ChatGPT

Smart Fish Farm Monitoring System (Algeria) – Work Plan & PoC Solution

Introduction

Fish farming in Algeria is largely managed with **traditional methods**, relying on manual water quality checks and feeding routines [1]. This approach is labor-intensive and can miss rapid changes in pond conditions. To modernize aquaculture, we propose an **IoT-based smart pond monitoring system** that provides **continuous, real-time tracking** of water parameters and automates alerts for any unsafe conditions. The system is designed for **rural deployment with limited internet connectivity**, so it follows an **offline-first** approach: a lightweight web app that works offline, with data sync and alerts when connectivity is available [2] [3]. The user interface will be **farmer-friendly and bilingual (French/English)** to ensure accessibility for local farmers. In just **2 days** (with a team of 4), we will build a proof-of-concept MVP focusing on high-impact features: **simulated sensor data from ESP32 nodes, real-time MQTT data feeds, cloud processing with anomaly detection (Random Forest), and a responsive PWA dashboard with alerts**. Below, we detail the system architecture and a work plan for implementation.

System Architecture Overview

Our smart fish farming solution consists of several integrated components, from pond sensors to user-facing app. The high-level architecture includes:

1. **IoT Sensor Nodes (ESP32)** – Each fish pond has an ESP32 microcontroller with attached sensors (pH, temperature, turbidity, dissolved oxygen, etc.). These nodes simulate water-quality data and act as **wireless sensing units**.
2. **MQTT Communication** – Sensor nodes connect via Wi-Fi and publish readings to a cloud MQTT broker in real-time. MQTT is chosen for its low bandwidth usage and **low-latency, publish/subscribe** design [4], ideal for live sensor feeds.
3. **Cloud Backend & Data Storage** – A backend service subscribes to the MQTT topics for all ponds, ingesting the sensor data. The backend stores time-stamped readings in a database (using a free or low-cost cloud service). This layer also runs analytics/ML to detect anomalies and triggers alerts.
4. **Data Analytics & Anomaly Detection** – An ML module (using a Random Forest model) processes incoming data to identify problematic water conditions (e.g., out-of-range pH or low oxygen). Detected anomalies are logged and passed to the alerting system.
5. **Alerts & Notifications** – The system supports **real-time alerts** via both push notifications (to the web app or mobile device) and SMS text messages. If a pond's readings exceed safe thresholds, an alert is generated specifying the issue and pond ID. SMS ensures the farmer gets critical alerts even if the app or internet is down [5].
6. **Dashboard Web Application (PWA)** – Users access a **responsive web dashboard** to view live data, historical trends, and alerts. The app is built as a **Progressive Web App** so it can cache data for offline use and even send push notifications [3]. The UI supports **multiple languages (English and French)** and runs on basic smartphones (mobile-first design).

7. **Multi-Pond, Multi-User Support** – The system handles multiple ponds and multiple user accounts. Each user can log in to view their ponds' data. Ponds are identified by IDs or names, and the dashboard can filter or switch between ponds. Role-based access can be introduced (e.g. an admin overseeing all ponds vs. farmers with their own ponds) if needed.

Below we describe each component and how we will implement it in the MVP.

IoT Sensor Nodes – ESP32 Simulation

We will simulate **3–4 fish ponds** using ESP32 microcontroller boards. Each ESP32 will run a **MicroPython or Arduino C++ script** that periodically generates realistic sensor readings. The sensors simulated per pond include: **water temperature, pH level, dissolved oxygen (DO), turbidity**, and possibly **TDS or salinity**. These parameters cover the critical water quality indicators for fish health. For realism, the script will add slight random variations around normal ranges (e.g. pH ~7.5, temp ~25°C) and occasionally push values into abnormal ranges to test the anomaly alerts (e.g. pH dropping to 6.0 or DO falling below 4 mg/L). Each ESP32 connects to the farm's Wi-Fi or a hotspot and uses the MQTT protocol to send data.

