

MASTER 2 – IASD/GL
HAI912I DÉVELOPPEMENT MOBILE

PlantNanny

Système intelligent d’arrosage basé sur ESP-32



Auteurs :

HUREL Jérémy – N° Étudiant
MACKOW Anaïs – N°22016204

Janvier 2026

Table des matières

1	Introduction	3
2	Vue d'ensemble du projet	3
2.1	Objectifs fonctionnels	3
2.2	Fonctionnalités principales	4
2.2.1	Contrôleur (ESP32)	4
2.2.2	Backend	4
2.3	Architecture globale du projet	4
3	Architecture matérielle	5
3.1	Composants utilisés	5
3.2	Schéma de montage et configuration des pins	5
3.2.1	Montage réel du prototype	6
3.3	Capteurs	6
3.3.1	Calcul de la luminosité (LDR)	7
3.3.2	Calcul de l'humidité du sol (capteur capacitif)	7
3.4	Commande de la pompe à eau	7
3.5	LED rouge	8
3.6	Gestion du contrôleur	8
4	Interface utilisateur : écran TFT	8
4.1	Framework UI personnalisé	8
4.2	Utilisation des boutons	9
5	Architecture logicielle embarquée	9
5.1	Architecture orientée services	9
5.2	Programmation modulaire et bonnes pratiques	10
5.2.1	Principes SOLID	10
5.2.2	Patterns de conception	10
5.3	Communication Bluetooth Low Energy	11
5.4	Cycle d'exécution du firmware	12
5.5	Mise à jour OTA	12
5.6	Gestion des erreurs	12
6	Communication et protocoles	13
6.1	Architecture de communication	13
6.2	Protocole MQTT	13
6.2.1	Avantages de l'architecture MQTT	13
6.2.2	Topics MQTT	13
6.2.3	Commande de forçage de mise à jour	13
6.2.4	Bibliothèque utilisée	14
6.3	API REST	14
6.4	Sécurité du système embarqué	15
6.4.1	Sécurité MQTT	15
6.4.2	Sécurité Bluetooth	15
6.4.3	Stockage sécurisé des identifiants	15
6.4.4	Persistance de la configuration	16
7	Tests et validation	16
7.1	Stratégie de test	16
7.2	Framework de test	16
7.3	Exécution des tests	17
7.4	Résultats et limites	17
8	Outils de développement et infrastructure	17
8.1	Scripts de développement	17
8.2	Infrastructure Docker	18

1 Introduction

L'arrosage des plantes nécessite une attention régulière et une adaptation aux besoins réels du sol. Un excès d'eau peut nuire au développement des racines, tandis qu'un manque d'arrosage peut entraîner un stress hydrique et la mort de la plante. Selon des études agronomiques, le maintien d'un niveau d'humidité optimal du sol peut améliorer la croissance des plantes de 20 à 30%.

À l'origine de ce projet, l'objectif principal était de réaliser un **montage électronique autour d'un microcontrôleur ESP32**, intégrant différents capteurs environnementaux afin de collecter des données physiques réelles. Souhaitant dépasser le cadre académique et nous investir pleinement dans le projet, nous avons choisi de **joindre l'utile à l'agréable** en concevant un dispositif concret, pouvant être utilisé dans un contexte domestique réel.

Dans ce contexte, ce rapport présente la conception et la réalisation d'un système d'arrosage **semi-automatisé pour plantes**, baptisé *PlantNanny*. Ce dispositif vise à adapter l'arrosage aux besoins réels de la plante tout en limitant les interventions humaines, grâce à l'exploitation de données environnementales collectées en temps réel. Le système repose sur un microcontrôleur **ESP32 (LilyGo T-Display)** et intègre plusieurs capteurs permettant de mesurer l'état du sol et les conditions ambiantes (la température, l'humidité du sol et la luminosité).

À partir de ces mesures, le microcontrôleur publie les données sur un **broker MQTT** (Mosquitto). Un service Python reçoit ces informations en temps réel, les persiste dans une base de données et les retransmet à l'application mobile Flutter. L'utilisateur peut ainsi être alerté en cas d'anomalie, consulter l'état des conditions environnementales et piloter l'arrosage à distance.

