Team 2

**IoT Automation for Atmospheric Water Harvesting**

Design Document

Aaryans Nepal Mausam Shrestha Alex Pham Sushant Dahal

CIT-381-001

NKU

Date: (05/06/2025)

# INITIA CONCEPT DOCUMENT

**Group Members**

● Aaryans Nepal

● Mausam Shrestha

● Alex Pham

● Sushant Dahal

Project Proposal: IoT Automation for Atmospheric Water Harvesting

Our project focuses on developing a smart IoT-based control and monitoring system for an Atmospheric Water Harvesting (AWH) unit. Rather than constructing the physical condensation or fog-based harvesting mechanism, our objective is to build the hardware and software necessary to manage, monitor, and optimize its operation through sensor-based automation.

# Internet Utilization and Data Access

We plan to use a Raspberry Pi (Model 4 or 5) as the primary controller with built-in Wi-Fi for internet access. When connectivity is unavailable (common in remote or rural setups), the device will log sensor data locally and push it to a cloud server once reconnected.

A BME280 sensor will be used to monitor ambient temperature and humidity, while an ultrasonic sensor (HC-SR04) will measure the water level in the tank. The system will be controlled via Python, and may include a manual button to trigger re-syncing of locally stored data to the cloud.

Our planned implementation includes:

● Real-time data monitoring via an LCD display

● Visual and audible alerts using LEDs and a buzzer

● Local logging of humidity and water levels

● MQTT-based data transmission to a cloud broker (hosted by a networking team)

● A simple button-triggered calibration routine for tank height

We will implement conditional logic where the system activates the water-harvesting fan or motor only when humidity is sufficient and the tank is not full, helping conserve power and prevent overflow.

# Challenges and Considerations

Some anticipated challenges include:

● Ensuring reliable operation and recovery in low-connectivity environments

● Implementing fallback logic for sensor or network failure

● Handling hardware integration like soldering, I2C communication, and GPIO configuration

● Designing a modular file structure for sensor reading, actuator control, data logging, and network management

● Building a system that can eventually support remote control or automation via a web dashboard

We will ensure the design is flexible enough to integrate with various harvesting methods and to scale across different AWH scenarios.

# Value of Network Connectivity

Internet connectivity enhances the system’s potential by enabling remote data access, log syncing, and possibly forecast-based decision making in future versions. However, the system will remain functional in completely offline modes—thanks to a local fallback mechanism that logs readings and manages control decisions using onboard logic.

# Conclusion

Through this project, we aim to develop a modular, resilient IoT control system for Atmospheric Water Harvesters that is capable of running in low-resource environments, storing data offline, and seamlessly syncing when connected. This will lay a solid foundation for smart resource management tools in rural or climate-sensitive regions.

**Table of Contents**

**1.INTRODUCTION**

1.1 Purpose

1.2 Scope

1.3 Overview of Document

1.4 Reference Materials

1.5 Definitions and Acronyms

**2. SYSTEM OVERVIEW**

2.1 Functional Summary

2.2 Key Features

2.3 Use Case: Treehouse Deployment

**3. SYSTEM ARCHITECTURE**

3.1 Architectural Design Overview

3.2 Subsystem Decomposition & Modular Structure

3.3 Design Rationale & Trade-offs

**4. SENSOR AND HARDWARE CONFIGUARTION**

4.1 Hardware Overview with Photographs

4.2 Sensor-to-GPIO Mapping

4.3 Breadboard Layout & Wiring Explanation

4.4 Schematic Diagram

**5. DATA DESIGN**

5.1 Data Structure & Storage Flow

5.2 Data Dictionary

5.3 Offline Logging (JSON) & Syncing Strategy

**6. COMPONENT DESIGN**

6.1 Sensor Control Logic

6.2 AWG Activation Algorithm

6.3 LED, Buzzer, LCD Feedback Handling

6.4 Button Logic: Calibration & Sync

6.5 MQTT Integration Logic

6.6 Key Python Functions and Pseudocode

**7. HUMAN INTERFACE DESIGN**

7.1 LCD Display Feedback States

7.2 LED Color Codes & Actions

7.3 Physical Button Mappings

7.4 LCD Fallback on Error

**8. REQUIREMENT MATRIX**

8.1 Feature-to-Requirement Trace Table

8.2 Functional Requirements Coverage

**9. CHALLENGES & SOLUTION**

9.1 Hardware Reliability (I2C / Echo)

9.2 Sensor Environment Issues (Low Humidity)

9.3 MQTT Failure Fallback Strategy

9.4 Power Management Notes

**10. Future Improvements**

10.1 Web-Based Dashboard  
10.2 AI-Based Decision Making  
10.3 Solar and Battery Integration  
10.4 Over-the-Air (OTA) Updates  
10.5 Enhanced Network Failover  
10.6 Multi-Tank Monitoring

