

Smart Water Management System for Buildings

by

Peter Clarke, Megan Field, Kiana Lee, Andrew Sudyk

Computer Engineering Capstone Design Project
Toronto Metropolitan University, Winter 2026

Acknowledgements

We would like to express our gratitude to our Faculty Lab Coordinator, Amr Mohamed for providing us with guidance, feedback and technical expertise through the design process of this engineering design project.

Certification of Authorship

We hereby certify that this paper is the result of contribution from all group members and entirely our own work. All sources paraphrased or used in the preparation of the report have been properly cited.

Kiana Lee

Peter Clarke

Megan Field

Andrew Sudyk

Table of Contents

Acknowledgements.....	2
Certification of Authorship.....	3
Table of Contents.....	4
Abstract.....	6
1. Introduction & Background.....	7
1.1 Problem statement.....	7
1.2 Related Work.....	8
2. Objectives.....	10
3. Theory and Design.....	11
3.1 Hardware Demonstration and Hydraulic Network.....	11
Electronics and Sensor Integration.....	15
ESP32 Program Structure.....	19
3.2 IoT Communication Protocol.....	20
Prototyping.....	21
MQTT Broker.....	21
IoT Gateway API.....	22
3.3 Software Architecture: Backend and Dashboard.....	22
Backend Services.....	22
Web Dashboard.....	22
4. Alternative Designs.....	26
4.1 Hardware-Focused vs. Software-Focused Design.....	26
4.2 Manual Threshold Calibration vs. Machine Learning.....	26
4.3 Sensor Selection: Pressure/Flow vs. Ultrasonic and Quality Sensors.....	26
4.4 Leak Detection Methodology: Flow Differential vs. Pressure Sensing.....	27
4.5 Data Analysis: Long-Term Usage Comparison.....	27
4.6 ESP32 Locking: delay() vs. millis().....	27
4.7 Native Application versus Website.....	28
4.8 React Router versus any other framework.....	28
4.9 Node.js versus C# for the backend.....	29
4.10 PCB Design Improvements.....	29
5. Material/Component list.....	30
6. Measurement and Testing Procedures.....	32
6.1 Sensor Data Transmission and Dashboard Integration.....	32
6.2 IoT Communication Integrity.....	32
6.3 Anomaly Detection and Threshold Calibration.....	32
6.4 Deployment Considerations.....	33
7. Performance Measurement Results.....	34
7.1 Sensor Data Transmission and Dashboard Integration.....	34
7.2 IoT Communication Integrity.....	36

7.3 Anomaly Detection and Threshold Calibration.....	38
8. Analysis of Performance.....	40
8.1 Sensor Data Transmission and Dashboard Integration.....	40
8.2 IoT Communication.....	40
8.3 Anomaly Detection and Threshold Calibration.....	40
9. Conclusions.....	42
9.1 Summary.....	42
9.2 Discrepancies.....	42
9.3 Major Difficulties Encountered.....	42
9.4 Future Work.....	43
References.....	44
Appendices.....	46
Server Source Code:.....	46
ESP32 Source Code.....	54

Abstract

Water distribution leaks in residential and commercial buildings lead to significant environmental waste, financial losses, and structural damage. While existing smart-home systems employ machine learning to monitor water networks, they frequently produce high false-positive rates due to unpredictable human behaviour, penalizing legitimate high-volume consumption. To address this, this project aims to develop an integrated IoT-based smart water management system providing real-time monitoring, early anomaly detection, and automated shutoffs. The design prioritizes deterministic user control to minimize false alarms and ensures reliable operation with real-time connectivity. A physical hydraulic demonstration unit was constructed using a centrifugal pump, pressure transducer, and a flow meter, all controlled on an ESP32 microcontroller. The IoT communication layer utilizes the MQTT protocol to transmit real-time sensor data to a Node.js backend, while a React-based web dashboard allows users to visualise pressure and flow trends, view event logs, and utilise a manual override to temporarily suspend anomaly detection during anticipated high-volume water events. The ESP32 processes anomaly detection locally using empirically calibrated static pressure thresholds. Physical testing validated the system's sensitivity and responsiveness. By evaluating flow dynamics the system successfully detected simulated leaks resulting in as little as 0.264 L/min of water loss, triggering an automated shutoff upon a distinct pressure drop. Similarly, blockage events successfully triggered an automated rerouting mechanism to prevent system damage. Furthermore, offline testing confirmed that the ESP32 executes its protective shutoff protocols completely independently of the centralized dashboard, eliminating cloud-latency safety risks. The Smart Water Management System provides a highly reliable, cost-effective solution for mitigating water damage and resource waste. By combining localized edge computed pressure logic with an accessible manual-override dashboard, the proposed system resolves the usability limitations of predictive market alternatives and delivers a resilient smart-home architecture.

Keywords: Internet of Things; Leak Detection; Pressure Sensing; Water Management; Automation

1. Introduction & Background

1.1 Problem statement

Water distribution in residential and commercial buildings is an essential but often overlooked component of everyday infrastructure. Leaks and abnormal consumption patterns can lead to significant environmental waste, financial losses, and structural damage, and often goes undetected until damages and costs become significant. As municipalities modernize their water networks and consumers increasingly adopt smart-home technologies, there is a growing need for accessible, real-time monitoring tools that help occupants understand and manage their water usage.

This project addresses this need by developing an integrated IoT-based water monitoring and leak-detection system that combines hardware pressure sensing, wireless communication, and a centralized dashboard. By enabling early leak and blockage detection, automated alerts and customizable behaviour, the system aims to support more efficient water usage, reduce household costs and contribute to broader sustainability efforts.

Water leaks in residential and commercial buildings represent a significant environmental, financial and safety concern, contributing to unnecessary freshwater waste, increased energy consumption, avoidable strain on municipal water systems, compromised building integrity, and significant utility cost for occupants. Non-revenue water (NRW) refers to water that is produced but never reaches the consumer due to leaks, theft or metering inaccuracies [1]. On average, about 20 to 30% of water in distribution systems is lost before reaching consumers [1]. For a medium-sized city with $450,000m^3$ per day of produced water, this water loss can result in an over US\$13,000,000 annual cost in non-recoverable chemical and energy costs and non-recoverable labour [2]. These environmental and financial expenses related to water loss do not exclusively stem from the loss of freshwater, as when water is lost due to leaks, the energy used to treat and transport that water is similarly wasted. Every cubic meter of water consumed generates 10.6 kg of carbon emissions [3]. According to the city of Toronto, a leaky toilet can create over $600m^3$ of water waste every month, or $20m^3$ a day [4]. The annual waste of a leaky toilet averages to $4400m^3$ a year, which emits 4.5 tons of greenhouse gasses. This is a similar quantity to the average emissions of a passenger car [3].

Size of Hole	Amount and Cost of Water Wasted
• 1/16" hole	Wastes $3.57m^3$ (3,570 litres) in 24 hours Costs \$17.36/day or \$2,083.20 in 120 days
• 1/8" hole	Wastes $14m^3$ (14,000 litres) in 24 hours Costs \$68.08/day or \$8,169.60 in 120 days
• 3/16" hole	Wastes $32.13m^3$ (32,130 litres) in 24 hours Costs \$156.24 /days or \$18,748.80 in 120 days

Figure 1.1.1: Cost of a continuous leak from a hole in a water service pipe. [4]

There are also substantial personal financial costs associated with water loss. When considering the previous leaky toilet example, the corresponding personal expense for this waste is a water bill increase of CA\$2,812.32 per month, or CA\$93.74 per day [4]. Costs resulting from leaks can also be accrued through water damage. On average, repairing water damage in Toronto ranges from \$2,000 to \$10,000. In more severe cases, this cost can range from \$5,000 to \$25,000 and a full house damage repair can cost somewhere between \$30,000 to \$60,000 [5]. Persistent leakage damages soil and building foundations by increasing moisture exposure, which can lead to structural damage, mold, or compromised building integrity [1]. Additionally, repeated failures in water distribution networks can accelerate wear and tear, requiring even more frequent repair or replacement of pipes, further increasing energy costs associated with infrastructure maintenance and reconstruction [6]. Additionally, water leakage prevention is found to be one of the most cost-effective methods of water conservation. For a median utility, reducing leaks can save more water per dollar invested than other conservation efforts like rebate programs or conservation campaigns [7]. Thus, failure to detect and fix leaks represents a lost opportunity, where individuals experience high water bills, wasted water resources, and avoidable energy and infrastructure costs.

Due to the environmental and financial impacts of water leakage being so significant, there is a growing interest in smart, sensor-based water management and leak detection solutions. Traditional techniques such as acoustic or visual monitoring, manual inspections or statistical predictive software are found to be time-consuming and have difficulty pinpointing leaks, especially small or slow ones, and cannot anticipate water-related issues [1]. Smart water management systems which combine flow meters, pressure sensors, data analytics and real time monitoring enable early detection of leaks, continuous monitoring, and automated alerts or shutoffs. This is significantly more effective at preventing water losses before they become large, minimizing wasted energy, reducing carbon footprint, avoiding structural damage and saving money for occupants and utilities alike.

1.2 Related Work

The advancement of Internet of Things (IoT) technology has led to the development of various smart water management systems designed to detect leaks and prevent catastrophic structural damage. These systems generally fall into two primary categories, localized point-sensing and whole-home flow monitoring. Localized sensors, such as the Eve Water Guard, utilize extended sensing cables placed in high-risk areas (e.g., basements, under sinks, or near washing machines) to detect the physical presence of water [8]. While effective for providing a layer of protection in specific zones, their primary drawback is spatial limitation, where they are unable to detect leaks within walls or in unmonitored areas. In contrast, whole-home flow monitors, such as the Flo Smart Water Monitor and Shutoff by Moen, are installed directly on a building's main water supply line. These centralized systems continuously measure water flow rates, pressure, and temperature. They utilize machine learning algorithms to establish a baseline of normal household water usage and can automatically trigger a shut-off valve if an anomaly is detected [9].

Despite the widespread commercial adoption and environmental benefits of automated whole-home systems which are often highlighted through corporate sustainability initiatives and

service integrations [10], [11], their reliance on rigid algorithmic models introduces significant usability challenges. Systems like the Flo by Moen employ predictive pattern recognition to distinguish between normal consumption and potential leaks. However, post-market consumer feedback reveals a critical limitation, these systems frequently fail to accommodate dynamic or unpredictable human behavior [12]. Legitimate, high-volume water usage events that occur out of an established sequence, such as prolonged lawn watering, accommodating guests, or off-schedule showering, are frequently misclassified as critical anomalies. Consequently, these machine learning models generate high rates of false positives, incorrectly identifying routine tasks as severe leaks [12].

To address the limitations of purely predictive algorithms, there is a clear necessity for an interface that provides users with greater control over their households without losing on water damage security. While commercial systems penalize irregular schedules, integrating a centralized dashboard with a manual override, or "pause" function, allows users to temporarily suspend anomaly detection prior to initiating a known high-volume water event. This isolated architectural improvement ensures continuous water supply during unpredicted usage without sacrificing the baseline integrity of the leak detection system.

2. Objectives

1. Develop an IoT-based system using flow and pressure sensors to collect water usage data
2. Create a dashboard to view sensor readings and system status
 - a. Normal State
 - b. Leak Detected
 - c. Blockage Detected
3. Ensure system performs with real-time connectivity
4. Interpret flow and pressure sensor data to calculate thresholds for “Leak” state
5. Interpret flow and pressure sensor data to calculate thresholds for “Blockage” state
6. Detect potential leaks and implement an automated shut off feature based on the calculated pressure and flow threshold
7. Detect potential blockages and implement an automated rerouting feature based on the calculated pressure and flow threshold
8. Provide a password protected manual override function on dashboard to:
 - a. Turn pump on and off
 - b. Reroute flow to different valve

3. Theory and Design

This section details the theoretical framework and engineering design specifications of the Smart Water Management System. The system architecture is decided into three primary subsystems, the physical hardware and hydraulic network, the IoT communication layer, and the software backend with the user interface. The design philosophy prioritizes modularity, scalability through microservices, and robust data handling to ensure accurate leak detection and system automation.

The overall layout of the demo board was carefully thought through. The delta from the lowest point of the water system, and the highest point of the water system, is 72 cm. The initial 72 cm climb ensures the pressure sensor placed at height 20 cm, measures a guaranteed potential pressure value due to gravity. We can calculate this potential using a formula from fluid dynamics:

$$P_{gauge} = pgh$$

Where p is the fluid density ($1000 \frac{kg}{m^3}$)

g is the gravity ($9.81 \frac{m}{s^2}$)

h is the height of the pipe (52 cm)

By definition, gauge measures pressure compared to a reference atmospheric pressure. A value of 5.1 kPa is calculated, which converts to 0.74 PSI. Therefore, we know that the weight of the water acting on the sensor alone will always be 0.74 PSI. Any pressure thereafter is caused by kinetic energy of the water flowing, which can only be observed after testing. Section 4 of this report mentions alternative designs, and a potential flaw in this formula.

3.1 Hardware Demonstration and Hydraulic Network

The physical design of the system models a residential plumbing network to validate leak detection algorithms and automated control mechanisms. The hydraulic circuit is constructed using transparent PVC tubing with a 1/2-inch inner diameter, selected for its compatibility with standard hose adapters and the ability to visually inspect the fluid flow. The system is powered by a centrifugal 12V pump submerged within a 5-liter acrylic reservoir. To ensure structural integrity and vibration isolation, a custom 3-D printed raft was designed to hold the system's water reserve and centrifugal water pump.

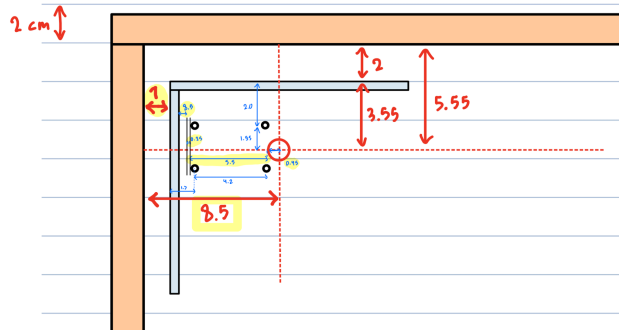


Figure 3.1.1: Diagram of Millimeter Measurements for Pump Placement.

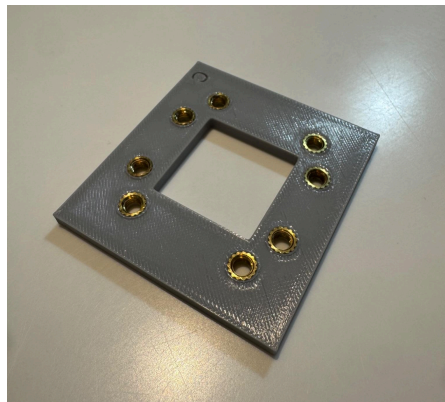


Figure 3.1.2: Pump Raft.

There were careful design considerations regarding how to securely attach a 220 g vibrating pump to the bottom of an acrylic tank, while also aligning the output of the pump exactly 5.5 cm away from the backboard. The inspiration for mounting came from the same welding technique used to assemble the tank itself, where a donut shaped plastic bracket would surround a centre acrylic square, and allow the pump to screw into melted brass inserts. The middle square would be exactly 1 mm taller than the ring around it, giving space for the acrylic sheets to physically weld together, while trapping the donut-shaped ring. To ensure a mechanical strength, multiple cheese-styled holes were cut into the acrylic sheets, allowing the dichloromethane solvent to pool into. In addition, the pump baseplate offered eight mounting holes, even though using eight screws would be unnecessary.