MQTT Topic Convention: Each device publishes sensor data to a topic unique to its pond (for example, `farm1/pond3/temperature`, `farm1/pond3/pH`, etc., or a JSON payload to `farm1/pond3/data`). The ESP32 code will loop every N seconds (say, 30s) to read (simulate) sensor values and publish them. We also program the ESP32 to subscribe to any control topics if needed for future extensions (e.g. receiving a command to turn on an aerator), though for this MVP we focus on monitoring. As a reference, similar IoT farming projects have used ESP32 to gather sensor data and **publish it to an MQTT server in real time** [6]. We will leverage existing MQTT client libraries for ESP32 (MicroPython's `umqtt` or Arduino PubSubClient) to implement this.

Task allocation: One team member will write and deploy the MicroPython scripts to the ESP32s. They will test that each device can connect to Wi-Fi and consistently publish data to the MQTT broker.

MQTT Data Transmission & Cloud Broker

For real-time data streaming, we utilize **MQTT** (Message Queuing Telemetry Transport). MQTT's lightweight publish/subscribe model is well-suited for unreliable networks and IoT devices. It minimizes data overhead and works with low bandwidth, ensuring our system can handle **limited connectivity** scenarios gracefully [4]. We will set up a **cloud-based MQTT broker** – using either a free public broker or a cloud service:

- **Broker Choice:** To save time and cost, we can use a free MQTT broker service (e.g. **Eclipse Mosquitto's public broker** or **HiveMQ Cloud (free tier)**). These allow quick setup without server maintenance. Alternatively, one team member could deploy Mosquitto on a small VM (AWS EC2 free tier or a Digital Ocean droplet) if more control is needed. We will secure the broker with at least a username/password since multiple users/ponds are involved.
- **Publishing Frequency:** Each ESP32 will publish a JSON message (containing all sensor readings) to its topic every ~30 seconds. This provides near real-time updates without overloading the network. The MQTT QOS can be set to 1 (at least once delivery) to ensure data is delivered reliably over spotty connections.
- **Offline Buffering:** In case of internet dropout, the ESP32s can be coded to buffer data locally and re-send when connectivity returns. MQTT clients can also be configured to retain last will messages for

alerting disconnections. This way, data isn't lost during short outages. In industrial IoT, edge devices often implement **offline message buffering** so readings queue up and transmit when back online [7]. Our ESP32 nodes will use a similar strategy: if the MQTT publish fails (no connection), data is queued in memory (or SD card) and sent later.

Task allocation: The team member handling devices will also configure the MQTT broker and verify that sample messages from the ESP32 are received by the broker (using a tool like MQTT Explorer for testing). This sets the foundation for the backend to subscribe next.

Backend Server – Data Ingestion & Storage

The backend is responsible for **collecting the MQTT data, storing it, and processing it for analysis**. We will implement a lightweight server application (in Python or Node.js) that runs an MQTT **subscriber client**. Key aspects of the backend design:

- **MQTT Subscription:** The backend subscribes to all relevant topics (e.g. using a wildcard `farm1/+/data` to get all pond data under farm1). As sensor messages arrive, the backend parses the JSON payload to separate out sensor values and timestamps. This design allows adding more ponds easily – new topics will automatically be picked up.
- **Data Storage:** We'll use a **cloud database** or a simple storage solution to log the readings. Given the 2-day MVP timeline and cost constraints, a good choice is **Firebase Firestore** or **Firebase Realtime Database** (free tier) for quick setup and built-in sync. Firestore can store data as collections of `pond_readings`, and our backend can add documents for each new reading. Firestore also offers offline caching and real-time updates which can benefit our dashboard. Alternatively, a **SQL database** (e.g. SQLite or a small MySQL instance) could be used if the team is more comfortable with SQL – but that would require setting up an API to query data for the frontend. For speed, we lean toward Firestore or a simple NoSQL store accessible from frontend.
- **Data Model:** Each record will include pond ID, timestamp, and sensor values. We will also keep an **anomaly flag** or status for each record (normal vs anomaly), as determined by the ML module. Additionally, we maintain a list of recent alerts (to display in the UI and avoid duplicate notifications).

By having a backend server subscribe and store data, we **decouple** the devices from the UI. Even if users are offline, the backend accumulates data which can sync to them later. As noted in similar IoT pond systems, sensor devices send data to a **remote server where it is stored and analyzed for decision-making** [8]. We follow that approach here.