**11. APPENDICES**

11.1 Full Parts List

11.2 Full Python Folder Structure

11.3 Extra Code Snippets (On-Demand Logic, Logger, Calibrator)

# Introduction

## Purpose

This document presents the comprehensive design and implementation strategy for the Smart Atmospheric Water Harvester (AWG) system developed by Team 2. It details both the hardware and software components, outlining their functionality and interactions for potential review by instructors, peers, or prospective employers.

## Scope

The project centers on an IoT-enabled AWG controller system that simulates atmospheric water harvesting. The system includes environmental sensing, actuator control, data logging, MQTT communication, and offline fallback, optimized for remote deployments like treehouses.

## Overview

This document follows IEEE 1016-style guidelines and includes system architecture, hardware wiring, data design, logic components, human interface design, challenges faced, and appendices with component specifications and code snippets.

## Reference Material

* IEEE Std 1016-1998
* Raspberry Pi GPIO documentation
* Adafruit Circuit Python libraries
* MQTT protocol and broker setup guidelines

## Definitions and Acronyms

* *AWG: Atmospheric Water Generator*
* *MQTT: Message Queuing Telemetry Transport*
* *GPIO: General Purpose Input/Output*
* *LCD: Liquid Crystal Display*
* *JSON: JavaScript Object Notation*

# SYSTEM OVERVIEW

## **2.1** Functional Summary

The system automates atmospheric water harvesting based on environmental conditions. It monitors humidity and tank water levels to activate a relay-driven motor. A real-time LCD display and visual/audio indicators communicate system status to users.

## 2.2 Key Features

* Humidity and water level monitoring
* Relay control logic based on thresholds
* Real-time status on 1602 LCD
* MQTT publishing with offline fallback
* Calibration and sync buttons

## 2.3 Use Case: Treehouse Deployment

Optimized for low-connectivity, power-constrained setups in treehouses or rural areas. Offline logging ensures continuity, and sync functionality uploads data when the internet is restored.

# System Architecture

## Architectural Design

The system uses a Raspberry Pi 4 as the central controller. It reads environmental data, controls output devices, and logs/publishes operational data.

## 3.2 Subsystem Decomposition & Modular Structure

* sensors/: BME280, HC-SR04 handlers
* actuators/: relay, LCD, LEDs, buzzer
* utils/: constants, tank calibration, logger
* network/: MQTT publishing, connectivity checker

## 3.3 Design Rationale & Trade-offs

Python modules ensure modularity and clarity. MQTT was selected for lightweight communication. GPIO pins were mapped with care to avoid pin conflicts and support simultaneous input/output handling.

# Sensor and hardware configuration

## 4.1 Hardware Overview with PhotographsBefore Presentation Setup

**A cup with wires and wires on a table

Description automatically generatedA computer with a cup and wires

Description automatically generated**

Final Setup At Home

**A cup with wires and a circuit board

Description automatically generated**

LCD Screen Live Display From Sensor

**A blue screen with wires and wires

Description automatically generated**

LEDS, BUZZER, BUTTONS, WIRING, BREADBOARD, RELAY MODULE

**A circuit board with wires

Description automatically generated**

BME280 SENSOR – Did Soldering At Home

**A hand holding a small device

Description automatically generated**

Relay Module

**A circuit board with wires and a battery

Description automatically generated**

Motor

**A close up of a device

Description automatically generated**

HC-SR04 – For Water Level

**A cup with a circuit board and wires

Description automatically generated**

Raspberry Pi 4

A black rectangular device with a grey strap

Description automatically generated  
  
Live Terminal Output  
The system successfully boots using a previously saved tank height (0.118 m), automatically computing the trigger distance for tank full condition (0.018 m). It reads humidity (63.5%) and water level continuously (~0.12 m) while logging the data locally due to an MQTT configuration error (BROKER not defined).