Each sheet was sanded with 400-600 grit sandpaper, allowing the solvent to disperse in between sheets, since a property of dichloromethane is low surface tension, and the solvent spreads through capillary action.

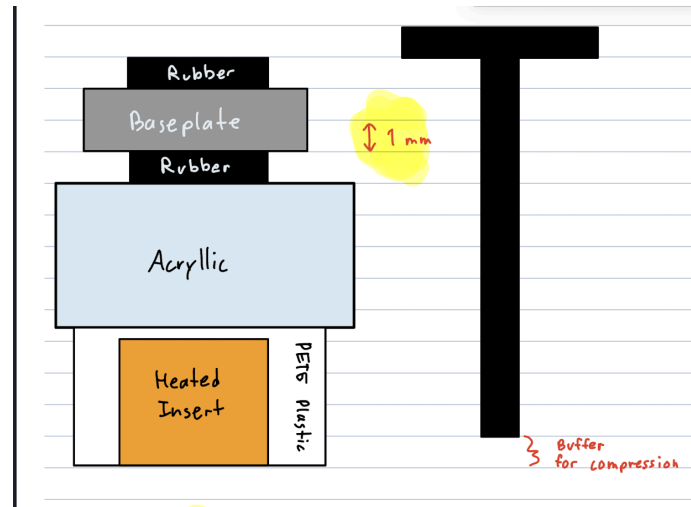


Figure 3.1.3: Drawn layered view [13]

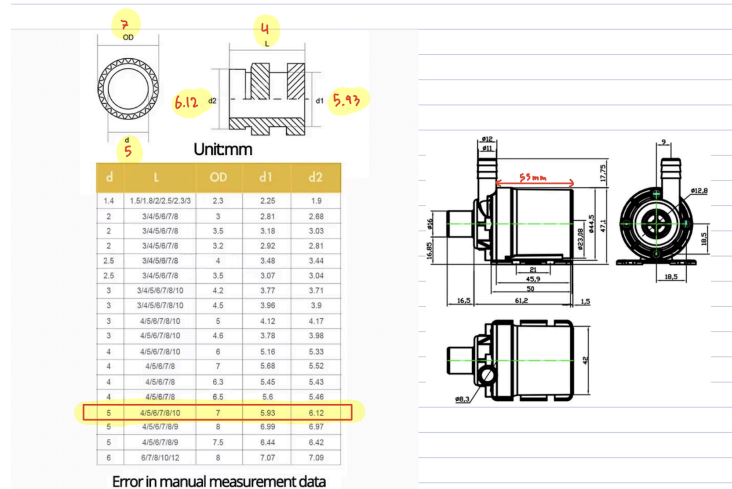


Figure 3.1.4: Heated insert datasheet (left), pump measurement schematics (right)

The problem was approached from multiple angles. First, a layered sketch was made, to measure the thickness of each material between the bottom of the tank, and the screw head. The appropriate length M3 screws were purchased, based on the cumulative height of all the layers sandwiched together (plus a buffer). For example, two rubber washers, that are 1 mm in height, were placed between the pump baseplate and the screwhead, since the source of any vibrations would start here. Even though the total height was 11 mm, a slightly smaller screw length of 10 mm was chosen, so that the bottom of the screw doesn't hit the bottom of the water tank, causing cracking. We also accounted for the compression of the rubber washers.

The structural framework consists of a plywood backboard segmented into three pieces to form a 3-foot vertical demonstration unit as featured in Figure 3.1.6. Components are mounted to this backboard using custom 3D-printed brackets. These brackets have undergone rigidity testing, with final prototypes secured with threaded inserts and stainless steel M5 screws to ensure even pressure distribution against the plastic components.

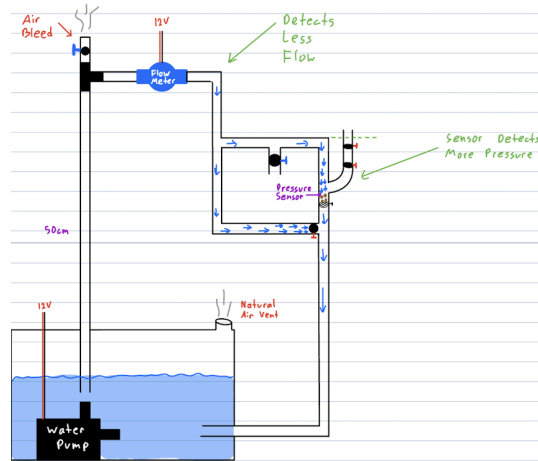


Figure 3.1.5: Blueprint of the System Framework [13].

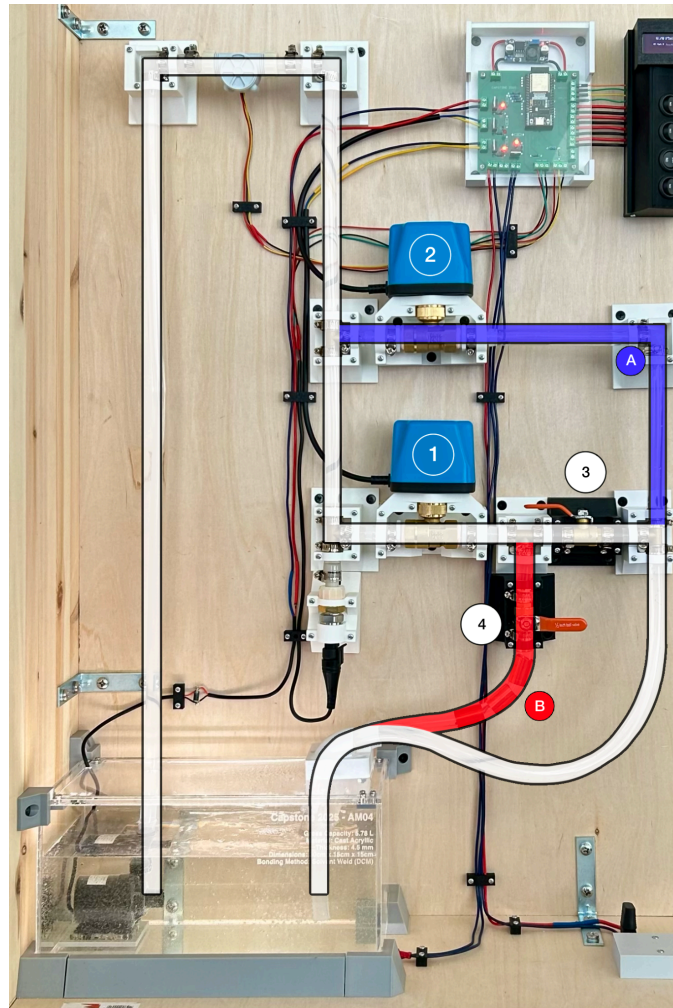


Figure 3.1.6: Annotated Image of Hardware.

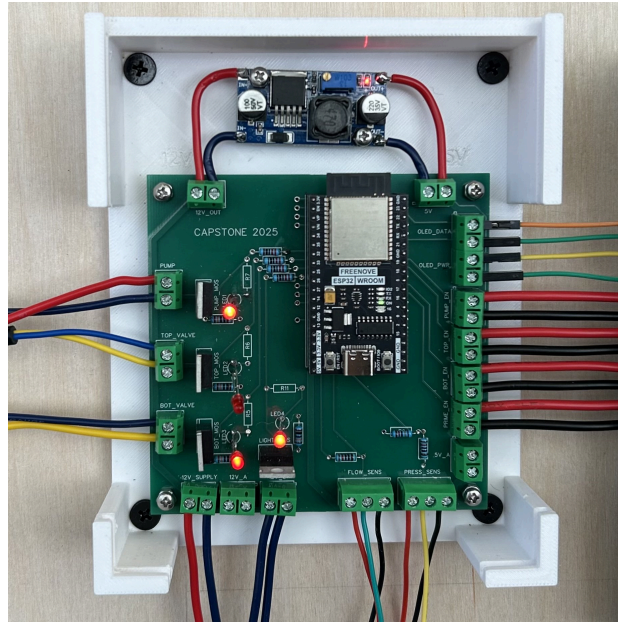


Figure 3.1.9: Physical PCB Wired to System.

PCB design choices are listed below:

- Dedicated 12V traces for the pump and valves, and 5V traces for the microcontroller logic. Using larger trace widths were used at no extra cost from the manufacturer, to safely add a buffer to the maximum expected amperage.
- An off-board LM2596 buck converter was used to step down 12V to 5V.
- IRLZ44N's were used as the MOSFET of choice, to provide low-side logic switching. PWM (pulse width modulation) was possible, but only used to fade the tank strip light as an indicator
- 3mm red LED diodes were added to the MOSFET gate traces, to signal the state of the microcontroller pins.
- 2P and 3P screw terminals were chosen for simplicity
 - 2 extra terminals were left unused, for possible future expansion or testing
- 4 user-programmable rocker switches were added to the input pins of the microcontroller. Currently, they are set to exit error states, and override MQTT connection
- Female header pins were soldered onto the board, allowing the microcontroller to be swapped easily.
- M3, 5mm tall brass standoffs were used between the PCB housing and the board
- 1-inch OLED screen was mounted near-eye level, providing live pressure and flow readings

In practical application, the system would only require electrical energy to operate the PCB, as the demonstration's water pump would be replaced by the naturally pressurized flow of a building's water main. Because the power consumed by the microcontroller board is so minimal, the substantial environmental and financial benefits of water conservation greatly trump the minor energy footprint required to drive the hardware.

A 1/4-inch pressure transducers are positioned at the low point of the system to monitor pressure differentials that indicate blockages or leaks. A digital flow meter verifies pump operation and volumetric flow rates. Finally, an automated valve is installed to execute shutoff commands when a leak state is detected.

The pressure value is derived from the raw ADC reading using the following equation, where 4095 represents the maximum ADC resolution and 5 the maximum sensor range in PSI.

$$pressure = \frac{v}{4095} \cdot 5 \text{ psi}$$

Equation 3.1.1: Pressure Equation

The flow rate is calculated from the sensor's output pulse frequency.

$$flow \text{ rate} = \frac{frequency}{7.5} \frac{L}{min}$$

Equation 3.1.2: Flow Equation

To establish normal operating conditions, the system computes a baseline average and standard deviation across both sensors during startup.

$$avg(t, n) = \frac{\sum(x)}{t \cdot n}$$

$$sd = \sqrt{\frac{\sum (x - \mu)^2}{n}}$$

Equation 3.1.3: Threshold Range

The function to calculate thresholds *CalculateThresholds(x, y)* accepts two integer arguments, x being how many seconds to sum the values, and y being how many measurements are taken per second. Therefore, the total amount of measurements equals x times y. A larger x value delays the process by x amount of seconds, but allows for more total values. A larger y collects more values per second, and doesn't add more time to the process, but is more resource intensive on the esp32 processor. In addition, there is a point where taking more measurements per second becomes unnecessary. For all times the function *CalculateThresholds(x, y)* is called, we use inputs (10, 10), and so the pressure baseline is computed from 100 samples over 10 seconds.

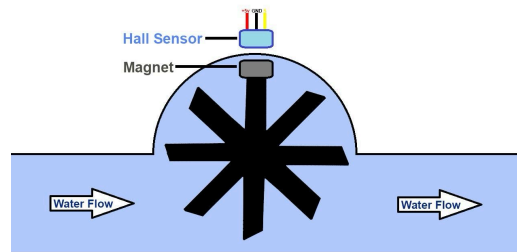


Figure 3.1.10: Internal View of Flow Sensor. [15]

The flow sensor operates differently from the pressure transducer. The YF-S201 flow sensor measures water flow using a small turbine inside the pipe that spins as water passes through it. This turbine has a tiny magnet embedded in it, and as it rotates, a hall effect sensor detects the changing magnetic field and generates electrical pulses. The faster the water flows, the faster the turbine spins, which means more pulses per second. Our microcontroller counts these pulses over 1 second, and converts the frequency specified by the datasheet as $F=7.5 * Q(L/min)$, with an error of 2%.

Since adequate time is required for the turbine to hit the hall sensor, the flow readings can't be polled at the same rate as pressure. The pressure readings are taken every 0.5 seconds, while the flow readings require 1 second. Either way, their values are sent every 0.5 seconds over MQTT.

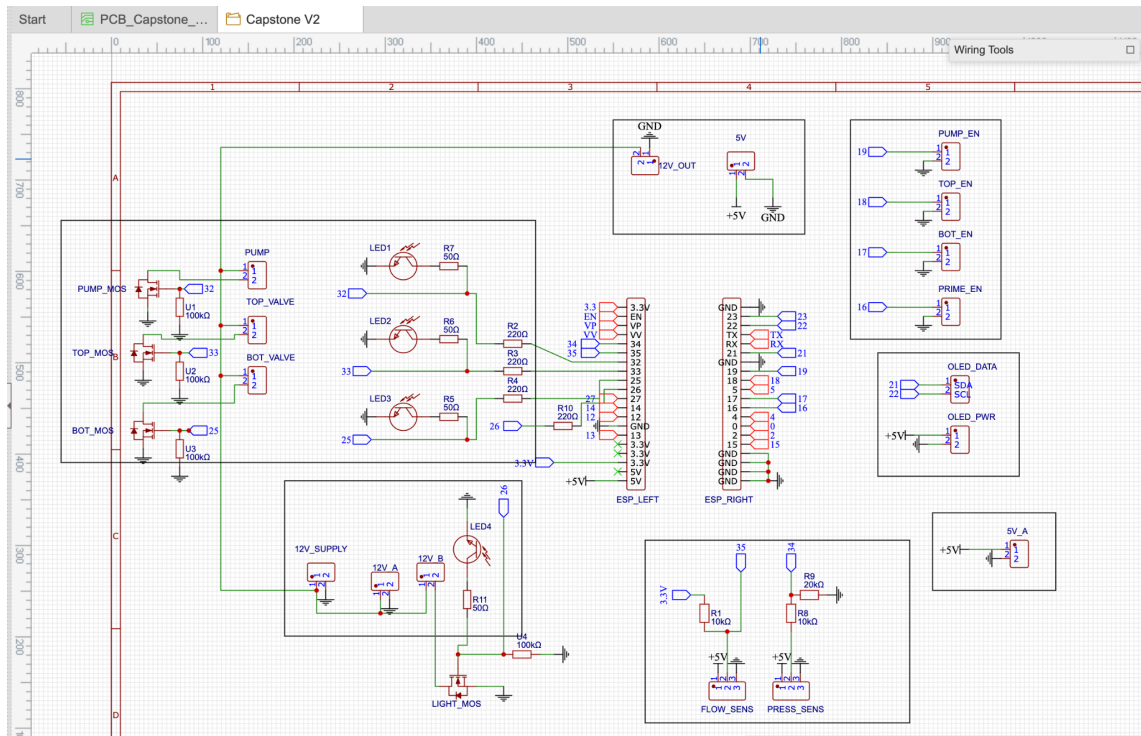


Figure 3.1.11: PCB Schematic designed in EasyEDA.

It's important to note, that the hall sensor can induce a 5V signal to the signal wire, which is not directly readable by the esp32 input pin. Since the maximum voltage the pin can read is 3.3V, we step 5V down to 3.3V using a simple voltage divider, as seen in figure 3.1.11 (bottom right black box, names are swapped).