Dans la suite de ce rapport, nous détaillerons la mise en œuvre ainsi que l'orchestration de l'ensemble du système.

2 Vue d'ensemble du projet

2.1 Objectifs fonctionnels

Le système développé vise à fournir une solution complète de surveillance et de gestion de l'arrosage des plantes. À ce titre, il doit être capable de répondre aux objectifs fonctionnels suivants :

1. **Surveillance continue de l'environnement de la plante** : le dispositif doit mesurer en temps réel la température ambiante, l'humidité du sol ainsi que la luminosité.
2. **Action physique sur le système d'irrigation** : le système doit être en mesure de déclencher un arrosage en activant une pompe à eau. En raison d'un dysfonctionnement matériel de la pompe et pour respecter la consigne, cette action est simulée par l'allumage d'une LED pendant toute la durée théorique de fonctionnement de la pompe.
3. **Configuration initiale simplifiée via Bluetooth** : afin de faciliter la mise en service du dispositif, celui-ci doit proposer un mode de configuration initiale reposant sur une communication Bluetooth, permettant notamment la configuration des paramètres réseau.
4. **Communication avec un serveur distant** : le système doit être capable d'échanger des données avec un serveur distant afin d'assurer la persistance des mesures collectées, la consultation de l'historique des données et le contrôle à distance du dispositif.
5. **Fiabilité et gestion des erreurs** : le fonctionnement global du système doit être robuste et sécurisé, avec une gestion appropriée des erreurs telles que les pertes de communication, les valeurs aberrantes des capteurs ou les défaillances matérielles, afin de garantir un fonctionnement stable dans le temps.

2.2 Fonctionnalités principales

Les fonctionnalités principales de *PlantNanny* peuvent être décrites en distinguant les responsabilités du **contrôleur embarqué** et celles du **backend** (communication et persistance). Cette séparation permet de garder un microcontrôleur simple, réactif et économique en ressources, tout en déportant les traitements liés au stockage, à la mise en historique et à la supervision vers un environnement serveur.

2.2.1 Contrôleur (ESP32)

Le microcontrôleur ESP32 constitue le cœur du système. Il orchestre d'abord l'acquisition des capteurs (température via thermistance NTC, humidité du sol via sonde et luminosité via LDR), puis applique les conversions nécessaires pour passer de valeurs brutes ADC à des grandeurs interprétables ($^{\circ}\text{C}$ et pourcentages).

Pour la mise en service, le contrôleur propose un mode d'appairage **Bluetooth Low Energy (BLE)** où l'utilisateur peut transmettre les identifiants Wi-Fi ainsi que les paramètres de connexion au broker MQTT sans avoir à reprogrammer le firmware.

Une fois configuré, le contrôleur communique en **Wi-Fi** en s'appuyant sur **MQTT** en publant les mesures, l'état du système et les logs sur des topics dédiés (chaînes de caractères qui servent de canal de communication), et il peut également s'abonner à des topics de commande (par exemple pour forcer un rafraîchissement des capteurs ou déclencher un arrosage). Enfin, afin de simplifier la maintenance, le système supporte la mise à jour à distance grâce au mécanisme *Over-The-Air* (OTA), permettant d'installer une nouvelle version sans accès physique à l'appareil.

2.2.2 Backend

Le backend s'appuie sur un **broker MQTT** (Mosquitto) qui joue le rôle de seul point de rendez-vous entre les composants. Ainsi, le contrôleur y publie ses messages, et les services s'y abonnent. Cette approche publish/subscribe apporte un fort découplage où l'ESP32 n'a pas besoin de connaître l'adresse d'un serveur applicatif, et l'ajout d'un nouveau utilisateur (par exemple un outil de monitoring) ne nécessite pas de modifier le firmware.

Un **service Python** s'abonne aux topics de mesures et d'état, agrège les informations reçues et assure la **persistance** dans une base **PostgreSQL**. Cette historisation rend possible la consultation des statistiques. Le même service peut aussi publier des événements ou des notifications (alertes) sur des topics spécifiques, afin que l'application Flutter affiche des avertissements lorsque des seuils sont dépassés.