When the water level temporarily drops below the 5 cm safety threshold, the AWG shuts off to prevent overflow or sensor error. The sync button (GPIO 16) is pressed, attempting to send unsynced logs—however, the MQTT error prevents this.

Later, the calibration button (GPIO 5) is pressed, and the system successfully recalibrates the tank height to 0.117 m, updating the trigger distance accordingly. Continuous monitoring resumes with updated configuration and stable readings

Note: Internet connectivity and MQTT integration depend on configuration parameters (e.g., broker address and port) to be provided by the On-Demand Networking Team. Once these values are supplied, real-time publishing will be enabled, seamlessly replacing the current offline logging fallback.  
A computer screen shot of a program

Description automatically generated

## 4.2 Sensor-to-GPIO Mapping

|  |  |  |
| --- | --- | --- |
| **Component** | **GPIO Pin** | **Notes** |
| HC-SR04 Trigger | GPIO 23 | Output pin |
| HC-SR04 Echo | GPIO 24 | Input pin |
| BME280 | I2C (SCL: GPIO 3, SDA: GPIO 2) | Address: 0x76 |
| Relay | GPIO 18 | Controls motor |
| LCD (I2C) | Address: 0x27 | 1602 LCD |
| Sync Button | GPIO 16 | Pull-up config |
| Calibrate Button | GPIO 5 | Pull-up config |
| LEDs (G/Y/B) | GPIO 27/22/17 | State indicators |

## 4.3 Breadboard Layout & Wiring Explanation

* Use color-coded jumper wires
* Relay must be powered from 5V
* Ensure LCD shares GND with Pi
* Buttons wired with pull-up logic

## 4.4 Schematic Diagram

**A diagram of a circuit board

Description automatically generated**

# Data Design

## 5.1 DATA STRUCTURE AND STORAGE FLOW

Sensor readings and AWG events are logged as JSON. Logs are stored locally and synced to the MQTT broker.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| humidity | float | Relative humidity (%) |
| temperature | float | Air temperature (C) |
| distance | float | Tank distance (m) |
| session | int | Session number |
| reading | int | Reading index |

## 5.2 Data Dictionary

## 5.3 Offline Logging (JSON) & Syncing Strategy

If MQTT fails, logs are saved to data\_log.json. A manual sync button sends the data once internet is restored. The system also checks for internet every 2 hours.

# **COMPONENT DESIGN**

## 6.1 Sensor Control Logic

Humidity Monitoring Strategy:

We check humidity using the BME280 sensor every 2 hours, based on findings from a performance evaluation study of commercial AWG units. In that study, three commercially available AWGs were operated for 24 hours under controlled conditions, and it was observed that maximum water yield occurred when humidity exceeded 75%.

After analyzing average humidity trends in the Cincinnati area, which typically hover around 60%, we selected 60% as the operational threshold for activating the AWG. Although regional humidity tends to change approximately every 4 hours, we chose a 2-hour interval to achieve more granular monitoring without overreacting to short-term fluctuations.

Checking humidity at longer intervals also resolves a critical issue: if humidity were measured every few seconds, slight sensor fluctuations between 59% and 60% could cause the AWG to frequently start and stop, reducing efficiency. Our approach ensures that once the system activates, it runs for a minimum of 2 hours, maximizing water production. This strategy also enables future enhancements, such as estimating daily water yield using 24-hour weather forecasts obtained via the internet.

*if current\_time - last\_bme\_check >= BME\_CHECK\_INTERVAL or humidity is None:*

*try:*

*humidity = get\_humidity()*

*bme\_ok = True*

*last\_bme\_check = current\_time*

*reading\_counter += 1*

*print(f"Humidity: {humidity:.1f}%")*

*# --- 2. Record this reading (MQTT or local) ---*

*distance\_for\_log = get\_water\_distance()*

*if is\_connected():*

*success = publish\_sensor\_data(session\_id, reading\_counter, humidity, distance\_for\_log)*