Three detection states are derived from these baselines. The set threshold values were determined in reference to the hardware system developed in Figure 3.1.6, and would be adjusted to a dynamic model if used with systems with different specifications. A leak is flagged when pressure drops below a threshold based on the dynamic average.

$$Leak = avg - 0.2 \text{ psi}$$

Equation 3.1.4: Leak Threshold

A blockage is flagged when the pressure exceeds a threshold based on the dynamic average.

$$Blockage = avg + 2 \text{ psi}$$

Equation 3.1.5: Blockage Threshold

A critical threshold exists to catch rapid blockage events that may bypass the standard blockage threshold before it can register.

$$Critical = avg + 3 \text{ psi}$$

Equation 3.1.6: Critical Threshold

Each detection state triggers a distinct system response. Upon leak detection, the system halts operation entirely by closing the motorized valve and ceasing pump activity. The system cannot be restarted until the leak is manually addressed and the system is manually reset, as continued operation risks further water damage. Upon blockage detection, the system closes Latch 1 and opens Latch 2, diverting water from the main path to Path A, as labelled in Figure 3.1.6. This rerouting allows the system to continue operating despite the obstruction. If pressure continues to rise and the critical threshold is reached, the system immediately shuts off all flow to protect the pipes and hardware from pressure damage. In the hardware demonstration, a blockage is simulated using manual Latch 3, and a leak is simulated using manual Latch 4, where the leak drains into the same reservoir but vents to open air to produce a measurable pressure drop.

ESP32 Program Structure

The operational logic of the ESP32 guided by a Finite State Machine (FSM), as illustrated in Figure 3.1.12. The system transitions through 3 primary phases: initialization, normal operation and error handling/recovery.

Initialization state is responsible for connection to the WiFi hotspot and subsequently the MQTT broker on the same network. Once connected, the system is in a hold state until switch 3 is flipped. Upon trigger switch 3, the system activates the pumps and opens the valve to start the simulation. Then the ESP32 enters a 10 second threshold calculation phase, recording baseline pressure and flow data.

Normal Operation State is entered once the thresholds are established. In this state, pressure and flow values are measured every 0.5 seconds and published to their respective MQTT topics. If a user presses the Pause button via the dashboard, the system enters the pause detection state where sensor data collection continues but threshold comparisons are stopped. Detection resumes once the resume button on the dashboard is pressed.

Once a threshold is breached, it triggers one of the three error states each executing a unique hardware-level response, leak triggered by pressure drop, block triggered by pressure spike and critical triggered by immense pressure spike. Once the safety response is executed, the system enters a lockout state for the leak or obstruction to be addressed. Switch 4 is to be flipped to go into the reset system state and recalculate thresholds before going back to normal operation state.

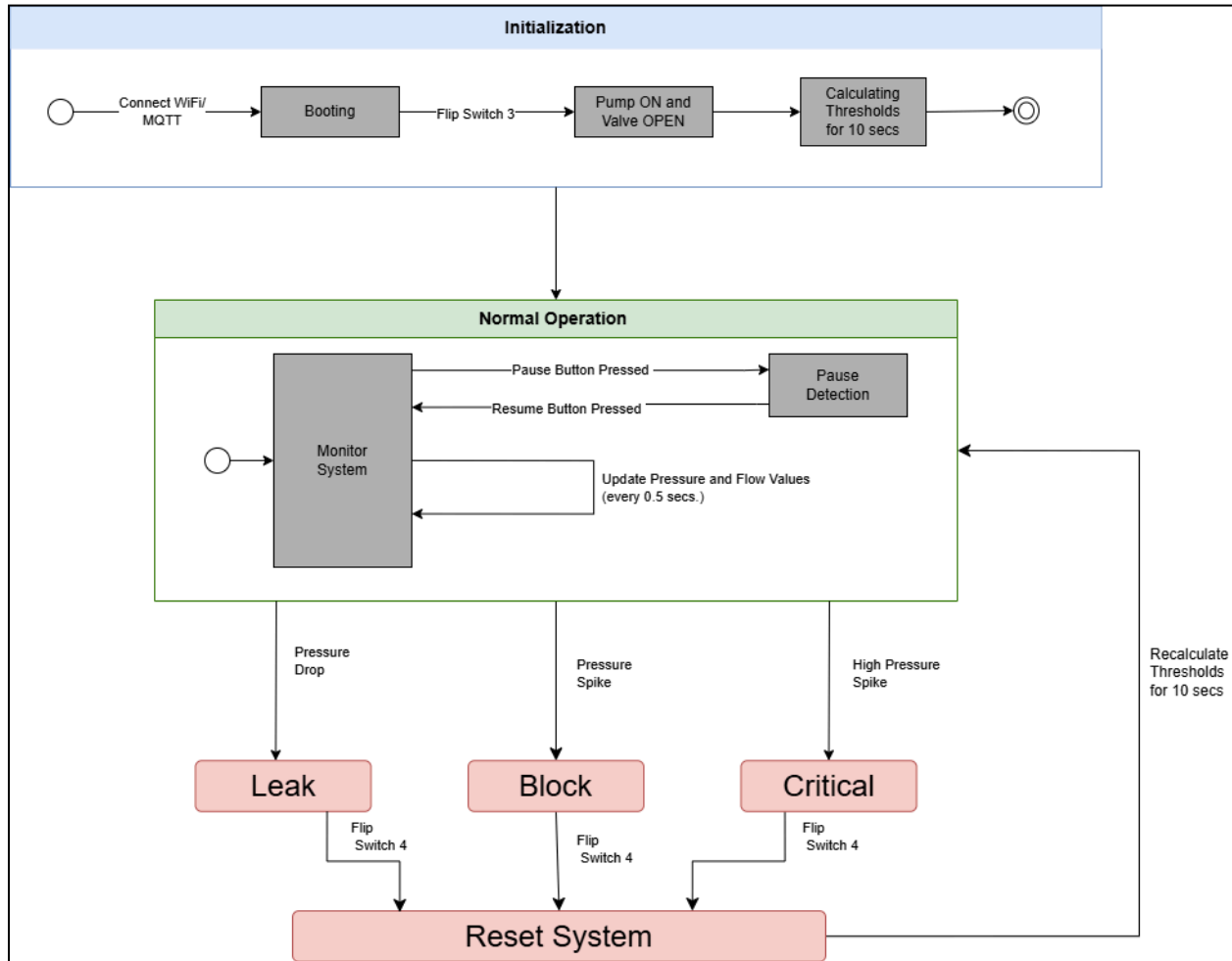


Figure 3.1.12: State Diagram of Overall Structure of ESP32 Program.

3.2 IoT Communication Protocol

The communication protocol relies on the Message Queue Telemetry Transport (MQTT) protocol, a lightweight publish-subscribe messaging transport ideal for connecting remote devices with a small code footprint, as well as minimal network bandwidth. The ESP32 operates as a client that connects to a Mosquitto MQTT broker running locally on a laptop. Due to network security restrictions on the university Wi-Fi which prevent device-to-device communication, a dedicated Wi-Fi hotspot hosted on an android mobile device is used to bridge the connection. The ESP32 publishes sensor data to specific topics on the broker, which is then routed to the backend system. Mosquitto command line interface is utilized for real-time tracking of message payloads and connection diagnostics by subscribing to topics.

Prototyping

To validate the IoT communication pipeline independently of hardware constraints, a simplified prototype was developed using an ultrasonic sensor and a temperature sensor on a breadboard. This approach avoided complications associated with water-based sensors which needed piping and controlled testing conditions.

The prototype allowed for rapid testing of data collection, microcontroller processing and transmission to dashboard via MQTT and backend infrastructure.

Findings from this prototype which were later applied to the final system included:

- Structuring sensor data for consistent backend processing
- Verifying end to end communication (ESP32 -> MQTT broker -> API -> database -> dashboard)
- Evaluating the real-time behaviour of the system

This step proved to be time-efficient as it allowed for development of functional software whose code structure was reused in the final system.

MQTT Broker

A MQTT broker was implemented to handle the communication between the ESP32 and the Gateway API. The MQTT broker was installed onto a laptop with a Mosquitto installation guide [16]. It was configured to be username and password protected, only allowing authorized users/devices to connect to the broker. The ESP32 sends system statuses along with component states/values through the broker. The dashboard has the ability to have remote control of certain components of the system, these include the pump, top and bottom valve, and pausing the leak/block detection. The dashboard sends these signals to the microcontroller by sending a HTTP POST request to /api/command which then publishes the command to the appropriate topic.

Table 3.2.1: MQTT Topics Publishers and Subscribers.

Purpose	Message Example	Topic	ESP32	Gateway API
Pressure Values	"2.5"	esp32/pressure	publish	subscribe
Flow Meter Values	"2"	esp32/flow	publish	subscribe
Pump State	ON / OFF	esp32/pump	pub/sub	pub/sub
Top Valve State	OPEN / SHUT	esp32/top_valve	pub/sub	pub/sub
Bottom Valve State	OPEN / SHUT	esp32/bottom_valve	publish	pub/sub
System Ready	YES / NO	system_ready	publish	subscribe
System Status	BLOCK/LEAK/ WAITING/IDLE	system_status	publish	subscribe
Pause Monitoring	PAUSE/RESUME	system_pause	subscribe	pub/sub
Blockage Threshold	"3.8"	block_threshold	publish	subscribe

Leakage Threshold	"0.5"	leak_threshold	publish	subscribe
Timestamp of Startup		esp32/latency_test	publish	subscribe

IoT Gateway API

The `bridge.js` program acts as a central bridge between the ESP32 hardware and frontend dashboard. The API handles data ingestion, command routing and localized processing. Connecting to an MQTT broker as a client to monitor all topics published by the ESP32, then inserts the incoming data to an SQLite database. The SQLite database is structured to append timestamps to all incoming data (sensor readings and sensor status), creating chronological time-series tables, allowing for the front end to plot data over time. It runs alongside it, an Express.js HTTP server to send this data to the React dashboard through GET requests.

The API enables remote control of certain hardware components. It controls components through `/api/command` endpoint, turning HTTP POST requests from the dashboard into MQTT messages published to the respective ESP32 topic. The frontend and backend operate on different local ports, default browser security does not allow cross-port communication, to bypass this, Cross-Origin Resource Sharing (CORS) is enabled. Additionally, the API acts as an edge-processing node, calculating the water usage over time through trapezoidal integration on flow readings per session. The program is designed to average out the two most recent flow readings and multiply that by the time passed between those two readings. This is summed with `totalLitres` variable to track water usage of session.

3.3 Software Architecture: Backend and Dashboard

The software design follows a client-server model underpinned by a microservices architecture. This theoretical approach decouples the client (dashboard) from the server, allowing for independent development and scalability.

Backend Services

The server-side application is developed in Node.js utilizing established libraries for SQL database management and RESTful API generation. The architecture utilizes an SQL database for the storage of sensor logs and system events. The backend communicates with the IoT layer and the frontend through a REST API. The MQTT-interfacing component functions as a distinct client, forwarding data from the broker to the REST API for processing and storage. This separation ensures that the core logic remains language-agnostic for the IoT implementation.

Web Dashboard

The user interface is a web application using the React framework and React Router. Figure 3.3.1 features the front page of the dashboard implementation.



Figure 3.3.1: Dashboard Front Page.

The dashboard is designed to provide users with visualization of real-time system statistics and manual control overrides for the valve hardware. The frontend communicates with the backend exclusively via API calls, ensuring a secure and structured data exchange pipeline.

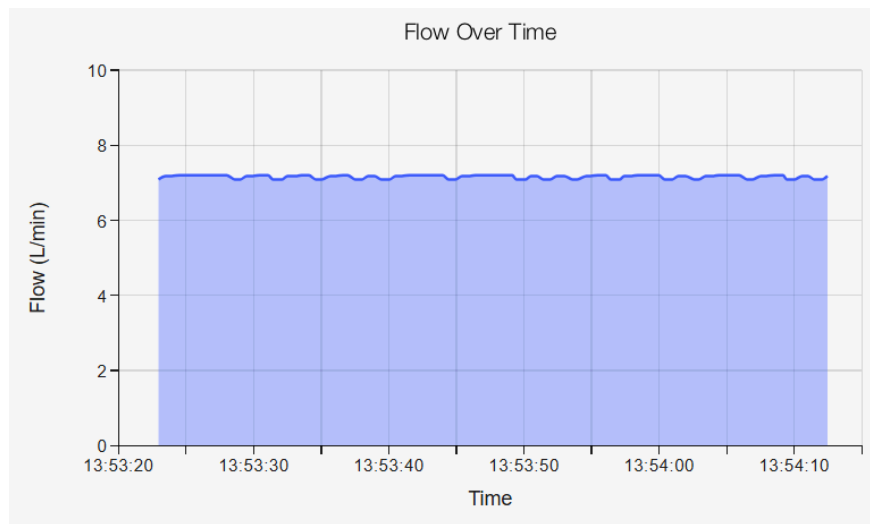


Figure 3.3.2: Total Flow Over Time Graph on Dashboard.

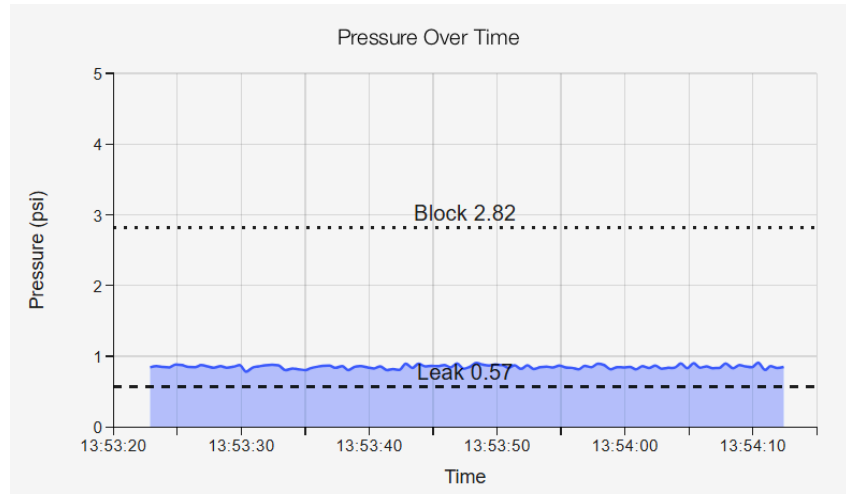


Figure 3.3.3: Total Pressure Over Time Graph on Dashboard.

Figures 3.3.2 and 3.3.3 display the graphical output of the system's flow and pressure readings over the span of the last 50 seconds, where the graph takes 100 readings, received every 0.5s. The dashboard also displays the current pressure, flow rate system status, system integrity, and water usage as of system startup. Additionally, the live *leak_threshold* and *blockage_threshold* is sent to the dashboard over a separate topic, and displayed as a horizontal dashed line. This way, it is very intuitive to see when each condition could be triggered, relative to the norm. Overall, this set of data provides users with clear and digestible information regarding their household's water usage, and can help users visualize changes in their water usage over time.