Enfin, le backend permet le **pilotage à distance** en publant des commandes MQTT vers le contrôleur. Dans ce modèle, l'application Flutter peut soit publier directement sur le broker (si le choix d'architecture et les contraintes de sécurité le permettent), soit passer par le service Python qui agit alors comme médiateur.

2.3 Architecture globale du projet

Un microcontrôleur ESP32 (LilyGo T-Display) est chargé de centraliser les mesures des capteurs et de piloter les actionneurs. Les capteurs sont connectés aux entrées analogiques du microcontrôleur, tandis que la pompe à eau (branchée sur 4 piles AAA) est commandée via un relais assurant une isolation électrique.

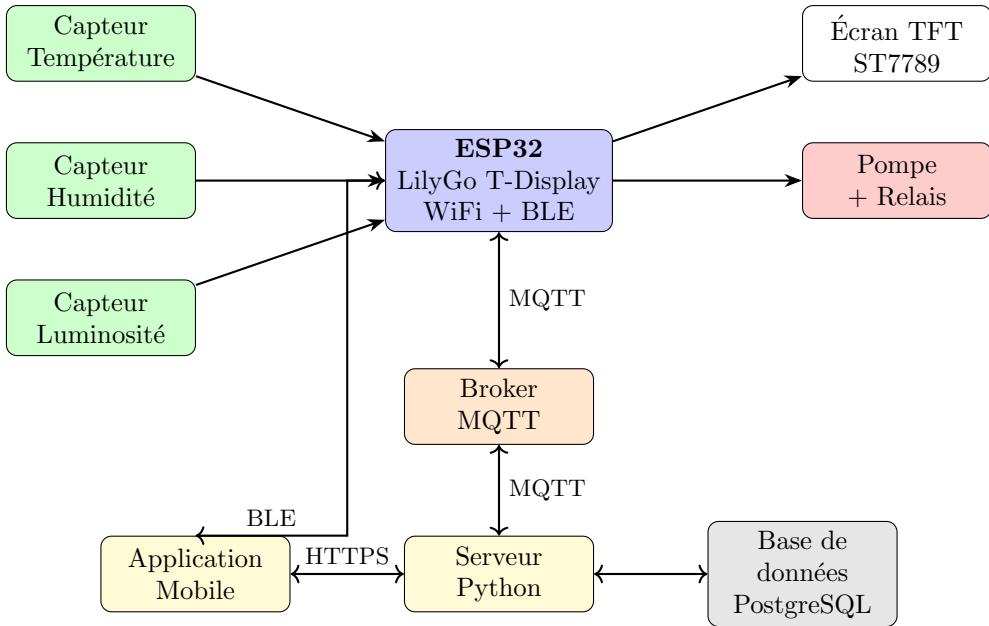


FIGURE 1 – Architecture globale du système PlantNanny

3 Architecture matérielle

3.1 Composants utilisés

Le tableau 1 liste les composants principaux du système.

TABLE 1 – Liste des composants du système

Composant	Description	Référence
Microcontrôleur	ESP32 avec WiFi, Bluetooth et écran TFT intégré	LilyGo T-Display
Écran	Écran TFT 1.14" 135×240 pixels, driver ST7789	Intégré
Capteur humidité	Capteur d'humidité de l'air	DHT11/DHT22
Capteur température	Capteur de température intégré ou externe	Thermistance NTC
Capteur luminosité	Photorésistance (LDR) pour mesure luminosité	GL5528
Relais	Module relais 5V pour commande pompe	SRD-05VDC
Pompe	Pompe submersible 5V, 120 L/h	–

3.2 Schéma de montage et configuration des pins

La figure 2 présente le schéma de montage simplifié du système *PlantNanny* ainsi que l'affectation des différentes broches du microcontrôleur ESP32 LilyGo T-Display. Les capteurs environnementaux (température, humidité du sol et luminosité) sont raccordés aux entrées analogiques du microcontrôleur, tandis que l'actionneur principal (pompe à eau) est commandé via une sortie numérique.

En raison d'un dysfonctionnement matériel de la pompe et afin de respecter les consignes du projet, l'activation de celle-ci est simulée par une **LED rouge** connectée à la broche **GPIO13**. Celle-ci s'allume pendant toute la durée théorique de fonctionnement de la pompe. Cette approche permet de valider la logique logicielle et la chaîne de commande sans compromettre le reste du système.