Task allocation: One developer will build the backend service. On Day 1, they will create the MQTT subscriber (using Python's `paho-mqtt` or Node's `mqtt` library) and test inserting incoming data into the chosen DB. They will also set up environment configurations (broker URL, DB credentials, etc.). By end of Day 1, we expect to see data flowing from the ESP32 sims through MQTT into the database (verifiable by inspecting DB entries).

Anomaly Detection & Machine Learning

A core feature is **automated anomaly detection** – identifying when pond conditions become problematic so that farmers can be alerted immediately. We plan to implement a **Random Forest** based model to analyze the sensor data. Our approach:

- **Feature Inputs:** The model will take the latest sensor readings of a pond (pH, temperature, DO, turbidity, etc.) as features. We may also include short-term trends (e.g. change in temperature over last hour) if time permits. The goal is to classify whether the pond's state is **normal** or **abnormal** (and possibly identify the specific issue).
- **Training Data:** As we likely don't have real historical data in 2 days, we will generate a synthetic dataset for training the Random Forest. We will use known safe ranges for each parameter to label "normal" data, and create "anomalies" where one or more parameters go outside safe bounds. For example, we know a healthy pond typically has pH ~6.5–8.5 and DO >~5 mg/L for most fish [9] [10]. So we label data with pH 5.8 or DO 3 mg/L as problematic. We can also leverage domain guidance from literature to define anomalies (e.g. high turbidity, extreme temperature).
- **Random Forest Model:** We'll use a Random Forest classifier (from scikit-learn) because it's quick to train and provides importance of features. The model can learn decision boundaries in the multi-dimensional sensor space better than simple threshold rules. In prior research, Random Forests have been used in aquaculture IoT systems to **forecast water quality trends and enable early intervention** [11]. In our MVP, the RF model will serve as a basic anomaly detector (essentially learning threshold combinations). If more time, we can train it to classify specific issues (e.g. "Low Oxygen" vs "High Ammonia"), but initially a binary anomaly flag is sufficient.

- **Real-time Inference:** The backend will run the Random Forest model on each new batch of readings that comes in (or at least at a regular interval per pond). If the model predicts an anomaly (or if any sensor crosses a critical threshold), we mark that pond as in "alert" state. In our design, a pond is automatically flagged **"problematic" if one or more parameters leave the ideal range**, and normal otherwise – this mirrors the logic in other smart pond apps [12]. We will also implement simple threshold checks alongside the model as a fallback (for transparency and failsafe). For instance, if pH < 6 or DO < 3, trigger an alert immediately even if the model hasn't flagged it.

- **Anomaly Logging:** When an anomaly is detected, the event is recorded (with timestamp, pond ID, and reason) in an "Alerts" collection in the database. This will be used to display alerts on the dashboard and to avoid duplicate notifications (e.g., we won't send another SMS for the same issue every minute – maybe one alert and then require the condition to clear before re-alerting).

Notably, a recent IoT aquaculture system (EcoGuard) combined Random Forest models for water quality prediction with real-time monitoring via MQTT and mobile alerts [11]. Our approach is simpler but along the same lines: **Random Forest ML for intelligent monitoring** integrated with an IoT data pipeline.

Task allocation: One team member (with ML expertise) will prepare the anomaly detection. On Day 1, they can script the generation of synthetic training data and train a Random Forest model (this can be done quickly with scikit-learn). On Day 2, they will integrate the model into the backend: likely exporting it (e.g. as a pickle or JSON of trees) and loading it in the backend code to run predictions on incoming data. They will test the anomaly logic by feeding some abnormal readings to ensure it flags correctly (e.g., simulate a DO drop and see that an alert is logged/sent).

Real-Time Dashboard (Web App)

We will develop a **web-based dashboard application** to visualize pond data and notify the farmers of any issues. This will be a **responsive web app** (accessible via browser on phone or PC) and built with an **offline-first** mindset to accommodate spotty internet. Key features of the dashboard:

- **Live Data View:** The main screen will list the ponds (or allow selection of a pond) and show current sensor readings for each (pH, temp, DO, etc.). We will highlight readings that are outside normal range (e.g., in red). This overview lets the farmer see all ponds' status at a glance. For example, the NusaPond app's dashboard shows multiple ponds and even categorizes them by health status (normal vs problematic) [13] – we aim for a similar multi-pond snapshot.
- **Historical Charts:** Users can tap a pond to view detailed charts of each parameter over time (last 24 hours, last week, etc.). We'll implement a simple chart view using a library like **Chart.js** or **Google Charts** to plot sensor values. This helps in identifying trends (e.g. steadily rising temperature). If time is short, we will focus on at least showing a history graph for the most critical parameter (say temperature or DO) and provide textual logs for others.
- **Alerts & Notifications:** The UI will prominently display alerts. We can have a section or pop-up for "Active Alerts" listing any ponds currently in alert state and the cause (e.g. "Pond 2: Low oxygen level!"). Past alerts could be viewable in a log. If a new anomaly comes in while the user is online, the app (via WebSocket or Firebase realtime updates) will show a notification icon or banner. Additionally, since the app is a PWA, we can utilize the **browser push API** or **Firebase Cloud Messaging (FCM)** to push notifications even when the app is not open, provided the user has internet [3]. This duplicates the SMS functionality but via data/internet.
- **Multilingual UI:** The interface will support **English and French** for the MVP. We'll externalize all text labels and messages, and provide a language toggle (or auto-detect browser language if French, else default to English). For example, buttons like "Refresh" / "Actualiser", or an alert "Low oxygen" / "Oxygène bas". Algeria's farmers often speak Arabic dialects, but French is common in technical contexts; we ensure French is available as requested. The design will use simple icons and clear color-coding so even with basic literacy the user can understand statuses (green = good, red = warning, etc.). We must also ensure the layout accommodates different text lengths in each language (following best practices for multilingual UI [14] [2]). In the future, adding Arabic is possible – our framework will be ready for it.

- **Offline Capability:** Using PWA features, we will cache the app's static assets and last known data so that **if the user opens the app with no internet, they can still see the most recent synced readings and alerts**. We'll implement a Service Worker to cache the HTML/JS/CSS and use IndexedDB or localStorage to save recent data. This "offline mode" allows farmers to at least view historical info when offline, and once connectivity is back, the app will sync new data in the background. This approach aligns with the **"Offline First"** principle for progressive apps, crucial for rural areas [3]. It ensures a fast, seamless UX regardless of network quality.

- **Usability for Farmers:** The UI will be kept **very simple and mobile-friendly** (responsive design). We will use large fonts, high-contrast colors, and possibly graphics (e.g., a fish icon for each pond). Navigation will be minimal: just a dashboard page and maybe a settings/profile page. The design will consider that users might have older devices or slow internet, so pages will be lightweight (few images, optimized assets) [2]. We will test on a smartphone to ensure it's easily usable (tapping targets are big enough, etc.).

Tech stack: We can quickly build the frontend using a framework like **React** or **Vue.js** (which the team is familiar with) and leverage their ecosystem for PWA support (both have templates or plugins for PWA). If time is extremely limited, even a simple HTML/JavaScript static page using CDN links (for Chart.js, etc.) could do, coupled with Firebase for data (Firebase can directly feed data to a JS app, simplifying the need for our own API). Given our 2-day window, we aim to use whichever approach yields a functional UI fastest.

Task allocation: Two team members will focus on the frontend: one on the overall UI design and components, another on integrating data and ensuring offline/push features. On Day 1, they'll set up the basic UI layout (using dummy data) and implement language toggling. On Day 2, they'll integrate live data (via the backend API or directly via Firebase SDK) and test the PWA offline capability and notifications. By the end, we expect a farmer to log in and see their pond statuses updating in near-real-time on the dashboard.

Alerts and Notification Mechanisms

Timely alerts are critical for this system to provide value. We implement two complementary alert channels: **push notifications in the app, and SMS for out-of-band alerts.**

- **Push Notifications:** The web app (PWA) will register for push notifications. We can use **Firebase Cloud Messaging (FCM)** as it offers a free and easy way to send pushes to web clients. When the backend flags an anomaly, it will send a message via FCM to the specific user's device (we will have obtained the user's FCM token when they logged in or subscribed). The push notification will contain a short message like "Alert: Pond 3 temperature high at 30°C" in the user's language. Clicking it opens the app to the detailed view. If building a full FCM integration is too heavy for the MVP, we at least ensure the Service Worker can generate a local notification when new data arrives indicating an issue. Push notifications give an **app-like experience** and are part of what makes PWAs powerful for remote monitoring [3].