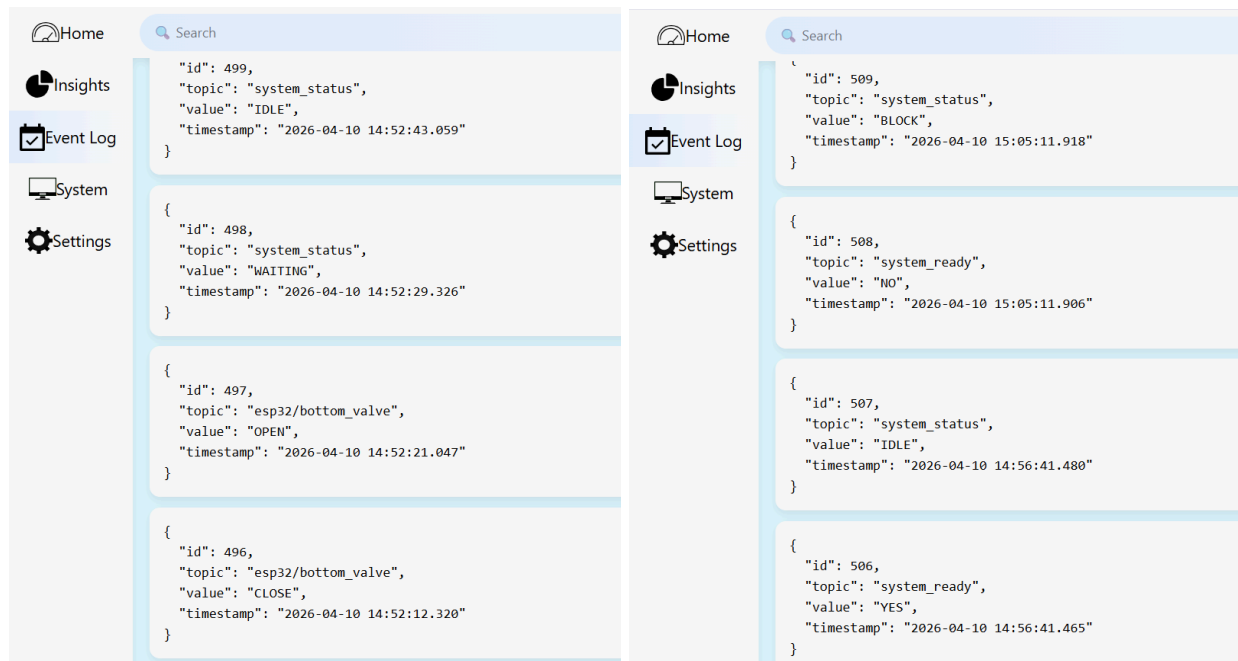


Figure 3.3.4: Dashboard Event Log.

Figure 3.3.4 displays the dashboard's event log, which prints out an itemized list of timestamped event occurrences received by the database. These include changes in valve states, changes in ready state, blockage detection, and leak detection.

To address the high false-positive rates and observed user frustration regarding existing commercial smart water systems [10], [11], rather than relying exclusively on a black-box machine learning algorithm that penalizes unpredictable user schedules, the system architecture prioritizes deterministic user control with a Manual Pause/Override function. This feature allows users to temporarily suspend the leak detection loop when there is an anticipated high-volume water event, such as prolonged shower use or lawn sprinklers. This ensures that legitimate water usage does not get misclassified as a leak, and subsequently avoids major user inconvenience.

4. Alternative Designs

The development of the Smart Water Management System involved several critical design decisions where alternative solutions were considered. This section provides a comparative analysis of these alternatives across hardware, software, and sensing methodologies to justify the final architecture.

4.1 Hardware-Focused vs. Software-Focused Design

Initially, the team weighed if this project would be best suited for a software-based design that would rely heavily on virtual simulations, or a hardware-focus that allowed for practical means of measurement and design. Ultimately, the hardware-based approach was selected over the software implementation for several reasons. Firstly, a physical hardware unit demonstrates the project specifications in a more effective and tangible manner, allowing for real-time visual inspection of fluid dynamics. There also existed difficulty during the research phase in finding a software that was capable of accurately modeling a complex, closed-loop hydraulic network that integrated real-time water flow, pressure physics, and interactive IoT controls simultaneously. Additionally, developing the physical hydraulic circuit using transparent PVC and a centrifugal pump provided immediate empirical data that software-only models could not replicate, particularly regarding the nuances of pressure fluctuations and sensor latency.

4.2 Manual Threshold Calibration vs. Machine Learning

While commercial systems utilize machine learning (ML) to establish usage baselines [9], this project opted for manual threshold calibration. Given the small scale of the demonstration unit, the extensive time required to train an ML model to obtain accurate thresholds was deemed impractical for the context of the project timeline. Manual thresholds allowed for immediate deterministic system responses without the high false-positive rates often associated with ML models misinterpreting dynamic human behavior. However, ML remains a viable alternative for future wide-scale implementations where the diversity of building specifications might warrant automated pattern recognition.

4.3 Sensor Selection: Pressure/Flow vs. Ultrasonic and Quality Sensors

Alternative sensing technologies were evaluated for detecting anomalies and monitoring water status. An integration option that was ultimately decided against was water quality testing. This would have included implementing water pH and temperature sensors in order to push water quality data to the user. However, the components required to add these features had a high out-of-pocket cost that was deemed unnecessary as it did not align with the project's primary objectives, which focused on leak and blockage detection, rather than chemical analysis.

There were also considerations for implementing ultrasonic sensors to monitor water levels in the pipes or tank, which would have provided an additional way to implement leak detection. However, the combination of the flow meter and pressure transducer provided a more efficient and cost-effective method for detecting both leaks and blockages in a pressurized system. Additionally, the PVC piping used had a relatively small diameter to its length, rendering

the effectiveness of an ultrasonic sensor to be significantly lower than on other systems that use large diameter pipes for their systems.

4.4 Leak Detection Methodology: Flow Differential vs. Pressure Sensing

A common alternative for leak detection is the use of two flow sensors at different points in a system to find a discrepancy in volumetric flow. For the purposes of this project, a single flow sensor was placed on the high point of the system, in series with the pump and the rest of the system. This reduced costs while ensuring the sensor could monitor the full system regardless of leaks or rerouting. The alternative of placing the sensor at the low side of the system was decided against, as with one flow meter the meter would either need to:

1. Be placed in series with the main line which would prevent the system from getting a full reading of the system flow when a leak occurs.
2. Be placed in series with the leak line (path B in Figure 3.1.6) which would prevent the system from getting a reading of the system flow outside of a leak state.
3. Connect the main line and the leak line to one node and connect the flow sensor in series with this node, which would require a significantly larger amount of piping and a larger display size in order to avoid splash back from one line into the other, which would compromise the accuracy of the simulation results.

The overall approach to detecting leaks in this project is pressure-based. An alternative option was to place one or more leak detectors underneath a section of pipe, which would inherently trigger with a single drop of water, but the location of a leak could never be predicted. A similar idea of creating a leak basin under critical sections of pipe was proposed, but similarly, the exact horizontal location would have to be assumed. With our approach, a leak can occur anywhere in the system, where the difference in pressure can be detected.

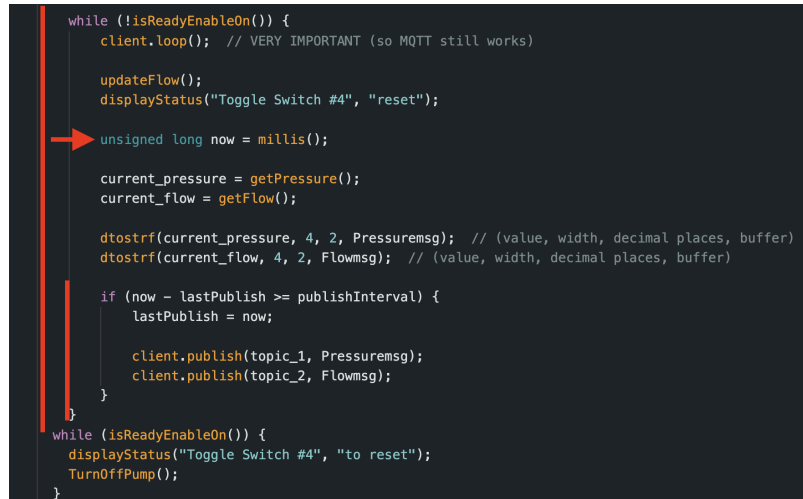
4.5 Data Analysis: Long-Term Usage Comparison

An alternative approach to detecting pinhole leaks is the long-term comparison of water usage data, where a user would be able to compare their daily water usage and average daily water pressure over a period of time, to compare to received water bills. This method was not implemented in the current state of the system because there was not a realistic way to simulate and track usage over the months or years required for such analysis. It would be more feasible to properly test the system's long-term data collection capabilities under a different project timeline, where it would not be system exhaustive to leave it constantly running.

4.6 ESP32 Locking: `delay()` vs. `millis()`

We encountered an obstacle when designing the program, where using `delay(x)` in the code would pause the entire chip, and wait for the delay to finish. In the meantime, no other code can be executed. The esp32 we used technically does have two separate cores, but assigning tasks two different cores would've been a risky challenge. To mitigate this issue, we used a feature called `millis()`, a function of the Arduino core library. Essentially, the chip keeps an

uptime timestamp value measured in milliseconds. And during the main *loop()*, and *if statement* constantly compares the difference between the current *millis()* time and the user-set interval.



```

while (!isReadyEnableOn()) {
  client.loop(); // VERY IMPORTANT (so MQTT still works)

  updateFlow();
  displayStatus("Toggle Switch #4", "reset");

  → unsigned long now = millis();

  current_pressure = getPressure();
  current_flow = getFlow();

  dtostrf(current_pressure, 4, 2, Pressuremsg); // (value, width, decimal places, buffer)
  dtostrf(current_flow, 4, 2, Flowmsg); // (value, width, decimal places, buffer)

  if (now - lastPublish >= publishInterval) {
    lastPublish = now;

    client.publish(topic_1, Pressuremsg);
    client.publish(topic_2, Flowmsg);
  }
}

while (isReadyEnableOn()) {
  displayStatus("Toggle Switch #4", "to reset");
  TurnOffPump();
}

```

Figure 4.6.1: Location of *millis()* Function in ESP32 Program.

Figure 4.6.1 shows an example within the loop, after a “blockage detected” condition is triggered, where the program waits for a rocker switch to be toggled from on to off, and back to on. While the button is waiting to be toggled off, we still wanted to have pressure and flow values reported over MQTT and shown on the dashboard. The classic solution is to iterate through a *for loop*, and set a millisecond delay between each loop. But again, during this delay no other code can execute. In this example, if it’s been 500ms since *now*, then publish the sensor values. Meanwhile the *while loop* runs continuously without pausing, and watching for the next rocker state in real time.

The more convenient alternative in this case was to integrate two separate esp32 modules, for true multithreading. Where one module would always wait and execute state changes, while the other could run the main program. Although *millis()* was a little cumbersome to code, it allowed us to execute any code in realtime without software delays

4.7 Native Application versus Website

The dashboard could have been made as either a website or as a native application. The main reasons for choosing a website are that a website could be easily accessed on different platforms (our team’s personal computers run on Windows, MacOS, and Linux) and it can be further developed to access online from anywhere.

4.8 React Router versus any other framework

The dashboard was created using React Router. React Router allows for the easy construction of websites through the construction of reusable javascript components, and due to its general popularity, has vast resources to provide information on dashboard implementation.

4.9 Node.js versus C# for the backend

An early version of the backend was written in C#, as in, so early that all that was present was a REST API. During the development of the MQTT section, a program that puts raw MQTT values straight into a SQL database was written using Node.js, so for ease of development the REST API was bolted onto this program instead.

4.10 PCB Design Improvements

Using a custom PCB was a great idea, but a few items would have been addressed if more time was given, and budget allowed.

- Add 400uF buck capacitors to the load terminals (pump and valves). During testing, we noticed that the 12V Led Strip under the water tank would flicker as soon as the pump gate pin went high. With further research, it was discovered that inductive loads cause inrush current upon startup. This current spike mainly stresses the power supply, but can also affect other components on the PCB by sagging the voltage of the 12V rail temporarily. In our case, a small dip in voltage doesn't affect the performance of other peripherals, including the CPU itself, which is protected by a buck converter. A buck capacitor would've provided immediate relief by dumping excess charge into the positive line.
- A barrel fuse rated for 5 amps soldered on the PCB would've been convenient, as it is easy to visually see inside the fuse. Our demo still includes a blade fuse, shielded by a rubber encasing.
- An audible indicator, such as a beeper, would've added an extra cue for when an error state occurred
- For a commercially ready product, SMD (surface mounted device) components would be preferred over through hole components, but for the purpose of the demo, most of the components were through hole.
 - Every time we wanted to flash a new sketch onto the microcontroller, the chip had to be pulled from its connection, and flashed separately. It would be convenient to plug in the USB-C cable while the board is attached to the PCB, but we noticed the buck converter would power on, caused by backfeeding 5V from the computer into the output terminals of the buck converter. A diode would've prevented this side-effect, although the output voltage of the buck converter would have to account for the forward voltage of the diode (~0.7V).
- In a real-world scenario, the PCB would be fully enclosed and shielded from the elements, especially in environments involving water. In our case, it was located high enough where water was not expected to infiltrate

5. Material/Component list

Tabulate the cost of all materials and components used in the project. Both unit cost and the quantity are required.

Table 5.1: Components and Associated Cost for Project.

Product	Unit Cost	Quantity	Final Cost
12V Water pump	\$28.49	1	\$32.19
Flow Meter	\$8.88	1	\$10.03
Pressure Sensor	\$18.77	1	\$18.77
X2 Esp32 Wroom	\$25.95	1	\$29.32
X50 M3 Heated Inserts	\$3.13	1	\$3.13
X50 M3 Heated Inserts	\$3.39	1	\$3.39
X50 M3 Brass Standoffs	\$4.69	1	\$4.69
X50 Stainless Steel Screws	\$2.47	1	\$2.47
X50 Stainless Steel Screws	\$2.59	1	\$2.59
12 V Power Supply	\$18.99	1	0
Barrel jack Adapter	borrowed	1	0
1/2 inch OLED display	borrowed	1	0
Buck Converter	borrowed	1	0
3D Casing/Brackets	borrowed	1	0
26 ft ft Clear PVC Pipes	\$29.89	1	\$33.78
10 pcs Rocker Switch	\$13.99	1	\$15.81
5pcs tee fittings barb	\$16.99	1	\$19.20
5pcs elbow barbs	\$10.29	1	\$11.63
2 pcs ball valves	\$23.99	1	\$27.11
20 pcs Stainless Steel Clamps	\$7.05	1	\$7.05
Motorized Ball Valve	\$27.54	1	\$27.54
Motorized Ball Valve	\$27.57	1	\$27.57
5 pcs (For Flow Meter) Adapter	\$7.49	1	\$7.49
(For Pressure Sensor) adapter	\$8.56	1	\$8.56
X6 Plastic 1/2 inch male to barb adapter	\$8.64	1	\$9.76
3/4 inch plywood Backboard	\$63.36	1	\$71.60
8 pcs Metal Brackets	\$11.93	1	\$11.93
X50 M5 Wood Threaded Inserts	\$5.03	1	\$5.03
X50 M5 4mm Heated Inserts	\$3.89	1	\$3.89
X30 M5 16 mm Screws	\$2.90	3	\$8.70
X20 M5 16 mm Screws	\$3.31	2	\$6.62
X20 M5 16mm Screws	\$2.43	2	\$4.86
X50 M5 washers	\$2.48	1	\$2.48

Product	Unit Cost	Quantity	Final Cost
12V Water pump	\$28.49	1	\$32.19
Flow Meter	\$8.88	1	\$10.03
Pressure Sensor	\$18.77	1	\$18.77
X50 M3 nuts	\$2.55	1	\$2.55
X50 M5 rubber washers	\$3.89	1	\$3.89
500 mL Methelyne Chloride	borrowed	1	0
3mm red LED	borrowed	4	0
4.5 mm Cast Acrylic	\$9.00	5	\$50.85
5 boards + shipping	\$4.83	1	\$4.83
Version 2: 5 boards + shipping	\$4.83	1	\$4.83
IRLZ44N	\$5.60	1	\$5.60
2P Terminal Block	\$2.50	1	\$2.50
2P Terminal Block	\$2.45	1	\$2.45
3P Terminal Block	\$2.92	1	\$2.92
Multimeter	borrowed	1	0
Soldering Iron	borrowed	1	0
Solder	borrowed	1	0
Breadboard	borrowed	1	0
Wires	borrowed	1	0
220-440 grit sandpaaper	borrowed	1	0
Ultrasonic Sensor	borrowed	1	0
DHT22 Sensor	borrowed	1	0

6. Measurement and Testing Procedures

This chapter details the measurement and testing procedures used to verify the functionality of the Smart Water Management System according to its design specifications. Testing was conducted across four primary areas, sensor data transmission and dashboard integration, IoT communication integrity, automated anomaly detection, and threshold calibration.