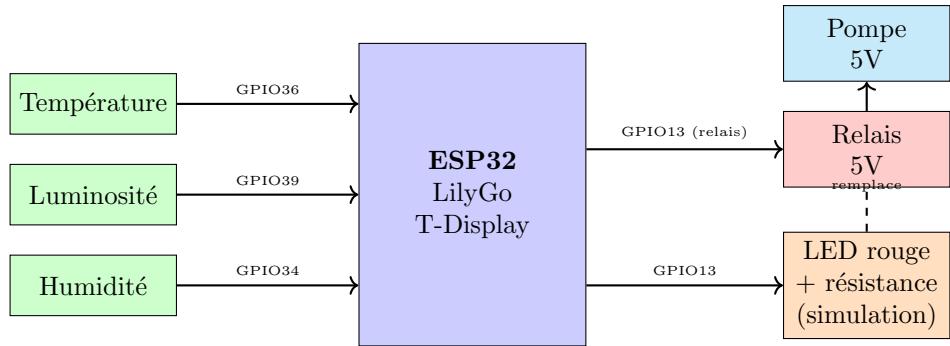


FIGURE 2 – Schéma de montage simplifié du système (la LED simule l’activation de la pompe)

Afin de garantir un captage fiable des données et de limiter la consommation énergétique, l’alimentation des capteurs est pilotée par la broche **GPIO26**. Cette broche est positionnée à l’état haut uniquement lors des phases de mesure, permettant ainsi de forcer l’acquisition des données avant lecture par l’ADC, puis de couper l’alimentation des capteurs entre deux cycles d’acquisition.

3.2.1 Montage réel du prototype

Les figures 4a à 4b présentent le montage réel du prototype.

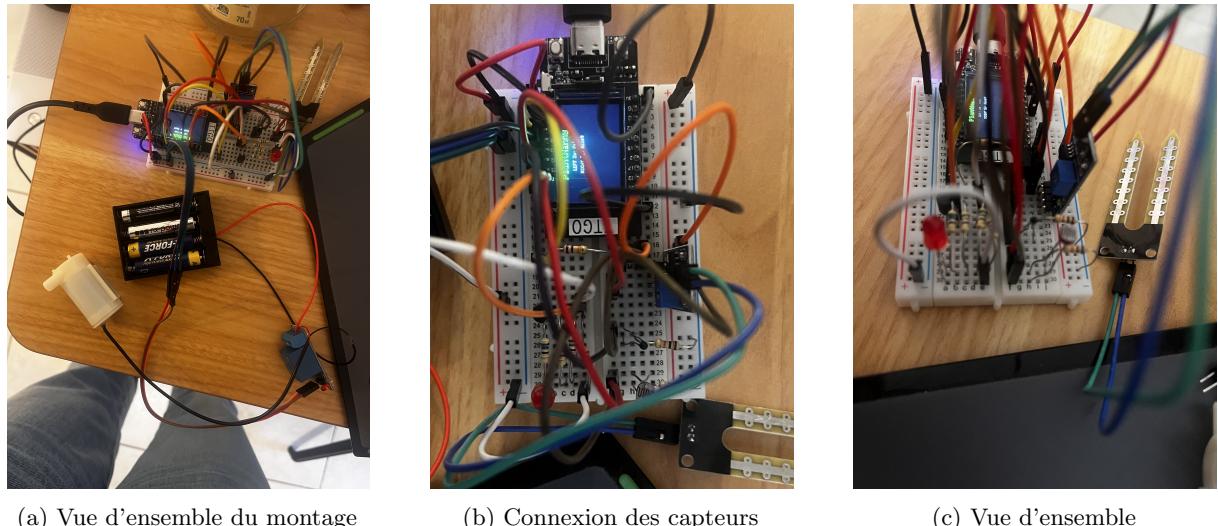


FIGURE 3 – Visuel du montage réel du système PlantNanny

3.3 Capteurs

Les capteurs utilisés présentent une variation de résistance ou de tension en fonction de la grandeur mesurée. Pour rendre ces variations exploitables par l’ESP32, chaque capteur est intégré dans un **pont diviseur de tension** avec une résistance fixe de **10 kΩ**.