*if not success:*

*save\_locally(session\_id, reading\_counter, humidity, distance\_for\_log)*

*stop\_no\_internet()*

*else:*

*save\_locally(session\_id, reading\_counter, humidity, distance\_for\_log)*

*indicate\_no\_internet()*

*# --- 3. Check distance every 10 seconds ---*

*try:*

*distance = get\_water\_distance()*

*if distance is not None:*

*print(f"Water Level: {distance:.2f} m")*

*distance\_ok = True*

*else:*

*print("[Distance] No reading")*

*distance\_ok = False*

*except Exception as e:*

*print(f"[Distance ERROR] {e}")*

*distance = None*

*distance\_ok = False*

## 6.2 AWG Activation Algorithm

* Idea of when it should run:

*if humidity >= 60 and distance > 0.05:*

*turn\_on\_awg()*

*else:*

*turn\_off\_awg()*

*# ---AWG logic ---*

*if humidity\_ok and not tank\_full:*

*if not AWG\_RUNNING:*

*session\_id += 1 # New session when AWG starts*

*indicate\_good\_humidity()*

*stop\_alarm()*

*turn\_on\_awg()*

*AWG\_RUNNING = True*

*elif tank\_full:*

*print("Stopping: Tank full.")*

*indicate\_low\_humidity()*

*sound\_alarm()*

*turn\_off\_awg()*

*AWG\_RUNNING = False*

*else:*

*print("Stopping: Humidity low or sensor error.")*

*indicate\_low\_humidity()*

*stop\_alarm()*

*turn\_off\_awg()*

*AWG\_RUNNING = False*

## 6.3 LED, Buzzer, LCD Feedback Handling

* Yellow LED: No internet
* Blue LED: System not running
* Green LED: System running
* Buzzer triggers on overflow
  + - LCD cycles through: humidity, temp, water level
* LED Logic:  
  *LED\_BLUE = 17 # System not running*

*LED\_GREEN = 27 # System running*

*LED\_YELLOW = 22 # No internet*

*BUZZER = 26 # Tank full*

*blue\_led = LED(LED\_BLUE)*

*green\_led = LED(LED\_GREEN)*

*yellow\_led = LED(LED\_YELLOW)*

*buzzer = Buzzer(BUZZER)*

*def indicate\_low\_humidity():*

*blue\_led.on()*

*green\_led.off()*

*def indicate\_good\_humidity():*

*green\_led.on()*

*blue\_led.off()*

*# No internet indicator*

*def indicate\_no\_internet():*

*yellow\_led.on()*

*def stop\_no\_internet():*

*yellow\_led.off()*

*def clear\_leds():*

*blue\_led.off()*

*green\_led.off()*

*yellow\_led.off()*

*def sound\_alarm():*

*buzzer.on()*

*def stop\_alarm():*

*buzzer.off()*

## 6.4 Button Logic: Calibration & Sync

* GPIO 5: On boot, calibrates tank height  
    
  After calibration calculates threshold to shutdown. The default threshold value for shutdown is 15% of the height.  
  *FULL\_TRIGGER\_DISTANCE = round(height \* 0.15, 3)*  
  And then later:  
  *tank\_full = is\_tank\_full(distance, FULL\_TRIGGER\_DISTANCE)*

Load Tank Height or Fallback Logic:

*height = load\_tank\_height()*

*if height:*

*FULL\_TRIGGER\_DISTANCE = round(height \* 0.15, 3)*

*print(f"[LOAD] Using saved tank height: {height:.3f} m → Trigger at {FULL\_TRIGGER\_DISTANCE:.3f} m")*

*else:*

*FULL\_TRIGGER\_DISTANCE = DEFAULT\_FULL\_TRIGGER\_DISTANCE*

*print("[FALLBACK] No saved height. Using default tank height.")*

Complete Logic:  
*if GPIO.input(CALIBRATION\_BUTTON\_PIN) == GPIO.LOW:*

*print("[CALIBRATION] Button pressed. Recalibrating tank height...")*

*new\_height = calibrate\_tank\_height()*

*if new\_height:*

*save\_tank\_height(new\_height)*

*FULL\_TRIGGER\_DISTANCE = round(new\_height \* 0.15, 3)*

*print(f"[CALIBRATION] New height: {new\_height:.3f} m → Trigger at {FULL\_TRIGGER\_DISTANCE:.3f} m")*