6.1 Sensor Data Transmission and Dashboard Integration

To verify that pressure and flow values were being correctly transmitted and displayed, sensor readings were observed live on the dashboard while the system was running under normal operating conditions. Correct real-time display confirmed that the full pipeline from ESP32 through the MQTT broker, Gateway API, database, and frontend was functioning as intended. Dashboard control buttons were also tested under two conditions, when the system was in the ready state and when it was not, either from a leak or a blockage. When the system was in the non-ready state due to a detected leak or blockage, all control commands including valve open/close, pump on/off and pause/resume detection were rejected server-side, producing no change in hardware state. When the system was in a ready state, all commands executed successfully, confirming the functional integrity of the bidirectional transmission path.

6.2 IoT Communication Integrity

To confirm that the ESP32 was publishing correctly to the MQTT broker, the Mosquitto command-line interface (CLI) subscribed to each topic to verify that message payloads arrived with the correct format and timing. The REST API was validated by cross-referencing database entries against dashboard-displayed values, confirming that the GET and POST requests were functioning correctly and that data written to the database matched what was being rendered on the frontend.

6.3 Anomaly Detection and Threshold Calibration

Leak detection was simulated by opening the manual Latch 4, as annotated in Figure 3.1.6. This vents water to open air within the reservoir, producing a measurable drop in line pressure. Blockage detection was simulated by closing manual Latch 3, which restricts flow along the main path. In both cases, the system responded as soon as pressure crossed its respective threshold. Anomaly detection operates independently of dashboard or API connectivity, where the ESP32 monitors sensor values and triggers responses locally. This ensures that the system maintains protective function at all times, even if the broader network is unavailable.

Threshold values were determined empirically given the operating constraints of the hardware. The pressure transducer has a measurement range from 0 PSI - 5 PSI, and under normal operation with the pump running at its fixed maximum flow rate, a stable average pressure was established. The manual latches used to simulate leak and blockage conditions were opened and closed repeatedly to identify the degree of deviation that produced a clear and

repeatable sensor response. A pressure drop from approximately 0.8 to 0.55 PSI corresponds to a volume loss per minute of roughly:

$$volume\ lost = (Q_{leak} - Q_{normal}) \cdot (\frac{\theta}{90})\ L/s$$

Equation 6.3.1

Where Q_{normal} is the system flow rate in normal state

Q_{leak} is the system flow rate in leak state

θ is the angle of the lever for Valve 4

This loss is sufficient to represent a detectable and significant anomaly under the system's operating scale, justifying a leak threshold set at 0.25 PSI below the established average. The blockage threshold was set at 2 PSI above the average, as this level of pressure increase clearly indicated a restriction without posing a risk of damage to the hardware. The critical threshold at 3 PSI above the average was established as a failsafe for rapid blockage events. On a larger deployment scale, these fixed offsets could be replaced with dynamic thresholds derived from the standard deviation of measured pressure values, scaled proportionally to the size of the installation and the expected variation introduced by typical usage patterns such as showers or sprinkler systems.

6.4 Deployment Considerations

The system is designed not as a compact consumer product but as an installation intended for contractors to integrate during the construction of new buildings. This approach allows the bypass rerouting functionality triggered on blockage detection to be incorporated directly into the building's plumbing layout without requiring invasive modification of existing pipes. The overall component cost is low and the installation footprint is flexible, as the system can be configured around any main supply line or major water activity pipe. Homeowners and landlords would have access to their water usage data upon move-in, providing access to information that would help with their water conservation efforts and reduced water bill cost. These properties suggest that a production version of this system could achieve reasonable yield and customer satisfaction across a range of residential and commercial building types.

7. Performance Measurement Results

7.1 Sensor Data Transmission and Dashboard Integration

Dashboards in non normal operation state is shown in Figures 7.1.1 - 7.1.4.



Figure 7.1.1: Dashboard Alerted of Leak Detected State.

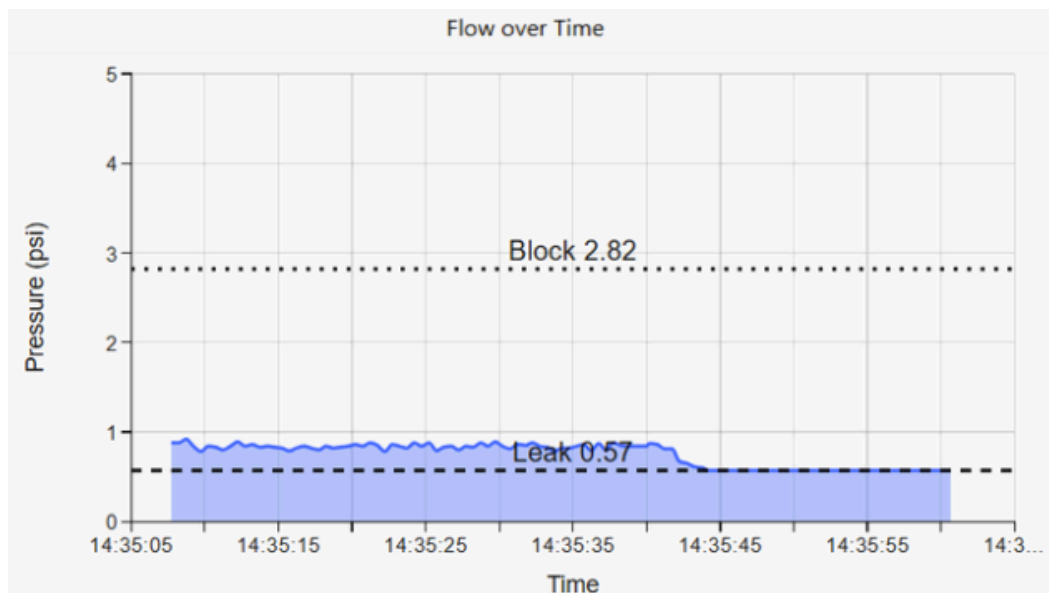


Figure 7.1.2: Pressure Graph on Dashboard when Leak State is Triggered.

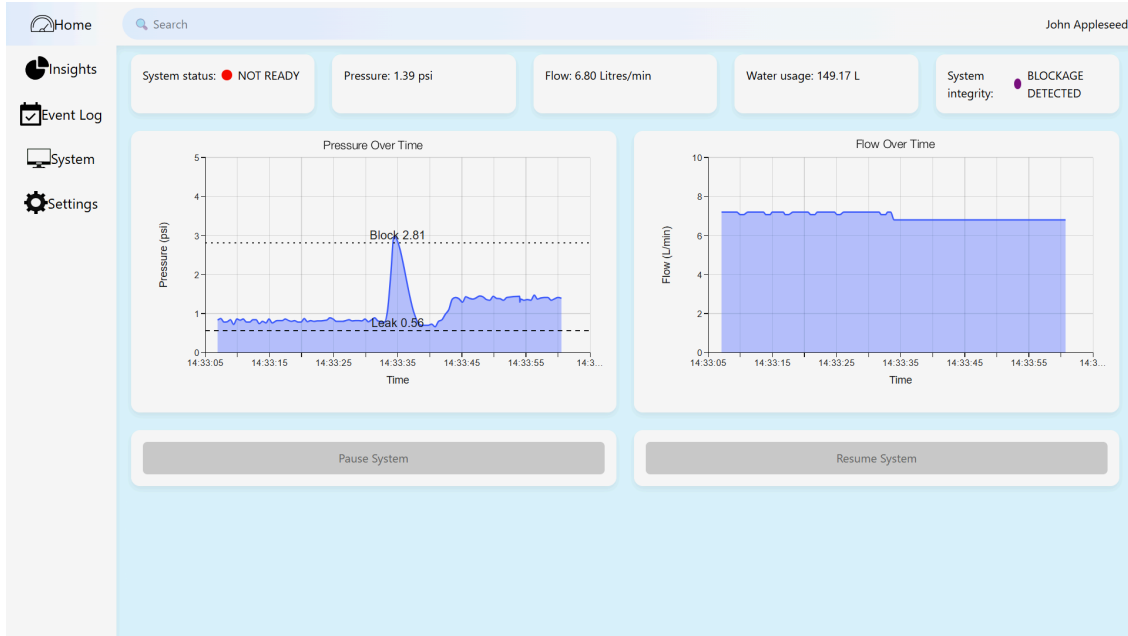


Figure 7.1.3: Dashboard Altered of Blockage State.

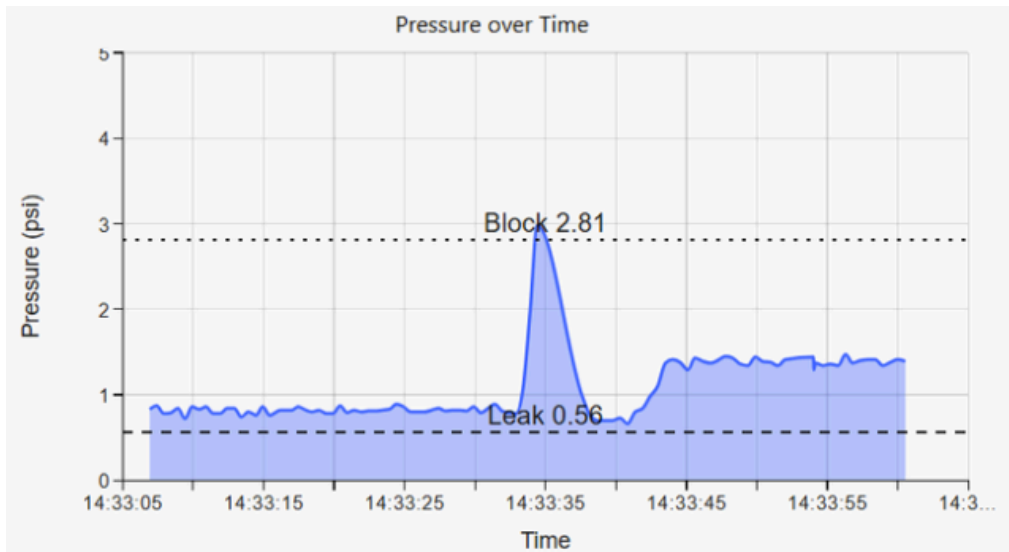


Figure 7.1.4: Pressure Graph on Dashboard When Blockage State is Triggered.

Table 7.1: Latency Between Communication Points Along Pipeline.

Communication Points	Time Duration (ms)
ESP32 -> Bridge receives data	134
Bridge -> Publish to Database	6
API -> Dashboard	20.80
End to End Latency	160.80

```
{
  "id": 568,
  "topic": "esp32/pump",
  "value": "ON",
  "timestamp": "2026-04-10 15:35:02.558"
}

{
  "id": 567,
  "topic": "esp32/pump",
  "value": "OFF",
  "timestamp": "2026-04-10 15:35:02.248"
}
```

Figure 7.1.5: Event Log When Conflicting Requests (Button Pressed) near-simultaneously.

Figure 7.1.5 displays the event log when two different users on the dashboard input a button command at the same time. This event is handled to prioritize the most recent request for execution.

7.2 IoT Communication Integrity

Figures 7.2.1 - 7.2.4 show data transmissions between end points of ESP32 and dashboard.



Figure 7.2.1: OLED Screen on Hardware System Displaying Live Sensor Values.


```

PS C:\Program Files\mosquitto> mosquitto_sub -i mosq_sub1 -t "esp32/pressure" -d
Client mosq_sub1 sending CONNECT
Client mosq_sub1 received CONNACK (0)
Client mosq_sub1 sending SUBSCRIBE (Mid: 1, Topic: esp32/pressure, QoS: 0, Options: 0x00)
Client mosq_sub1 received SUBACK
Subscribed (mid: 1): 0
Client mosq_sub1 received PUBLISH (d0, q0, r0, m0, 'esp32/pressure', ... (4 bytes))
0.92
Client mosq_sub1 received PUBLISH (d0, q0, r0, m0, 'esp32/pressure', ... (4 bytes))
0.88
Client mosq_sub1 received PUBLISH (d0, q0, r0, m0, 'esp32/pressure', ... (4 bytes))
0.80
Client mosq_sub1 received PUBLISH (d0, q0, r0, m0, 'esp32/pressure', ... (4 bytes))
0.81
Client mosq_sub1 received PUBLISH (d0, q0, r0, m0, 'esp32/pressure', ... (4 bytes))
0.83
Client mosq_sub1 sending PINGREQ
Client mosq_sub1 received PINGRESP
Client mosq_sub1 received PUBLISH (d0, q0, r0, m0, 'esp32/pressure', ... (4 bytes))
0.81
Client mosq_sub1 received PUBLISH (d0, q0, r0, m0, 'esp32/pressure', ... (4 bytes))
0.85

```

Figure 7.2.2: Mosquitto CLI subscribed to “esp32/pressure” receiving pressure values being published by ESP32.

```

Saved [esp32/pump]: ON
Saved [pressure]: 0.83 psi
Saved [flow]: 4.36 L/min
Saved [pressure]: 0.8 psi
Saved [flow]: 4.36 L/min
Saved [pressure]: 0.83 psi
Saved [flow]: 7.33 L/min
Saved [pressure]: 0.84 psi
Saved [flow]: 7.33 L/min
Saved [pressure]: 0.77 psi
Saved [flow]: 7.2 L/min
Saved [pressure]: 0.78 psi
Saved [flow]: 7.2 L/min
Saved [pressure]: 0.8 psi
Published [esp32/pump]: OFF
Saved [esp32/pump]: OFF

```

Figure 7.2.3: Terminal Output of bridge.js program printing out success message of writing data to SQLite database.

pressure_readings (35635 rows)			Export
SELECT * FROM 'pressure_readings' LIMIT 7944,30			Execute
id	value	timestamp	
7945	0.78	2026-04-08 14:33:32.918	
7946	1.08	2026-04-08 14:33:33.415	
7947	1.99	2026-04-08 14:33:33.916	
7948	2.98	2026-04-08 14:33:34.412	
7949	0.98	2026-04-08 14:33:37.619	
7950	0.76	2026-04-08 14:33:38.166	
7951	0.7	2026-04-08 14:33:38.700	

Figure 7.2.4: SQLite Database Pressure Values Displaying on Dashboard.

7.3 Anomaly Detection and Threshold Calibration



Figure 7.3.1: Minimum Angle of Lever of Valve 4 to Trigger Leak State.