- **SMS Alerts:** Given that internet connectivity may be intermittent, SMS provides a reliable fallback since basic GSM coverage might be available even when data is not. Our backend will integrate with an SMS gateway to send text alerts to the farmer's phone. In Algeria, we can use an international service like **Twilio** (which supports sending SMS to Algerian networks [15] [16]) or a local SMS provider API (several local telecoms offer SMS APIs). For the MVP, Twilio's trial or low-cost pay-as-you-go can be used due to ease of setup. Whenever a critical threshold is crossed (as determined by the anomaly module), the backend will trigger an SMS with a clear message: e.g. "Fish Farm Alert: Pond 2 dissolved oxygen low (3.5 mg/L). Check aerators. – SmartPondSystem". The message includes which pond and what the issue is in simple terms.

This approach follows the precedent of other systems embedding GSM/SMS for aquaculture alerts. A prior study added a GSM modem to **send SMS notifications to the farmer when the pond is in a critical condition, which is especially helpful if the farmer has no internet at that moment** [5]. In our design, the cloud backend replaces the GSM modem by using an SMS API, but the benefit is the same – emergency alerts get through via cellular network.

- **Thresholds for Alerts:** We will define safe ranges (possibly configurable per pond or species) and only send alerts when values are truly problematic to avoid spamming the user. For example, if pH drifts slightly to 6.3 (just below ideal), we might *not* immediately SMS, but if it drops below 6 or changes rapidly, then we do. Similarly, a mild temperature rise isn't urgent, but crossing, say, 30°C

might be. The Random Forest can encapsulate some of this logic, but we will also implement straightforward rules for absolute cut-offs. This ensures important alerts aren't missed – essentially combining **data-driven ML alerts with rule-based fail-safes**.

- **Alert Frequency and Clearing:** We will prevent alert storms by perhaps requiring a condition to be continuously abnormal for a few minutes before alert, or by not repeating the exact same SMS alert within a short window. An alert can be considered "active" until the readings return to normal, at which point we could send a resolution message or at least mark it cleared in the app.

Task allocation: The backend developer will integrate the Twilio (or SMS API) on Day 2. This involves setting up API keys and writing a function to send SMS given a phone number and message. We will test it with our own phone first. The frontend developers will implement the push notification subscription in the app (likely also Day 2), working with the backend person to send a test push. Testing will include triggering a fake alert and ensuring an SMS is received and a push notification pops up.

Offline-First and Multilingual Support

Offline-First: From the start, we design the solution to tolerate internet outages, since rural areas in Algeria may have unreliable connectivity. On the device side, as mentioned, the ESP32 will buffer and send data when possible [7]. On the client side, the PWA will use cached data and allow basic functionality without a network. Concretely, the service worker will cache the last X hours of data for each pond in local storage. If the user opens the app offline, it will load the cached readings and indicate "offline mode" (so they know data may be stale). They can still view trends and past alerts that were saved. Any actions that require network (like acknowledging an alert or switching user accounts) will be queued or disabled until connection resumes. This way, the app always provides **some value even when offline**, aligning with PWA best practices for rural deployment [3].

Data Sync: Once connection is available, the app will sync new readings. If using Firebase realtime DB or Firestore, a lot of this sync is automatic – the client SDK will get updates when online, and we can leverage that. We will also implement a "Last updated at HH:MM" indicator so the user knows how fresh the data is.

Multilingual UI: Our MVP will be bilingual (English and French). We will use a simple i18n approach: a JSON file for each language with all UI strings. The user can toggle language in the settings (with the default likely French, given local usage, but English available for international team members or future scaling). We must ensure the UI layout handles French vs. English text lengths. For example, French phrases can be longer; we might use slightly smaller font or flexible width containers. We have considered multi-language from the wireframing stage, so the design **won't break when switching languages** [14]. In future, adding Arabic or Tamazight dialect can be done by adding another JSON and perhaps changing font if needed (since Arabic is RTL, we'd also ensure our CSS can flip for RTL languages – though that may be beyond MVP scope).

Crucially, by delivering content in the farmers' preferred language and designing for **low bandwidth and older devices** [2], we maximize the system's adoption and usability in the target context.