Pont diviseur de tension La tension mesurée est donnée par :

$$V_{\text{out}} = V_{\text{alim}} \times \frac{R_{\text{capteur}}}{R_{\text{capteur}} + R_{\text{fixe}}} \quad (1)$$

Avec $R_{\text{fixe}} = 10 \text{ k}\Omega$ et $V_{\text{alim}} = 3.3 \text{ V}$, le courant maximal est :

$$I_{\text{max}} = \frac{V_{\text{alim}}}{R_{\text{capteur}} + R_{\text{fixe}}} \approx \frac{3.3}{20\,000} \approx 0.165 \text{ mA} \quad (2)$$

Ce courant faible limite la consommation énergétique et l’échauffement des composants.

Calcul de la température (thermistance NTC) Pour une thermistance NTC, la relation entre résistance et température suit l'équation de Steinhart-Hart simplifiée (équation bêta) :

$$\frac{1}{T} = \frac{1}{T_0} + \frac{1}{B} \ln \left(\frac{R}{R_0} \right) \quad (3)$$

où :

- T : température en Kelvin ;
- $T_0 = 298.15$ K (25°C) : température de référence ;
- $R_0 = 10 \text{ k}\Omega$: résistance à T_0 ;
- $B \approx 3950$ K : coefficient bêta de la thermistance.

Conversion ADC vers température La conversion des données brutes de l'ADC vers la température s'effectue en plusieurs étapes dans le code :

1. **Lecture ADC moyennée** : 10 échantillons sont prélevés pour réduire le bruit :

$$ADC_{\text{moy}} = \frac{1}{10} \sum_{i=1}^{10} ADC_i \quad (4)$$

2. **Calcul de la résistance** de la thermistance à partir du pont diviseur :

$$R_{\text{therm}} = R_{\text{série}} \times \frac{ADC_{\text{max}} - ADC_{\text{moy}}}{ADC_{\text{moy}}} \quad (5)$$

avec $ADC_{\text{max}} = 4095$ (12 bits) et $R_{\text{série}} = 5.6 \text{ k}\Omega$.

3. **Application de l'équation bêta** :

$$T(C) = \left[\frac{1}{T_0 + 273.15} + \frac{1}{B} \ln \left(\frac{R_{\text{therm}}}{R_0} \right) \right]^{-1} - 273.15 \quad (6)$$

3.3.1 Calcul de la luminosité (LDR)

La photorésistance (LDR) présente une résistance qui diminue avec l'intensité lumineuse. La conversion en pourcentage de luminosité (0–100%) s'effectue par mapping linéaire de la valeur ADC :

$$L(\%) = \begin{cases} 100 \times \left(1 - \frac{ADC}{ADC_{\text{max}}} \right) & \text{si lecture inversée (LDR vers GND)} \\ 100 \times \frac{ADC}{ADC_{\text{max}}} & \text{sinon (LDR vers VCC)} \end{cases} \quad (7)$$

3.3.2 Calcul de l'humidité du sol (capteur capacitif)

Le capteur d'humidité capacitif mesure l'humidité du sol par variation de capacité. La conversion utilise des valeurs de calibration (ADC_{sec} et ADC_{humide}) :

$$H(\%) = \begin{cases} 100 \times \frac{ADC_{\text{sec}} - ADC}{ADC_{\text{sec}} - ADC_{\text{humide}}} & \text{si inversé (capteur capacitif)} \\ 100 \times \frac{ADC - ADC_{\text{sec}}}{ADC_{\text{humide}} - ADC_{\text{sec}}} & \text{sinon} \end{cases} \quad (8)$$

Les valeurs par défaut sont : $ADC_{\text{sec}} = 4095$ (sol sec) et $ADC_{\text{humide}} = 1500$ (sol mouillé).

3.4 Commande de la pompe à eau

La pompe à eau constitue l'actionneur principal du système. Elle est commandée via un relais piloté par la broche **GPIO13** de l'ESP32.

Le câblage est réalisé comme suit :

- La broche **IN** du relais est connectée à GPIO13
- La broche **VCC** du relais est connectée à 5V
- La broche **GND** du relais est connectée à la masse
- Le fil positif de l'alimentation de la pompe est connecté à la borne **COM**
- La borne **NO** (Normally Open) est reliée au fil positif de la pompe
- Le fil négatif de la pompe est relié directement à la masse de l'alimentation.