Figure 7.3.1 features the exact latch angle of Valve 4 when the pressure falls below the leak detection threshold. This angle is 30° from the fully opened state, and 60° from the fully closed state.

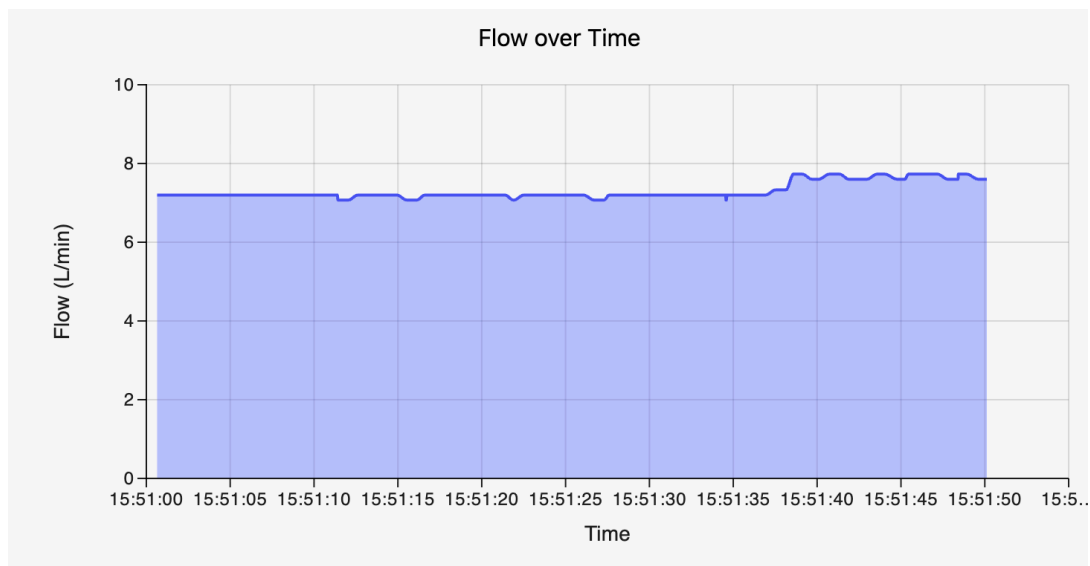


Figure 7.3.2: Flow Over Time Graph as System Transitions from Normal State to Leak State.

Figure 7.3.2 features a flow over time graph as the system transitions from the normal state to the leak state. The average flow rate in normal state is measured as 7.2 L/min. The average flow rate in the leak state is 7.6 L/min.

8. Analysis of Performance

8.1 Sensor Data Transmission and Dashboard Integration

To avoid overloading the REST API, the dashboard polls it every second, as opposed to how the sensor is sampled every five hundred milliseconds. However, because the dashboard pulls the hundred most recent entries in the database, this results in no loss of information. As the end-to-end latency of all communication from the microcontroller to the dashboard is less than the sampling interval, there will be no instances in which the information displayed on the dashboard is unacceptably delayed from what's happening in real life.

An issue encountered was the use of different time servers for different devices. As this data is measured with millisecond precision, the clocks of the microcontroller, the server, and the device opening the webpage in its web browser being a few milliseconds out-of-sync with one another caused considerable grief when attempting to measure the actual latency.

When two users press a button on the dashboard at 'the same time', the one which arrives latest ends up getting processed because the server carries out the most recent action. In Figure 7.1.5, one can see that this can be as little as 200ms.

Table 7.1 displays the system's latency between communication points along a timeline. The end-to-end latency was recorded to have a round time transmission of 160.80ms. This rapid response time qualifies the system as real-time functional [17].

8.2 IoT Communication

As shown in Figure 7.2.1, a pressure reading of 8.3 PSI is captured by the sensor and displayed locally on the OLED screen directly interfaced with the ESP32. This value is then encapsulated into a MQTT payload and published to the "esp32/pressure" topic. The successful transmission of this data packet to the Mosquitto broker is verified in Figure 7.2.2, showing the Mosquitto CLI output confirming the arrival of the exact 8.3 PSI. The data being successfully stored is documented in Figure 7.2.3, displaying the console output of the bridge.js file (API) confirming receiving the MQTT packet and adding it to the SQLite database.

Figure 7.1.4 illustrates the pressure values being successfully pulled via the "/api/pressure" endpoint at 14:33 timestamp and being displayed into a graph. When compared to the raw values in the SQLite pressure_readings table shown in Figure 7.2.4, the values and timestamps are identical, providing a direct correlation between the stored data and the API output. This consistency demonstrates that the bridge.js program is accurately fulfilling GET requests and serving JSON payloads to the front end.

8.3 Anomaly Detection and Threshold Calibration

Figure 7.3.2 features a flow over time graph which clearly displays the shift observed when transitioning from the normal state to the leak state. In a closed loop system, introducing a leak acts as a parallel pathway to the atmosphere, which reduces the total head pressure of the

hydraulic network. This reduction in resistance explains why the total system flow rate increases from the normal state (7.2 L/min) to the leak state (7.6 L/min) at the sensor node. The absolute difference between these two states is 0.4 L/min, which represents the maximum theoretical water loss when Valve 4 is fully opened at 90°.

However, the system does not require a fully open leak to trigger a response. As demonstrated in Figure 7.3.1, the system successfully detects the corresponding pressure drop when the valve is at an angle of 60° from the closed position (which is equal to 30° from the open position). Applying the empirical volume loss equation defined in Section 6.4, the exact water loss occurring at the moment of detection is calculated as follows:

$$Water\ Loss = (7.6 - 7.2) \cdot \frac{60}{90} = 0.264 \frac{L}{min}$$

Equation 8.3.1

This calculation proves that the system accurately detects leaks that are losing only 0.264 L/min. In the context of the total baseline flow (7.2 L/min) this means the hardware and control logic are sensitive enough to identify a volumetric diversion of less than 4% of the system's total flow capacity.

9. Conclusions

9.1 Summary

The project successfully designed, implemented, and tested an IoT-based Smart Water Management System capable of real-time fluid monitoring and automated fault mitigation. A physical hydraulic demonstration unit was constructed, integrating a centrifugal pump, pressure transducers, and a flow meter, all controlled by an ESP32 edge-computing microcontroller. The system met its primary objective by effectively detecting leaks and blockages using empirically calibrated static pressure thresholds, immediately triggering automated shutoffs and rerouting mechanisms without relying on cloud connectivity. Furthermore, a complete IoT data pipeline was established using the MQTT protocol to transmit real-time telemetry data to a Node.js backend, which is visualized on an interactive React-based web dashboard. Ultimately, the system provides a reliable, fail-safe water monitoring solution that operates independently of internet latency while still offering comprehensive remote user oversight.

9.2 Discrepancies

While the core functional objectives of the system were successfully met, a few discrepancies exist between the initial project proposal and the final deliverable.

In the original plan for the project, there was talk about the system being modular such that an end user could add valves and sensors to the system without modifying any source code. This was not added due to being deemed too complex to implement relative to the project specifications, resulting in a fixed sensor and value array for the final prototype.

There were also plans for a user-designed scheduling feature that would allow users to create automated schedules for individual valves. This was ultimately committed to focus development efforts entirely on the core anomaly detection logic and system stability.

Finally, the password protected remote control page was not fully implemented. The buttons were successfully implemented to control certain aspects of the system, displayed on a separate page than the main data page. The password protection of this page was not implemented due time constraints and it is not a high priority during project specification.

9.3 Major Difficulties Encountered

Throughout the development lifecycle, the team encountered a number of difficulties that affected the project timeline and scope.

A challenge arose from a lack of clear communication regarding code versions and developer responsibilities. This resulted in overlapping work and integration conflicts between the hardware functionality and the communication aspects of the ESP32 code.

Strict out-of-pocket cost and time limitations forced the team to simplify a number of physical aspects of the project. This resulted in bypassing more expensive water quality testing functionality, and limited the number of points of measurement that were available in the system.

9.4 Future Work

In a real deployment, the dashboard would be accessible over the internet, while here the dashboard is only accessible over LAN. This is because renting a VPS, registering a URL, and setting up the proper security for such a thing is beyond the scope of this project. In addition, controls for things such as the pump and individual valves should be put behind some sort of password protection.

The current system does not detect small/minor leaks, to address this, water usage data over a long period can be compared against normal usage values as mentioned in Chapter 4.5.

The long-term data collection mentioned above could additionally be expanded into a user-facing analytics module, providing end-users with detailed breakdowns of their water consumption patterns to assist in conservation efforts and help reduce utility bills.

References

- [1] E. Farah and I. Shahrour, "Water leak detection: A comprehensive review of methods, challenges, and future directions," MDPI, <http://mdpi.com/2073-4441/16/20/2975> Accessed on: November 22, 2025.
- [2] M. C. Donoso and Y. S. Kim, in Water security and the sustainable development goals, vol. 1, UNESCO, 2019, pp. 47–65
- [3] WINT, "The Carbon Impact of Water," Wint, <https://wint.ai/wp-content/uploads/2022/02/White-paper-Carbon-Impact-of-Water-Consumption-Final.pdf> Accessed on: November 22, 2025.
- [4] City of Toronto, "Water leaks: Costs & how to spot a leak," City of Toronto, <https://www.toronto.ca/services-payments/property-taxes-utilities/utility-bill/high-water-bill-leaks/water-leaks-costs-how-to-spot-a-leak/> Accessed on: November 22, 2025.
- [5] "How much does water damage repair cost in Toronto," Restoration Mate, <https://restorationmate.ca/how-much-does-water-damage-repair-cost-in-toronto/#:~:text=the%20repair%20process.,The%20Cost%20Of%20Water%20Damage%20Repair%20in%20Toronto,more%20for%20larger%20restoration%20jobs>. Accessed on: November 22, 2025.
- [6] M. K. Dixit, "Embodied impacts of buildings from energy-carbon-water nexus perspective: A case study of university buildings," ScienceDirect, <https://doi.org/10.1016/j.cles.2024.100108> Accessed on: November 22, 2025.
- [7] A. Rupiper, J. Weill, E. Bruno, K. Jessoe, and F. Loge, "Untapped potential: leak reduction is the most cost-effective urban water management tool," IOPscience, <https://iopscience.iop.org/article/10.1088/1748-9326/ac54cb> Accessed on: November 22, 2025.
- [8] Eve Systems. "Five places where Eve Water Guard adds a watertight layer of protection." Eve Blog. Available: <https://www.evehome.com/en/blog/five-places-where-eve-water-guard-adds-a-watertight-layer-of-protection>. Accessed on: Apr. 8, 2026.
- [9] Moen. "Flo Smart Water Monitor and Shutoff." Moen Canada. Available: <http://moen.ca/products/Flo-by-Moen/Flo-Smart-Water-Monitor-Shutoff---0-75-/900-001>. Accessed on: Apr. 8, 2026.
- [10] PR Newswire. "Moen Celebrates Earth Day with Meaningful Strides Toward its Mission Water Conservation Goal." ProQuest. Available: <https://www.proquest.com/wire-feeds/press-release-moen-celebrates-earth-day-with/docview/2393272854>. Accessed on: Apr. 8, 2026.
- [11] PR Newswire. "Frontdoor and Moen Expand Relationship with Addition of Six New States." ProQuest. Available:

- <https://www.proquest.com/wire-feeds/frontdoor-moen-expand-relationship-6-additional/docview/3163176335>. Accessed on: Apr. 8, 2026.
- [12] Amazon Customer Reviews. "Flo by Moen Smart Water Monitor and Automatic Shutoff Sensor." Amazon.ca. Available: https://www.amazon.ca/product-reviews/B00C03D01Q/ref=cm_cr_getr_d_paging_btm?ie=UTF8&pageNumber=7. Accessed on: Apr. 8, 2026.
- [13] A. Sudyk, "Constellation Project" unpublished manuscript, Toronto Metropolitan University, Toronto, ON, Canada, Mar. 2025.
- [14] jw5rcqz8w7. "Freenove ESP32 WROOM Board Pinout." Scribd, 7 June 2025, www.scribd.com/document/873016061/Freenove-ESP32-WROOM-Board-Pinout. Accessed 10 Apr. 2026.
- [15] "Choosing the Best Type of Flow Meter for Your Application," Dultmeier.com, 2021. <https://www.dultmeier.com/choosing-the-best-type-of-flow-meter-for-your-application/> (accessed Apr. 10, 2026).
- [16] Serhii Orlivskyi, "How to Install Mosquitto MQTT Broker on Windows," Cedalo, Mar. 25, 2025. <https://www.cedalo.com/blog/how-to-install-mosquitto-mqtt-broker-on-windows> (accessed Apr. 10, 2026).
- [17] "Real-Time," Suse.com, 2026. <https://www.suse.com/topics/definition/real-time/> (accessed Apr. 11, 2026).

Appendices

Server Source Code:

```
Bridge.js Program
const express = require('express');
const mqtt = require('mqtt');
const Database = require('better-sqlite3');
const cors = require('cors');
```

```
const app = express();
app.use(express.json());
app.use(cors()); // Allows React to call the API
```

//	_____	Database		Setup
----	-------	----------	--	-------

```
const db = new Database('sensor_data.db');

db.exec(`
  CREATE TABLE IF NOT EXISTS pressure_readings (
    id INTEGER PRIMARY KEY,
    value REAL NOT NULL,
    timestamp DATETIME DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime'))
  );
  CREATE TABLE IF NOT EXISTS flow_readings (
    id INTEGER PRIMARY KEY,
    value REAL NOT NULL,
    timestamp DATETIME DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime'))
  );
  CREATE TABLE IF NOT EXISTS system_updates (
    id INTEGER PRIMARY KEY,
    topic TEXT NOT NULL,
    value TEXT NOT NULL,
    timestamp DATETIME DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime'))
  );
  CREATE TABLE IF NOT EXISTS session_starts (
    id INTEGER PRIMARY KEY,
    timestamp DATETIME DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime'))
  );
  CREATE TABLE IF NOT EXISTS leak_threshold(
    id INTEGER PRIMARY KEY,
    value text NOT NULL,
    timestamp DATETIME DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime'))
  );
  CREATE TABLE IF NOT EXISTS block_threshold(
    id INTEGER PRIMARY KEY,
    value text NOT NULL,
```

```

        timestamp DATETIME DEFAULT (strftime('%Y-%m-%d %H:%M:%f', 'now', 'localtime'))
    );
    CREATE TABLE IF NOT EXISTS latency (
        id INTEGER PRIMARY KEY,
        value TEXT NOT NULL
    );
');

console.log('SQLite database ready');

//          _____ MQTT          Bridge
const mqttClient = mqtt.connect('mqtt://10.229.170.200:1883', {
    username: 'master',
    password: 'masterpass'
});

mqttClient.on('connect', () => {
    console.log('Connected to MQTT broker');

    db.prepare(`
        INSERT INTO session_starts DEFAULT VALUES
    `).run();

    mqttClient.subscribe('esp32/pressure', { nl: true });
    mqttClient.subscribe('esp32/flow', { nl: true });
    mqttClient.subscribe('esp32/pump', { nl: true });
    mqttClient.subscribe('esp32/top_valve', { nl: true });
    mqttClient.subscribe('esp32/bottom_valve', { nl: true });
    mqttClient.subscribe('system_ready', { nl: true });
    mqttClient.subscribe('system_status', { nl: true });
    mqttClient.subscribe('system_pause', { nl: true });
    mqttClient.subscribe('block_threshold', { nl: true });
    mqttClient.subscribe('leak_threshold', { nl: true });
    mqttClient.subscribe('esp32/latency_test', { nl: true });

});

mqttClient.on('message', (topic, message) => {
    if (topic === 'esp32/pressure' || topic === 'esp32/flow' || topic === 'block_threshold' || topic
    === 'leak_threshold') {
        const value = parseFloat(message.toString());
        if (isNaN(value)) {
            console.warn(`Invalid value received on ${topic}: ${message.toString()}`);
            return;
        }
    }
});