Task allocation: Multilingual support will be handled by the frontend team on Day 1 (setting up the language files and toggle). Offline caching will be set up on Day 2 once the data flow is in place – likely the person who did backend can assist with making sure the PWA can fetch some recent data to cache upon loading.

User Management and Security

Since the system must support **multiple users and multiple ponds**, we will include a basic authentication and user management setup. For the MVP, we can use Firebase Authentication (which has a quick email/password auth and even phone number OTP if needed). This saves time versus building auth from scratch. Each user account will have certain ponds associated with it. We'll maintain a simple mapping in the database (e.g., a collection `users` where each user document lists their pond IDs or farm ID). This way, when a user logs in, the app knows which topics/ponds to display.

Roles: If a role-based access is needed (not priority for MVP unless specifically required), we could define roles like "farmer" vs "admin". An admin might see all ponds in the system or have permission to add new users/ponds. Given the 2-day MVP timeline, we will likely implement just one role (all users equal) but design the DB schema in a way that can accommodate a role field later. For now, ensuring each user only sees their own data is the main goal – which is handled by scoping the data queries to their pond IDs.

Security: Communication security is also a consideration. MQTT will be configured with TLS if using a cloud broker that supports it, or at least with authentication. The web app will be served over HTTPS (especially if we use Firebase Hosting, it's HTTPS by default). We won't expose the MQTT broker directly to the web client (unless we use it via WebSockets for live updates – but even then, we'd secure it). We'll also validate data on the backend to prevent any injection or malformed data from crashing our system (though with our own simulated devices, this risk is low in MVP).

Adding Ponds/Users: In MVP, we will pre-configure the ponds and user accounts (to save time on building full UI for creation). For example, we'll have 1 farm account with 3 ponds simulated. But we note that the architecture is scalable – new ESP32 units can be added with new topics, and by creating a user entry for a farmer with that pond, the system would automatically include them. In a more polished version, an admin UI could allow creating new pond entries (the NusaPond app had a feature to add new pond info in their app [17]). For now, we assume static configuration to focus on monitoring features.

Task allocation: Likely the backend dev will set up Firebase Auth or a simple user check in the backend. The frontend dev will integrate the login page (from Firebase UI or a basic form). This can be done in parallel with other tasks early on, since it's mostly boilerplate.

MVP Development Work Plan (48-Hour Timeline)

To deliver a functional MVP in 2 days with 4 team members, we will **parallelize tasks** and focus on essentials:

Day 1: Architecture Setup & Core Functionality
- *Morning:* - **Device Simulation Setup:** (Dev A) Flash ESP32s with MicroPython/Arduino code to simulate sensor readings. Test publishing to a test MQTT topic.
- **Cloud Backend & MQTT Broker:** (Dev B) Set up MQTT broker (e.g. create HiveMQ Cloud instance or configure Mosquitto on a server). Establish the backend subscriber (a Python script) and log sample data. Also set up the database (e.g. initialize Firebase project and Firestore rules).
- **Frontend Scaffold:** (Dev C & Dev D) Initialize the web app project (using React/Vue or plain HTML). Create

main pages: Login, Dashboard, and placeholder for charts. Implement language toggle with a few sample strings. Ensure the page is responsive on mobile.

- *Afternoon:*
- **Data Pipeline Test:** (Dev A & B) Connect the ESP32 publishers to the backend subscriber. Verify that real sensor data goes into the DB. Adjust any JSON format issues. Simulate a variety of data (including some anomalies) to have test entries.
- **ML Model Prep:** (Dev B or separate ML specialist Dev) Generate synthetic data for normal vs abnormal conditions. Train a Random Forest classifier on it (this can be done quickly in a Jupyter notebook). Save the model. Begin integrating this model into the backend code (e.g. load the model and test it on recent data from DB).
- **Frontend Live Data Integration:** (Dev C) Set up a connection from the frontend to the data source. If using Firebase, use Firestore's JS SDK to listen for new data for the user's ponds. If using a custom backend API, set up AJAX calls or WebSocket. For now, perhaps just pull the latest reading per pond for display.
- **UI Design Refinement:** (Dev D) Polish the dashboard layout – maybe design a card for each pond showing key values and an icon for status (normal/problem). Add a simple chart component (with dummy data for now). Ensure bilingual text fits nicely.
- *Evening:*
- Team sync-up: test the **entire flow manually** – simulate sensor on ESP32 -> see data in DB -> see data on UI. This will expose integration issues to fix.
- Identify any blockers for offline or notifications early. For instance, test if service worker can cache a page. Dev D can begin adding a service worker using available PWA libraries or manual coding (if using React, turn on service worker in Create React App template).