*else:*

*print("[CALIBRATION ERROR] Calibration failed. Retaining previous config.")*

*time.sleep(0.5) # Debounce*

* GPIO 16: On press, pushes data\_log.json to MQTT

*if GPIO.input(SYNC\_BUTTON\_PIN) == GPIO.LOW:*

*print("[SYNC] Sync button pressed!")*

*time.sleep(0.5) # debounce*

*if is\_connected():*

*stop\_no\_internet()*

*try:*

*if LOG\_PATH.exists():*

*with open(LOG\_PATH, "r") as f:*

*data = json.load(f)*

*if not data:*

*print("[SYNC] No offline data found.")*

*else:*

*for entry in data:*

*success = publish\_sensor\_data(*

*entry["session"],*

*entry["reading"],*

*entry["humidity"],*

*entry["water\_level"]*

*)*

*if not success:*

*print("[SYNC] Failed to send one entry. Stopping sync.")*

*break*

*else:*

*with open(LOG\_PATH, "w") as f:*

*json.dump([], f)*

*print("[SYNC] All offline data pushed and cleared.")*

*else:*

*print("[SYNC] data\_log.json not found.")*

*except Exception as e:*

*print(f"[SYNC ERROR] {e}")*

*else:*

*indicate\_no\_internet()*

*print("[SYNC] No internet. Cannot sync now.")*

## 6.5 MQTT Integration LogicAWG Activation Algorithm

* Idea of when it should do:

*if is\_connected():*

*publish\_sensor\_data(...)*

*else:*

*save\_locally(...)*

* MQTT LOGIC + DATA RECORDING

*def publish\_sensor\_data(session, reading, humidity, water\_level):*

*client = mqtt.Client()*

*try:*

*client.connect(BROKER, PORT, 60)*

*payload = {*

*"session": session,*

*"reading": reading,*

*"humidity": humidity,*

*"water\_level": water\_level*

*}*

*client.publish(TOPIC, json.dumps(payload))*

*client.disconnect()*

*print("[MQTT] Data published.")*

*return True*

*except Exception as e:*

*print(f"[MQTT ERROR] {e}")*

*return False # ← This ensures fallback happens*

## 6.6 Check Internet Connection

* Logic for internet connectivity:  
  *def is\_connected(host="8.8.8.8", port=53, timeout=3):*

*try:*

*socket.setdefaulttimeout(timeout)*

*socket.socket(socket.AF\_INET, socket.SOCK\_STREAM).connect((host, port))*

*return True*

*except socket.error:*

*return False*

## 6.7 Key Python Functions and Pseudocode

* display\_status()
* is\_tank\_full()
* save\_locally()
* publish\_sensor\_data()

# Human Interface Design

## 7.1 LCD Display Feedback States

* Display 2-line: Humidity, Temp/Status
* On error: Sensor fails, Tank full

## 7.2 LED Color Codes & Actions

* Green: AWG ON
* Red: Overflow alert
* Blue: Humidity too low

## 7.3 Physical Button Mappings

* Calibrate (GPIO 5): Tank height
* Sync (GPIO 16): Force data upload

## 7.4 LCD Fallback on Error

* If LCD fails, logs print to terminal with [LCD ERROR] noted.

# Requirements Matrix

## 8.1 Feature-to-Requirement Trace Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Feature** | **Sensor** | **Logic** | **LCD** | **MQTT** | **Logging** |
| Auto ON/OFF | ✅ | ✅ | ✅ | - | ✅ |
| Overflow alert | ✅ | ✅ | ✅ | - | ✅ |
| Offline backup | - | ✅ | - | - | ✅ |
| Sync button/ Calibration button | - | ✅ | ✅ | ✅ | ✅ |

## 8.2 Functional Requirements Coverage

* All primary functionalities are met through sensor-based control, manual override buttons, and both online and offline data handling.

# Challenges and Solutions

## 9.1 Hardware Reliability (I2C / Echo)

* Occasional I2C timeout or echo stuck error required retry logic and error catch.

## 9.2 Sensor Environment Issues (Low Humidity)

* Added humidity threshold override for dry demo days.

## 9.3 MQTT Failure Fallback Strategy

* JSON logging implemented for guaranteed data persistence.

## 9.4 Power Management Note

* System can be powered by a USB battery bank; low draw makes it feasible.