Lorsque GPIO13 est à l'état haut, le contact **COM–NO** se ferme et alimente la pompe. À l'état bas, le circuit est ouvert et la pompe est arrêtée.

3.5 LED rouge

Cette LED représente l'état de fonctionnement de la pompe et permet de valider la logique de commande sans actionner d'élément mécanique.

La LED est alimentée en **5 V** et commandée par la broche **GPIO13** de l'ESP32 au moyen d'un **transistor de commutation**. Ce dernier joue le rôle d'interface entre le microcontrôleur (3,3 V) et la charge alimentée en 5 V, assurant l'adaptation de niveau logique et la protection de la broche GPIO.

Une limitation de courant est nécessaire pour protéger la LED. Dans notre montage, la résistance série n'est pas assurée par un seul composant, mais par **quatre résistances de 560 Ω placées en série avant la LED**. La résistance est donc :

$$R_{\text{eq}} = 4 \times 560 = 2240 \Omega \quad (9)$$

En prenant une tension directe typique pour une LED rouge de $V_{\text{LED}} \approx 2.0 \text{ V}$, le courant traversant la LED est approximativement :

$$I_{\text{LED}} = \frac{V_{\text{alim}} - V_{\text{LED}}}{R_{\text{eq}}} \approx \frac{5 - 2.0}{2240} \approx 1.34 \text{ mA} \quad (10)$$

La LED s'allume pendant toute la durée d'activation du signal de commande qui serait normalement envoyé au relais de la pompe. Elle représente indicateur fiable du déclenchement de l'arrosage et permet de valider le bon fonctionnement de la chaîne de commande matérielle et logicielle.

3.6 Gestion du contrôleur

Le contrôleur est organisé autour de **modes de fonctionnement** clairement séparés, afin de simplifier la compréhension du comportement global et de faciliter la maintenance.

En **mode normal**, l'appareil effectue des mesures périodiques, met à jour l'affichage, publie les données via MQTT et lance l'arrosage si action de l'utilisateur côté application.

En **mode configuration**, l'appairage Bluetooth (BLE) permet de transmettre les identifiants Wi-Fi et les paramètres MQTT depuis l'application mobile, ce qui évite toute reprogrammation.

Enfin, un **mode de réinitialisation** permet de remettre l'appareil à l'état initial (effacement des paramètres enregistrés) en cas de besoin (changement de réseau, dysfonctionnement, etc.).

4 Interface utilisateur : écran TFT

4.1 Framework UI personnalisé

Pour l'affichage, nous avons utiliser un framework UI personnalisé basé sur la bibliothèque TFT_eSPI, offrant une architecture à base de composants et d'un système automatique pour le postionnement des éléments.



(a) Écran du TT-Go



(b) Affichage du code PIN

FIGURE 4 – Interface graphique

4.2 Utilisation des boutons

La carte dispose de deux boutons intégrés :

- **Bouton Boot (GPIO0)** : Navigation entre les écrans, appui long pour mode BLE ;
- **Bouton Reset** : Redémarrage du système.

Ainsi, il suffit juste de rester appuyer 3sec sur un des deux boutons pour faire l'action désirée.

5 Architecture logicielle embarquée

5.1 Architecture orientée services

Pour le système, nous avons opté pour une architecture orientée services (SOA) avec une stricte séparation des responsabilités. Cette architecture permet une grande modularité et facilite les tests unitaires.

