” in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.
- [15] J. Hu, M. Xie, C. Pan, C. J. Xue, Q. Zhuge, and E. H.-M. Sha, “Low overhead software wear leveling for hybrid pcm+ dram main memory on embedded systems,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 4, pp. 654–663, 2014.
- [16] M. Xie, M. Zhao, C. Pan, H. Li, Y. Liu, Y. Zhang, C. J. Xue, and J. Hu, “Checkpoint aware hybrid cache architecture for nv processor in energy harvesting powered systems,” in *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 2016, pp. 1–10.
- [17] L. Benini, A. Bogliolo, and G. De Micheli, “A survey of design techniques for system-level dynamic power management,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 3, pp. 299–316, 2000.
- [18] A. Kansal, J. Hsu, S. Zahedi, and M. B. Srivastava, “Power management in energy harvesting sensor networks,” *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 4, p. 32, 2007.
- [19] B. Ransford, J. Sorber, and K. Fu, “Mementos: System support for long-running computation on rfid-scale devices,” in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, 2011, pp. 159–170.
- [20] A. Colin and B. Lucia, “Chain: Tasks and channels for reliable intermittent programs,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016, 2016, pp. 514–530.
- [21] J. Hester and J. Sorber, “Flicker: Rapid prototyping for the batteryless internet-of-things,” in *Proceedings of the 15th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys 2017, 2017, pp. 19:1–19:13.
- [22] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O’Reilly Media, 2019.
- [23] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang, P. Warden, and R. Rhodes, “Tensorflow lite micro: Embedded machine learning for tinyml systems,” in *Proceedings of Machine Learning and Systems*, vol. 3, 2021, pp. 800–811.
- [24] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, “MCUNet: Tiny deep learning on iot devices,” in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 11 711–11 722.

- [25] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau *et al.*, “MLPerf tiny benchmark,” *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.
- [26] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.
- [27] L. Lai, N. Suda, and V. Chandra, “CMSIS-NN: Efficient neural network kernels for arm cortex-m cpus,” 2018.
- [28] Texas Instruments, *MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers Datasheet*, Texas Instruments, 2016, rev. D.
- [29] ———, *bq25570 Nano Power Boost Charger and Buck Converter for Energy Harvester Powered Applications Datasheet*, Texas Instruments, 2019, rev. G.
- [30] Heltec Automation, *WiFi LoRa 32 V3 Documentation*, Heltec Automation, 2026, eSP32-S3 and SX1262 LoRa development board documentation.
- [31] Espressif Systems, *ESP32-S3 Series Datasheet*, Espressif Systems, 2026.
- [32] Semtech Corporation, *SX1261/SX1262 Datasheet*, Semtech Corporation, 2025, loRa Connect transceiver documentation.
- [33] J. Thielen, J. Bartholmes, M.-H. Ramos, and A. de Roo, “The european flood alert system – part 1: Concept and development,” *Hydrology and Earth System Sciences*, vol. 13, pp. 125–140, 2009.
- [34] L. Alfieri, F. Pappenberger, and F. Wetterhall, “The extreme runoff index for flood early warning in europe,” *Natural Hazards and Earth System Sciences*, vol. 14, pp. 1505–1515, 2014.
- [35] G. Schumann, P. D. Bates, M. S. Horritt, P. Matgen, and F. Pappenberger, “Progress in integration of remote sensing-derived flood extent and stage data and hydraulic models,” *Reviews of Geophysics*, vol. 47, no. 4, p. RG4001, 2009.
- [36] G. J.-P. Schumann, “Breakthroughs in satellite remote sensing of floods,” *Frontiers in Remote Sensing*, vol. 4, p. 1280654, 2024.
- [37] I. Gul, M. Abdelkader, K. Mackelburg, and M. Temimi, “Dual-purpose and low-cost iot-based sensor network for real-time rainfall and urban flood monitoring,” *IEEE Internet of Things Journal*, vol. 13, no. 4, pp. 6640–6654, 2026.
- [38] V. Seal, A. Raha, S. Maity, S. K. Mitra, A. Mukherjee, and M. K. Naskar, “A simple flood forecasting scheme using wireless sensor networks,” 2012.
- [39] Aosong (Guangzhou) Electronics Co., Ltd., *AM2302 Temperature and Humidity Module Product Manual*, Aosong (Guangzhou) Electronics Co., Ltd., 2012, digital temperature and humidity sensor module.
- [40] Texas Instruments, *LM393B, LM2903B, LM193, LM293, LM393 and LM2903 Dual Comparators Datasheet*, Texas Instruments, 2025, rev. AH.