# Future Improvements

As a prototype, the Smart Atmospheric Water Harvester (AWG) demonstrates core functionality in a modular and extensible fashion. Future enhancements could greatly improve system intelligence, energy efficiency, and user interaction:

10.1 Web-Based DashboardIntegrate a web interface for remote monitoring and control. This would visualize humidity trends, water levels, and session logs in real-time, offering a mobile-friendly control panel.

## 10.2 AI-Based Decision Making

Use machine learning to dynamically adjust AWG operation based on weather forecasts, historical data, and power availability. This would optimize water yield while minimizing energy consumption.

* 1. Solar and Battery IntegrationEnable fully off-grid deployment by powering the system through solar panels and battery banks. Intelligent power management could prioritize critical operations under low-energy conditions.
  2. Over-the-Air (OTA) UpdatesAllow software updates to be pushed remotely via the internet or MQTT, reducing the need for physical access to the Pi and increasing maintainability in rural deployments.
  3. Enhanced Network FailoverAdd support for cellular modules (4G/LTE) or LoRaWAN to improve internet redundancy in remote locations, ensuring reliable MQTT publishing.
  4. Multi-Tank MonitoringExpand the logic to manage and monitor multiple tanks simultaneously, each with individual sensors and control logic, useful in scaled implementations.

# Appendices

## 11.1 Full Parts List

|  |  |  |
| --- | --- | --- |
| **Component** | **Model** | **Qty** |
| BME280 | I2C Sensor | 1 |
| HC-SR04 | Ultrasonic | 1 |
| Relay | 5V Module | 1 |
| LCD | 1602 I2C | 1 |
| LED (Y/G/B) | 5mm | 3 |
| Buzzer | Passive | 1 |
| Buttons | GPIO Input | 2 |

## 11.2 Full Python Folder Structure

This repository implements a modular Python-based control system for a Smart Atmospheric Water Generator (AWG). The code is organized into functional subdirectories for clarity and scalability. Below is a breakdown of each folder and its responsibility:

AWG/

├README

├── main.py

├── sensors/

│ ├── bme280\_reader.py

│ └── distance\_sensor.py

├── actuators/

│ ├── relay\_control.py

│ ├── leds.py

│ └── lcd\_display.py

├── network/

│ ├── mqtt\_handler.py

│ └── connectivity.py

├── utils/

│ ├── constants.py

│ ├── logger.py

│ └── tank\_config.py

└── data/

└── data\_log.json

**main.py**

The entry point of the application. It contains the main control loop that:

* Reads humidity and water level
* Controls the AWG motor based on conditions
* Handles safety shutdowns
* Displays live data on LCD
* Detects button presses for calibration and syncing
* Performs offline logging and MQTT publishing

**sensors/**

Responsible for environmental data acquisition.

* bme280\_reader.py: Interfaces with the BME280 sensor to read temperature and humidity.
* distance\_sensor.py: Uses the HC-SR04 ultrasonic sensor to measure water level and handles tank calibration logic.

**actuators/**

Controls output components based on system logic.

* relay\_control.py: Turns the AWG motor (or fan) ON/OFF using a GPIO relay.
* leds.py: Manages RGB LED indicators and buzzer alarms to reflect system status.
* lcd\_display.py: Displays real-time humidity, temperature, water level, and error states on a 1602 I2C LCD.

**network/**

Handles internet connectivity and MQTT messaging.

* connectivity.py: Checks if the system is online.
* mqtt\_handler.py: Publishes sensor readings to a specified MQTT broker when the network is available.

**utils/**

Shared constants and support logic.

* constants.py: Stores system-wide constants like GPIO pins, thresholds, and default tank height.
* logger.py: Saves data locally to a JSON file (data\_log.json) for offline mode.
* tank\_config.py: Saves and loads the calibrated tank height to/from a file for persistent configuration.

**data/**

* data\_log.json: Stores sensor logs temporarily when internet is unavailable. Synced to MQTT later using the sync button.

This structure ensures the system is clean, maintainable, and hardware-adaptive. Each module is focused on a single responsibility, which supports modular testing and easier debugging.

## 11.3 Extra Code Snippets

* Include extended calibration logic, LCD scrolling messages, and error recovery routines if needed.