Day 2: High-Impact Features & Testing

- *Morning:* - **Anomaly Detection & Alerts:** (Dev B) Finalize integrating the Random Forest model in the backend. Implement the logic to flag anomalies and log/send alerts. For testing, intentionally inject an out-of-range reading and see if backend prints an "Alert" log. Connect the Twilio API (get API keys ready and use their Python/Node helper library to send an SMS). Test sending an SMS to team's phone. Also set up Firebase Cloud Messaging server key if using push.
- **Push Notifications:** (Dev C) Implement service worker push subscription in the app. This might involve generating VAPID keys or using Firebase Messaging SDK. Test receiving a push manually (FCM has a console to send test messages).
- **Chart and History View:** (Dev D) Replace dummy chart data with real historical data from DB. For example, query the last 24h of readings for a selected pond and plot temperature vs time. Ensure the chart is label-localized (e.g. "Température" in French).
- **Finalize Offline Capabilities:** (Dev C) Make sure the service worker caches important files (app shell and perhaps an offline page). Implement logic to store last known sensor values in localStorage each time data updates, so that it can be loaded on startup if offline. Test by disconnecting internet and loading the app – it should show previously fetched data.

- *Afternoon:*

- **End-to-End Testing:** All devs participate in testing the MVP end-to-end. Each person takes a user role and scenario:
    - Simulate normal operation (all readings normal) – check UI and no alerts.
    - Simulate an anomaly (e.g., lower the DO value on ESP32 script) – ensure backend logs it, SMS is received, and UI alert appears (or push notification if subscribed).
    - Test the app in French vs English – confirm all text changes and layout holds up.
    - Test offline mode – load data, turn off Wi-Fi, then navigate the app.
    - Multi-user test: create two dummy user accounts tied to different ponds (if possible) and ensure each only sees their ponds.
- **Performance tweaks:** If any part is laggy (maybe chart rendering or too frequent MQTT messages), adjust parameters (e.g., reduce publish rate if needed, or throttle UI updates).

- **UI Cleanup:** Fix any UI bugs discovered (overlapping text, etc.). Given farmers' usage, we ensure it's not only functional but **intuitive and clear** (perhaps add tooltips or help text if necessary for any complex data).

- *Evening (Wrap-up):*

- Prepare a short demo or screenshots for presentation. Ensure all team members know how to run each component (in case of live demo, one might show the device, another the app, etc.).
- List out possible next steps (for discussion, though not implemented) such as integrating actuators for automated control, scaling to more ponds, adding Arabic language, etc., to show foresight beyond MVP.

Throughout development, we keep the system simple and **prioritize features that deliver high value quickly**: real-time monitoring, anomaly alerts, and an accessible UI. Less critical features (like advanced role management or super-polished graphics) are left out or mocked, given the time constraint.

Conclusion

Our proposed MVP will demonstrate a **smart fish farming monitor** that digitizes pond management for Algerian farmers. Leveraging ESP32 IoT sensors, MQTT communication, cloud data processing, and a PWA dashboard, the system provides farmers with **live insights into water quality and instant alerts** for any dangerous conditions. Crucially, it is built to work under rural connectivity limitations – caching data offline and using SMS as a backup for alerts [5] [3]. The use of a Random Forest ML model adds an intelligent layer to detect subtle anomalies beyond simple thresholds [11]. With a bilingual interface and simple design, even non-technical users can adopt the tool easily.

In just two days, our team will focus on implementing the core end-to-end functionality: from **simulated sensor data to actionable alert**. This PoC will validate the concept and lay the groundwork for a more robust product (with additional languages, automated controls, and scalability) in the future. By bringing IoT and AI to fish farming, we aim to **increase yields and reduce losses** – helping farmers respond quickly to water quality issues and modernize their operations [1] [18]. This smart pond monitoring MVP not only pushes the boundaries of decentralized aquaculture tech, but also addresses local needs with an offline-first, user-friendly approach, positioning Algeria's fish farmers for a more sustainable and connected future.