```

```
if (topic === 'esp32/pressure'){
  const insert = db.prepare('INSERT INTO pressure_readings (value) VALUES (?)');
  insert.run(value);
  console.log(`Saved [pressure]: ${value} psi`);
}
else if (topic === 'esp32/flow'){
  const insert = db.prepare('INSERT INTO flow_readings (value) VALUES (?)');
  insert.run(value);
  console.log(`Saved [flow]: ${value} L/min`);
}
else if (topic === 'block_threshold'){
  const insert = db.prepare('INSERT INTO block_threshold (value) VALUES (?)');
  insert.run(value);
  console.log(`Saved [block_threshold]: ${value}`);
}
else if (topic === 'leak_threshold'){
  const insert = db.prepare('INSERT INTO leak_threshold (value) VALUES (?)');
  insert.run(value);
  console.log(`Saved [leak_threshold]: ${value}`);
}
}
else if (topic === "esp32/latency_test"){
  const t0 = parseInt (message.toString());
  if (isNaN(t0)) {
    console.warn(`Invalid latency value: ${message.toString()}`);
    return;
  }
  const t1 = Date.now(); // when bridge receives the message
  const data = {t0: t0, t1: t1};

  const insert = db.prepare('INSERT INTO latency (value) VALUES (?)');
  insert.run(JSON.stringify(data));

  const t2 = Date.now();
  data.t2 = t2;

  const update = db.prepare('UPDATE latency SET value = ? WHERE id = (SELECT
MAX(id) FROM latency)');
  update.run(JSON.stringify(data));
  //console.log(`Saved [latency]: ${data}`);
}
else{
  const insert = db.prepare('INSERT INTO system_updates (topic, value) VALUES (?, ?)');
  insert.run(topic, message.toString());
  console.log(`Saved [${topic}]: ${message.toString()}`);
}
```

```

    }
  });

```

	REST	API	Endpoints	(pressure)
--	------	-----	-----------	------------

```

// Get all readings
app.get('/api/pressure', (req, res) => {
  const rows = db.prepare('SELECT * FROM pressure_readings ORDER BY timestamp
DESC').all();
  res.json(rows);
});

```

```

// Get latest reading only
app.get('/api/pressure/latest', (req, res) => {
  const row = db.prepare('SELECT * FROM pressure_readings ORDER BY timestamp DESC
LIMIT 1').get();
  res.json(row);
});

```

```

// Get last N readings e.g. /api/pressure/last/10
app.get('/api/pressure/last/:n', (req, res) => {
  const n = parseInt(req.params.n);
  const rows = db.prepare('SELECT * FROM pressure_readings ORDER BY timestamp DESC
LIMIT ?').all(n);
  res.json(rows);
});

```

```

// Delete all readings (useful for testing)
app.delete('/api/pressure', (req, res) => {
  db.prepare('DELETE FROM pressure_readings').run();
  res.json({ message: 'All readings deleted' });
});

```

	REST	API	Endpoints	(flow)
--	------	-----	-----------	--------

```

// Get all readings
app.get('/api/flow', (req, res) => {
  const rows = db.prepare('SELECT * FROM flow_readings ORDER BY timestamp
DESC').all();
  res.json(rows);
});

```

```

// Get latest reading only
app.get('/api/flow/latest', (req, res) => {

```

```
    const row = db.prepare('SELECT * FROM flow_readings ORDER BY timestamp DESC
LIMIT 1').get();
    res.json(row);
  });

// Get last N readings e.g. /api/pressure/last/10
app.get('/api/flow/last/:n', (req, res) => {
  const n = parseInt(req.params.n);
  const rows = db.prepare('SELECT * FROM flow_readings ORDER BY timestamp DESC
LIMIT ?').all(n);
  res.json(rows);
});

// Delete all readings (useful for testing)
app.delete('/api/flow', (req, res) => {
  db.prepare('DELETE FROM flow_readings').run();
  res.json({ message: 'All readings deleted' });
});

app.get('/api/system_updates', (req, res) => {
  const rows = db.prepare('SELECT * FROM system_updates ORDER BY timestamp
DESC').all();
  res.json(rows);
});

// Get latest reading for a SPECIFIC system topic (e.g., /api/system/system_status)
app.get('/api/system/latest/:topic', (req, res) => {
  const topic = req.params.topic;
  const rows = db.prepare('SELECT * FROM system_updates WHERE topic = ? ORDER BY
timestamp DESC LIMIT 1').get(topic);
  res.json(rows);
});

// Get updates for a SPECIFIC system topic (e.g., /api/system/system_status)
app.get('/api/system/:topic', (req, res) => {
  const topic = req.params.topic;
  const rows = db.prepare('SELECT * FROM system_updates WHERE topic = ? ORDER BY
timestamp DESC').all(topic);
  res.json(rows);
});

// Get the latest state of ALL system components at once
app.get('/api/system/latest/all', (req, res) => {
  const query = `
    SELECT topic, value, timestamp
    FROM system_updates`
```

```
    WHERE id IN (SELECT MAX(id) FROM system_updates GROUP BY topic)
  `;
  const rows = db.prepare(query).all();
  res.json(rows);
});

app.get('/api/water_usage/latest', (req, res) => {

  const session = db.prepare(`
    SELECT timestamp
    FROM session_starts
    ORDER BY timestamp DESC
    LIMIT 1
  `).get();

  if (!session) {
    return res.json({
      litres: 0,
      message: "no session found"
    });
  }

  const rows = db.prepare(`
    SELECT value, timestamp
    FROM flow_readings
    WHERE timestamp >= ?
    ORDER BY timestamp ASC
  `).all(session.timestamp);

  if (rows.length < 2) {
    return res.json({
      litres: 0,
      since: session.timestamp
    });
  }

  let totalLitres = 0;

  for (let i = 1; i < rows.length; i++) {

    const prev = rows[i - 1];
    const curr = rows[i];

    const t0 = new Date(prev.timestamp).getTime();
    const t1 = new Date(curr.timestamp).getTime();
```

```

    const deltaMinutes =
      (t1 - t0) / 60000;

    // trapezoidal integration
    const avgFlow =
      (prev.value + curr.value) / 2;

    totalLitres +=
      avgFlow * deltaMinutes;
  }

  res.json({
    value: totalLitres,
    timestamp: session.timestamp
  });
});

// Delete all readings (useful for testing)
app.delete('/api/system_updates', (req, res) => {
  db.prepare('DELETE FROM system_updates').run();
  res.json({ message: 'All readings deleted' });
});

//—— REST API Endpoints (threshold values)
app.get('/api/thresholds/leak/latest', (req, res) => {
  const row = db.prepare('SELECT * FROM leak_threshold ORDER BY timestamp DESC
LIMIT 1').get();
  res.json(row);
});
//—— REST API Endpoints (threshold values)
app.get('/api/thresholds/block/latest', (req, res) => {
  const row = db.prepare('SELECT * FROM block_threshold ORDER BY timestamp DESC
LIMIT 1').get();
  res.json(row);
});

app.get('/api/latency/latest', (req, res) => {
  const row = db.prepare(
    'SELECT * FROM latency ORDER BY id DESC LIMIT 1').get();

  const data = JSON.parse(row.value);

  res.json(data);
}

```


	REST	API	Endpoints	(Commands)
<pre> }); // // Send a command to the ESP32 (e.g., POST { "topic": "system_ready", "message": "PAUSE" }) app.post('/api/command', (req, res) => { const { topic, message } = req.body; if (!topic !message) { return res.status(400).json({ error: 'Both topic and message are required' }); } mqttClient.publish(topic, message, (err) => { if (err) { console.error(`Failed to publish message to \${topic}:`, err); return res.status(500).json({ error: 'Failed to publish to MQTT' }); } console.log(`Published [\${topic}]: \${message}`); res.json({ success: true, topic, message }); }); }); // </pre>				
<pre> app.listen(3000, () => { console.log('Server running on http://localhost:3000'); }); </pre>			Start	Server

ESP32 Source Code

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>
#include <U8g2lib.h>
#include <time.h>
#include <sys/time.h>

//Time
uint64_t latencyTimestampMs = 0;
bool timestampPublished = false;

// WiFi Settings
const char *ssid = "Plymouth";    // Same network as MQTT broker
const char *password = "rwee2763";

float pressure_sd = 0;
float flow_sd = 0;

char Pressuremsg[10];
char Flowmsg[10];

void ConnectWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);
    Serial.print("Connecting to WiFi");
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("\nWiFi Connected!");
    Serial.print("IP Address: ");
    Serial.println(WiFi.localIP());
}

// MQTT Broker Settings
```

```
const char *mqtt_broker = "10.229.170.200";

const char *topic_1 = "esp32/pressure";
const char *topic_2 = "esp32/flow";
const char *topic_3 = "esp32/pump";
const char *topic_4 = "esp32/top_valve";
const char *topic_5 = "esp32/bottom_valve";
const char *topic_6 = "system_ready";
const char *topic_7 = "system_status";
const char *topic_8 = "system_pause";
const char *topic_9 = "block_threshold"; //displays a horizontal line on the dashboard
const char *topic_10 = "leak_threshold"; //displays a horizontal line on the dashboard
const char *topic_11 = "esp32/latency_test"; //displays a horizontal line on the dashboard


const char *mqtt_username = "master";
const char *mqtt_password = "masterpass";
const int mqtt_port = 1883;


// Initialize Clients
WiFiClient espClient; // Changed from WiFiClientSecure
PubSubClient client(espClient);


unsigned long lastFlowUpdate = 0;
unsigned long lastPressureRead = 0;
const unsigned long pressureInterval = 500; // 0.5 seconds


// ESP32 pins
// Mosfet
const int pump = 32;
const int top_valve = 33;
const int bottom_valve = 25;
const int tank_strip = 26;


bool SyncTimeFromNTP() {
    // UTC
```

```
configTime(0, 0, "pool.ntp.org", "time.nist.gov");

struct timeval tv;
Serial.print("Syncing time from NTP");

for (int i = 0; i < 30; i++) {
    gettimeofday(&tv, NULL);

    // If time is valid, tv_sec will be a large Unix timestamp
    if (tv.tv_sec > 1700000000) {
        Serial.println("\nTime synced!");
        return true;
    }

    Serial.print(".");
    delay(500);
}

Serial.println("\nFailed to sync time");
return false;
}

uint64_t GetEpochMs() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return ((uint64_t)tv.tv_sec * 1000ULL) + ((uint64_t)tv.tv_usec / 1000ULL);
}

void PublishLatencyTimestampOnce() {
    if (timestampPublished) return;

    if (!client.connected()) return;

    latencyTimestampMs = GetEpochMs();

    char TimeStamp[32];
```

```
    snprintf(TimeStamp, sizeof(TimeStamp), "%llu",
              (unsigned long long)latencyTimestampMs);

    client.publish(topic_11, TimeStamp);
    Serial.print("Published latency timestamp: ");
    Serial.println(TimeStamp);

    timestampPublished = true;
}
```

```
void OpenTopValve() {
    digitalWrite(top_valve, HIGH);
    //client.publish(topic_4, "OPEN");
}
```

```
void OpenBottomValve() {
    digitalWrite(bottom_valve, HIGH);
    client.publish(topic_5, "OPEN");
}
```

```
void TurnOnTankStrip() {
    ledcWrite(tank_strip, 255); // full brightness
}
```

```
void TurnOffPump() {
    digitalWrite(pump, LOW);
}
```

```
void CloseTopValve() {
    digitalWrite(top_valve, LOW);
    //client.publish(topic_4, "CLOSE");
}
```

```
void CloseBottomValve() {
    digitalWrite(bottom_valve, LOW);
    client.publish(topic_5, "CLOSE");
}
```

```
}

void TurnOffTankStrip() {
  ledcWrite(tank_strip, 0); // full brightness
}

void TurnOnPump() {
  if (digitalRead(top_valve) == LOW && digitalRead(bottom_valve) == LOW) {
    displayStatus("Cannot turn", "on Pump");          //Protection for Pump
  } else {
    digitalWrite(pump, HIGH);
  }
}

const int pump_enable = 19;
const int top_enable = 18;
const int bottom_enable = 17;
const int ready_enable = 16;

bool isPumpEnableOn() { return digitalRead(pump_enable) == HIGH; }
bool isTopEnableOn() { return digitalRead(top_enable) == HIGH; }
bool isBottomEnableOn() { return digitalRead(bottom_enable) == HIGH; }
bool isReadyEnableOn() { return digitalRead(ready_enable) == HIGH; }

//Sensors
const int pressureSensorPin = 35;

const int flowSensorPin = 34;
volatile int pulseCount = 0;
float flowRate = 0;

void IRAM_ATTR countPulse() {
  pulseCount++;
}
```

```
}
```

```
float getPressure() {  
    int rawPressure = analogRead(pressureSensorPin);  
    float voltage = (rawPressure / 4095.0) * 3.3;  
    float PSI = (voltage / 3.3) * 5.0;  
    return PSI;  
}
```

```
//OLED screen
```

```
U8G2_SSD1306_128X64_NONAME_F_HW_I2C u8g2(U8G2_R2, U8X8_PIN_NONE);
```

```
void OLED(int x, int y, const char* text) {  
    u8g2.drawStr(x, y, text);  
}
```

```
// =====
```

```
// DISPLAY HELPERS
```

```
// =====
```

```
void drawCenteredText(const char* text, int y) {  
    int w = u8g2.getStrWidth(text);  
    int x = (128 - w) / 2;  
    u8g2.drawStr(x, y, text);  
}
```

```
void displayStatus(const char* line1, const char* line2) {  
    u8g2.clearBuffer();
```

```
    u8g2.setFont(u8g2_font_ncenB10_tr);  
    drawCenteredText(line1, 25);  
    drawCenteredText(line2, 40);
```

```
    u8g2.sendBuffer();
```

```
}

void displayPressureAndFlow(float psi, float flow, float Pthreshold, float Fthreshold) {
    char line1[30];
    char line2[30];
    char psiStr[10];
    char flowStr[10];

    dtostrf(psi, 4, 2, psiStr);
    dtostrf(flow, 4, 2, flowStr);

    sprintf(line1, "%s PSI", psiStr);
    sprintf(line2, "%s L/min", flowStr);

    u8g2.clearBuffer();

    u8g2.setFont(u8g2_font_ncenB10_tr);
    drawCenteredText(line1, 15);
    drawCenteredText(line2, 40);

    char buffer[50];
    snprintf(buffer, sizeof(buffer), "P:%.2f F:%.2f", Pthreshold, Fthreshold);

    u8g2.setFont(u8g2_font_5x8_tr); // smaller font
    drawCenteredText(buffer, 62);

    u8g2.sendBuffer();
}

int flag = 0;

void callback(char *topic, byte *payload, unsigned int length) {
    Serial.print("Message arrived in topic: ");
    Serial.println(topic);

    String message = "";
```



```
Serial.print("Message: ");
for (unsigned int i = 0; i < length; i++) {
    char c = (char)payload[i];
    Serial.print(c);
    message += c;
}

Serial.println();
Serial.println("-----");

if (String(topic) == topic_3) { // "esp32/pump"
    if (message == "ON") {
        if (digitalRead(top_valve) == HIGH) {
            TurnOnPump();
            flag = 1;
            unsigned long start = millis();
            while (millis() - start < 2000) {
                client.loop();
            }
        }
        else if (digitalRead(top_valve) == LOW) {
            TurnOnPump();
            flag = 0;
            unsigned long start = millis();
            while (millis() - start < 2000) {
                client.loop();
            }
        }
    }
    else if (message == "OFF") {
        TurnOffPump();
        flag = 1;
    }
}

else if (String(topic) == topic_4) { // "top_valve"
    if (message == "OPEN") {
```

```
    OpenTopValve();
    flag = 1;
}
else if (message == "SHUT") {
    CloseTopValve();
    flag = 0;
    unsigned long start = millis();
    while (millis() - start < 6000) {
        client.loop();
    }
}
}
else if (String(topic) == topic_8) { // "user_pause"
    if (message == "RESUME") {
        flag = 0;
    }
    else if (message == "PAUSE") {
        flag = 1;
    }
}
}

void SetupMQTT() {
    Serial.println("\n=== Starting ESP32 as Access Point ===");

    // 1. Create Access Point
    WiFi.softAP("ESP32_Broker", "12345678");
    Serial.println("Access Point Started");
    Serial.print("AP IP Address: ");
    Serial.println(WiFi.softAPIP());

    // 2. Setup MQTT (connecting to broker on THIS device)
    client.setServer(mqtt_broker, mqtt_port);
    client.setCallback(callback);
}

void ConnectMQTT() {
```

```
while (!client.connected() && (!isBottomEnableOn())) {  
    String client_id = "esp32-client-" + String(WiFi.macAddress());  
    Serial.printf("Connecting to MQTT as %s...\n", client_id.c_str());  
    displayStatus("Connecting", "to Client");  
  
    if (client.connect(client_id.c_str(), mqtt_username, mqtt_password)) {  
        Serial.println("MQTT Connected!");  
        client.subscribe(topic_3);  
        client.subscribe(topic_4);  
        client.subscribe(topic_5);  
        client.subscribe(topic_6);  
        client.subscribe(topic_8);  
    } else {  
        Serial.print("Failed, rc=");  
        Serial.print(client.state());  
        Serial.println(" - try again in 5 seconds");  
        int length = 5000;  
        for (int i = 0; i < 5000; i++) {  
            delay(1);  
  
            if (isBottomEnableOn()) {  
                return; // skip connecting, leave function immediately  
            }  
        }  
    }  
}
```

```
float current_pressure_threshold = 0;
```

```
float current_flow_threshold = 0;
```

```
float getFlow() {  
    return flowRate;  
}
```

```
void updateFlow() {  
    unsigned long now = millis();
```

```
unsigned long elapsed = now - lastFlowUpdate;

if (elapsed >= 1000) {
  noInterrupts();
  int count = pulseCount;
  pulseCount = 0;
  interrupts();

  float frequency = (count * 1000.0) / elapsed; // pulses per second
  float rate = frequency / 7.5;                // L/min

  lastFlowUpdate = now;
}
}

float leak_buffer = 0.25;
float block_buffer = 2.0;
float critical_buffer = 3.5;

void CalculateThresholds(int valuesPerSecond, int Seconds) {

  client.publish(topic_7, "WAITING");

  lastFlowUpdate = millis();
  noInterrupts();
  pulseCount = 0;
  interrupts();

  float PressureSum = 0;
  float FlowSum = 0;

  float currentPressureReading = 0;
  float currentFlowReading = 0;

  int totalCalculations = valuesPerSecond * Seconds;
  int delayPerSample = 1000 / valuesPerSecond;
```

```
int flowSamples = 0;
unsigned long lastFlowSample = millis();

// Limit sample count
const int MAX_SAMPLES = 200;
if (totalCalculations > MAX_SAMPLES) {
    totalCalculations = MAX_SAMPLES;
}

// Arrays to store samples
float pressure_data[MAX_SAMPLES];
float flow_data[MAX_SAMPLES];

displayStatus("Calculating", "Thresholds");
int animation_counter = 0;

char PressuremsgHIGH[10];
char PressuremsgLOW[10];
char Pressuremsg[10];
char Flowmsg[10];

for (int i = 0; i < totalCalculations; i++) {

    updateFlow();

    const char* animation = "";
    if (animation_counter == 0) {
        animation = "";
    } else if (animation_counter == 1) {
        animation = ".";
    } else if (animation_counter == 2) {
        animation = "..";
    } else if (animation_counter == 3) {
        animation = "...";
        animation_counter = 0;
    }
}
```

```
animation_counter++;
char buffer[30];
snprintf(buffer, sizeof(buffer), "Thresholds %s", animation);
displayStatus("Calculating", buffer);

// Pressure sample every loop
currentPressureReading = getPressure();
PressureSum += currentPressureReading;
pressure_data[i] = currentPressureReading;

// Flow sample once per second
unsigned long now = millis();
if (now - lastFlowSample >= 1000) {
    lastFlowSample = now;

    currentFlowReading = getFlow();
    FlowSum += currentFlowReading;

    if (flowSamples < MAX_SAMPLES) {
        flow_data[flowSamples] = currentFlowReading;
    }

    flowSamples++;

    dtostrf(currentPressureReading, 4, 2, Pressuremsg);
    dtostrf(currentFlowReading, 4, 2, Flowmsg);

    client.publish(topic_1, Pressuremsg);
    client.publish(topic_2, Flowmsg);
}

unsigned long start = millis();
while (millis() - start < delayPerSample) {
    client.loop();
}
}
```

```
// --- Final averages ---
current_pressure_threshold = PressureSum / totalCalculations;

if (flowSamples > 0) {
    current_flow_threshold = FlowSum / flowSamples;
} else {
    current_flow_threshold = 0;
}

// --- Standard deviation ---
float pressure_sd_sum = 0;
float flow_sd_sum = 0;

for (int i = 0; i < totalCalculations; i++) {
    float diff = pressure_data[i] - current_pressure_threshold;
    pressure_sd_sum += diff * diff;
}

for (int i = 0; i < flowSamples; i++) {
    float diff = flow_data[i] - current_flow_threshold;
    flow_sd_sum += diff * diff;
}

pressure_sd = sqrt(pressure_sd_sum / totalCalculations);

if (flowSamples > 0) {
    flow_sd = sqrt(flow_sd_sum / flowSamples);
} else {
    flow_sd = 0;
}

float pressure_blockage = current_pressure_threshold + 2;
float pressure_leakage = current_pressure_threshold - 0.25;

dtostrf(pressure_blockage, 4, 2, PressuremsgHIGH);
dtostrf(pressure_leakage, 4, 2, PressuremsgLOW);
```

```
client.publish(topic_9, PressuremsgHIGH);
client.publish(topic_10, PressuremsgLOW);

client.publish(topic_7, "IDLE");

Serial.print("Pressure mean: ");
Serial.println(current_pressure_threshold);
Serial.print("Pressure SD: ");
Serial.println(pressure_sd);

Serial.print("Flow mean: ");
Serial.println(current_flow_threshold);
Serial.print("Flow SD: ");
Serial.println(flow_sd);

PublishLatencyTimestampOnce();
}
```

```
const int pwmFreq = 5000;
const int pwmResolution = 8; // duty 0-255

void fadeTankStripOnce() {
  for (int duty = 0; duty <= 255; duty++) {
    ledcWrite(tank_strip, duty);
    delay(3);
  }

  for (int duty = 255; duty >= 0; duty--) {
    ledcWrite(tank_strip, duty);
    delay(3);
  }
}
```



```
}  
}  
  
void setup() {  
  
  pinMode(pump, OUTPUT);  
  pinMode(top_valve, OUTPUT);  
  pinMode(bottom_valve, OUTPUT);  
  pinMode(tank_strip, OUTPUT);  
  
  pinMode(pump_enable, INPUT_PULLUP);  
  pinMode(top_enable, INPUT_PULLUP);  
  pinMode(bottom_enable, INPUT_PULLUP);  
  pinMode(ready_enable, INPUT_PULLUP);  
  
  pinMode(pressureSensorPin, INPUT);  
  pinMode(flowSensorPin, INPUT);  
  attachInterrupt(digitalPinToInterrupt(flowSensorPin), countPulse, RISING);  
  lastFlowUpdate = millis();  
  
  ledcAttach(tank_strip, pwmFreq, pwmResolution);  
  ledcWrite(tank_strip, 0); // start off  
  
  Serial.begin(115200);  
  Wire.begin(21, 22);  
  u8g2.begin();  
  
  TurnOnTankStrip();  
  
  delay(2000);  
  
  ConnectWiFi();  
  
  SyncTimeFromNTP();  
}
```

```
SetupMQTT();
```

```
ConnectMQTT();
```

```
client.publish(topic_6, "NO"); //Ready
```

```
//State 3
```

```
while (isBottomEnableOn()) {  
    displayStatus("Flip Switch 3", "");  
    // waiting...  
}
```

```
//State 4
```

```
OpenBottomValve();  
for (int i = 7; i >= 0; i--) {  
    char buffer[32];  
    sprintf(buffer, "Opening: %d", i);  
    displayStatus(buffer, "");  
    unsigned long start = millis();  
while (millis() - start < 1000) {  
    client.loop();  
}  
}
```

```
//State 5
```

```
TurnOnPump();  
displayStatus("Turning", "on pump");  
unsigned long start = millis();  
while (millis() - start < 3000) {  
    client.loop();  
} //let water settle in pipes
```

```
    CalculateThresholds(10,10);
    if (!client.connected()) {
    ConnectMQTT();
}
client.loop();

client.publish(topic_6, "YES"); //Ready
client.publish(topic_7, "IDLE");

}
```

```
unsigned long previousTime = 0;
const unsigned long interval = 1000;
```

```
float current_pressure = 0;
float current_flow = 0;
```

```
void loop() {
```

```
    updateFlow();
```

```
    unsigned long now = millis();
```

```
    // --- read pressure every 500 ms ---
```

```
    if (now - lastPressureRead >= pressureInterval) {
```

```
        lastPressureRead = now;
```

```
        current_pressure = getPressure();
```

```
        current_flow = getFlow();
```

```
Serial.print("Pressure: ");
Serial.println(current_pressure);

dtostrf(current_pressure, 4, 2, Pressuremsg); // (value, width, decimal places, buffer)
dtostrf(current_flow, 4, 2, Flowmsg); // (value, width, decimal places, buffer)

displayPressureAndFlow(current_pressure,current_flow, current_pressure_threshold,
current_flow_threshold);

client.publish(topic_1, Pressuremsg);
client.publish(topic_2, Flowmsg);

}

if ((current_pressure <= (current_pressure_threshold - leak_buffer)) && flag == 0) { //Leak
Detected
    client.publish(topic_6, "NO");
    client.publish(topic_7, "LEAK");
    displayStatus("Leak Detected", "");
    TurnOffPump();
    for (int i = 0; i < 2; i++) {
        fadeTankStripOnce();
    }
    unsigned long lastPublish = 0;
    const unsigned long publishInterval = 500; // 0.5 sec

    while (!isReadyEnableOn()) {
        client.loop(); // VERY IMPORTANT (so MQTT still works)

        updateFlow();
        displayStatus("Toggle Switch #4", "reset");
```

```
    unsigned long now = millis();

    current_pressure = getPressure();
    current_flow = getFlow();

    dtostrf(current_pressure, 4, 2, Pressuremsg); // (value, width, decimal places, buffer)
    dtostrf(current_flow, 4, 2, Flowmsg); // (value, width, decimal places, buffer)

    if (now - lastPublish >= publishInterval) {
        lastPublish = now;

        client.publish(topic_1, Pressuremsg);
        client.publish(topic_2, Flowmsg);
    }
}

while (isReadyEnableOn()) {
    displayStatus("Toggle Switch #4", "to reset");
}

if (digitalRead(top_valve) == HIGH ) {
    displayStatus("Resetting", "Valves");
    CloseTopValve();
    delay(6000);
}

TurnOnPump();
TurnOnTankStrip();
displayStatus("Turning", "on Pump");
delay(3000);
CalculateThresholds(10, 10);
flag = 0;
client.publish(topic_6, "YES");
client.publish(topic_7, "IDLE");

}

else if (current_pressure >= (current_pressure_threshold + critical_buffer)) { //Critical
Detected
    displayStatus("Pressure too high", "system shutdown");
    client.publish(topic_6, "NO");
```

```
for (int i = 0; i < 2; i++) {
    fadeTankStripOnce();
}
TurnOffPump();

unsigned long lastPublish = 0;
const unsigned long publishInterval = 500; // 0.5 sec

while (!isReadyEnableOn()) {
    client.loop(); // VERY IMPORTANT (so MQTT still works)

    displayStatus("Toggle Switch #4", "reset");

    unsigned long now = millis();

    if (now - lastPublish >= publishInterval) {
        lastPublish = now;

        client.publish(topic_1, Pressuremsg);
        client.publish(topic_2, Flowmsg);
    }
}
while (isReadyEnableOn()) {
    displayStatus("Toggle Switch #4", "to reset");
}
CalculateThresholds(10,10);
client.publish(topic_6, "YES");
flag = 0;
}
else if (current_pressure >= (current_pressure_threshold + block_buffer)) { //Blockage
    Detected
    client.publish(topic_6, "NO");
    client.publish(topic_7, "BLOCK");
    displayStatus("Blockage", "Detected");

    OpenTopValve();
```

```
for (int i = 0; i < 2; i++) {
    fadeTankStripOnce();
    displayStatus("Rerouting Water", "");
}
TurnOnTankStrip();
CloseBottomValve();

unsigned long lastPublish = 0;
const unsigned long publishInterval = 500; // 0.5 sec

while (!isReadyEnableOn()) {
    client.loop(); // VERY IMPORTANT (so MQTT still works)

    updateFlow();
    displayStatus("Toggle Switch #4", "reset");

    unsigned long now = millis();

    current_pressure = getPressure();
    current_flow = getFlow();

    dtostrf(current_pressure, 4, 2, Pressuremsg); // (value, width, decimal places, buffer)
    dtostrf(current_flow, 4, 2, Flowmsg); // (value, width, decimal places, buffer)

    if (now - lastPublish >= publishInterval) {
        lastPublish = now;

        client.publish(topic_1, Pressuremsg);
        client.publish(topic_2, Flowmsg);
    }
}
while (isReadyEnableOn()) {
    displayStatus("Toggle Switch #4", "to reset");
    TurnOffPump();
}
OpenBottomValve();
CloseTopValve();
```

```
    for (int i = 7; i >= 0; i--) {  
    char buffer[32];  
    sprintf(buffer, "Resetting: %d", i);  
    displayStatus(buffer, "");  
    delay(1000);  
    }  
    TurnOnPump();  
    CalculateThresholds(10,10);  
    client.publish(topic_6, "YES");  
    client.publish(topic_7, "IDLE");  
    flag = 0;  
}
```

```
// if (isPumpEnableOn()) {  
//   TurnOnPump();
```

```
// if (!isPumpEnableOn()) {  
//   TurnOffPump();  
// }
```

```
// if (isTopEnableOn()) {  
//   OpenTopValve();  
// }
```

```
// if (isBottomEnableOn()) {  
//   OpenBottomValve();  
// }
```

```
if (!client.connected()) {  
    ConnectMQTT();
```



```
}  
client.loop();  
}
```