Sources:

- Waddah A. Al-Mutairi et al., *"IoT-based smart monitoring and management system for fish farming,"* BEEI, 2023 – highlighting need to move from traditional to tech-based fish farm management [1].
- Ferdinan Darmalim et al., *"Smart Pond IoT solution for continuous real-time monitoring,"* IOP Conf. Ser.: EES 794, 2021 – describing an IoT pond system with sensors (temp, pH, DO, TDS, salinity) and a farmer-friendly mobile app [19] 【38†L214–L222】.
- *Instructables IoT Agriculture project* – demonstrates ESP32 publishing sensor data via MQTT in real time [6].
- EcoGuard Aquaculture System – utilizes **MQTT for low-latency data** and employs Random Forest models for water quality analytics [20] [4].
- NusaPond App (Indonesia) – example where ponds are flagged as *normal* or *"problematic"* based on sensor ranges, with automatic alerts for abnormal conditions [12].
- HiveMQ Blog – notes that edge MQTT clients can **buffer data offline** and sync when connectivity returns [7].
- Medium (Samyukta Mishra) – on PWAs in agriculture, noting **offline access, fast loading on slow connections, and push notifications** benefits [3].
- Fruto design case study (CABI) – emphasizes designing for developing regions: mobile-first, handle **slow networks and multi-language support** from the start [2].
- Sohail Karim et al., *"IoT Based Smart Fish Farming Aquaculture Monitoring System,"* IJET 2021 – describes using a GSM module to send SMS alerts when pond is in critical state, useful when no internet [5].

---

1 4 11 20 (PDF) IoT-based smart monitoring and management system for fish farming
https://www.researchgate.net/publication/367207333_IoT-based_smart_monitoring_and_management_system_for_fish_farming

2 14 Fruto – UX & UI case study – Multilingual UI design for international agriculture non-profit
https://fruto.design/case-studies/multilingual-ui-design-for-international-agriculture-non-profit

3 The Future of PWAs in the Agriculture Technology Industry | by Samyukta Mishra | Medium
https://medium.com/@samyukya_csm/the-future-of-pwas-in-the-agriculture-technology-industry-81dc845c3170

5 Microsoft Word - 8 IoT Based Smart Fish Farming Aquaculture Monitoring System Sohail Karim 3461
https://www.researchtrend.net/ijet/pdf/8%20IoT%20Based%20Smart%20Fish%20Farming%20Aquaculture%20Monitoring%20System%20Sohail%20Karim%203461.pdf

6 IoT Based Smart Gardening and Smart Agriculture Using ESP32 : 7 Steps - Instructables
https://www.instructables.com/IoT-Based-Smart-Gardening-and-Smart-Agriculture-Us/

7 Leveraging Open Standards like MQTT to Manage Data at the Industrial Edge
https://www.hivemq.com/blog/leveraging-mqtt-manage-data-industrial-edge/

8 18 19 (PDF) An integrated system of mobile application and IoT solution for pond monitoring
https://www.researchgate.net/publication/353649196_An_integrated_system_of_mobile_application_and_IoT_solution_for_pond_monitoring

9 pH Of Pond Water: Why Do Pond Experts Keep It Between 6.5-8.5?
https://premierpond.com/ph-level-guide/

10 [PDF] Dissolved Oxygen for Fish Production - Freshwater Aquaculture
https://freshwater-aquaculture.extension.org/wp-content/uploads/2019/08/Dissolved_Oxygen_for_Fish_Production.pdf

12 13 17 Login page (left) and the main dashboard of Nusapond (right) Since the... | Download Scientific Diagram
https://www.researchgate.net/figure/Login-page-left-and-the-main-dashboard-of-Nusapond-right-Since-the-proposed-Pond-IoT_fig3_353649196

15 Looking for SMS Gateway Providers for Mobilis, Djezzy, and ... - Reddit
https://www.reddit.com/r/algeria/comments/1ji0wzt/looking_for_sms_gateway_providers_for_mobilis/

16 SMS Pricing in Algeria for Text Messaging | Twilio
https://www.twilio.com/en-us/sms/pricing/dz