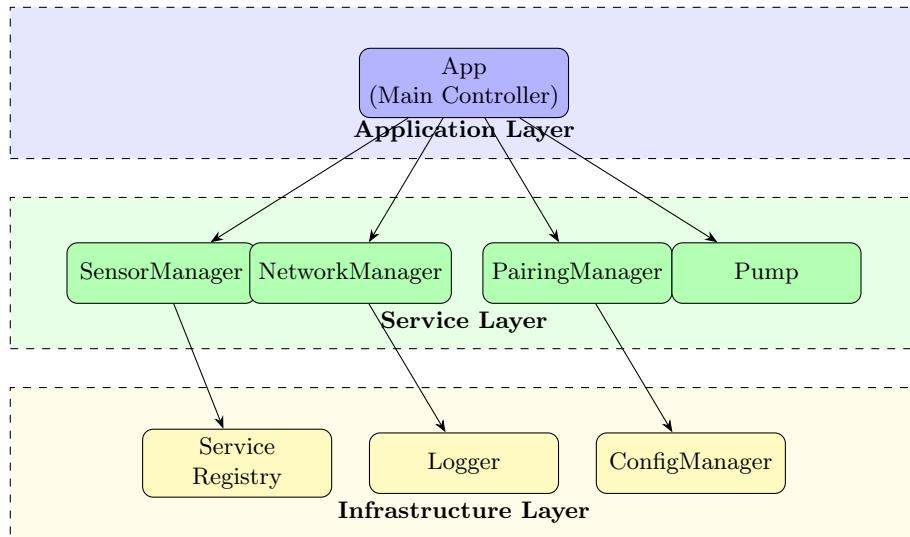


FIGURE 5 – Architecture en couches du firmware

Le point d'entrée de l'application est situé dans le fichier `main.cpp`, qui assure l'initialisation globale du système et le lancement de l'application principale. La logique métier et les fonctionnalités sont ensuite réparties dans des modules spécifiques, organisés sous forme de bibliothèques et de services.

```

src/
main.cpp                      # Point d'entrée Arduino
libs/
common/                         # Bibliothèques réutilisables
logger/                          # Système de logging (MQTT + Serial)
patterns/                        # Design patterns (Result, Singleton)
service/                         # Framework de services (Registry, DI)
ui/                             # Framework UI (composants, builders)
utils/                           # Utilitaires (Event, EspError)
plant_nanny/
    App.cpp                      # Application principale
    states/                       # Gestion des modes (Normal, Pairing, Reset)
    services/                     # Services applicatifs
        bluetooth/               # Gestion BLE (appairage)
        captors/                  # Services capteurs
            humidity/             # Capteur humidité du sol
            luminosity/            # Capteur luminosité (LDR)
            temperature/            # Capteur température (NTC)
    network/                      # Gestion WiFi et HTTP
    mqtt/                         # Service MQTT

```

```

ota/          # Mise à jour Over-The-Air
pump_manager/ # Contrôle de la pompe
config/       # Gestion de la configuration

```

La bibliothèque `common` regroupe l'ensemble des composants génériques et réutilisables, indépendants du projet *PlantNanny*. Elle inclut notamment un système de journalisation centralisé, capable de diffuser les logs à la fois sur la liaison série et via MQTT, facilitant ainsi le débogage local et distant. Des patrons de conception tels que *Singleton* et *Result* sont également utilisés afin d'encadrer la gestion des ressources et des erreurs. Un framework interne de services permet quant à lui l'enregistrement et l'injection des dépendances, favorisant un couplage faible entre les différents modules.

Le dossier `plant_nanny` contient l'ensemble du code spécifique à l'application. Le fichier `App.cpp` orchestre le cycle de vie de l'application et coordonne l'interaction entre les différents services. La gestion des états du système est isolée dans le sous-dossier `states`, permettant de distinguer clairement les modes de fonctionnement tels que le mode normal, le mode d'appairage Bluetooth ou encore le mode de réinitialisation.

Les fonctionnalités principales sont implémentées sous forme de services autonomes, regroupés dans le dossier `services`. Chaque service encapsule une responsabilité bien définie, comme la *communication Bluetooth* pour la configuration initiale, la *gestion des capteurs environnementaux*, la *connectivité réseau*, la *communication MQTT*, la *mise à jour* du système à distance. Cette organisation modulaire permet de faire évoluer ou remplacer un service sans impacter le reste de l'application.

Dans l'ensemble, cette architecture orientée services permet de structurer le code de manière claire et cohérente, tout en répondant aux contraintes spécifiques des systèmes embarqués. Elle facilite également la maintenance du projet et son extension vers de nouvelles fonctionnalités ou de nouveaux dispositifs.

5.2 Programmation modulaire et bonnes pratiques

Le projet est conçu selon des principes de programmation modulaire, favorisant la séparation des responsabilités, la réutilisabilité des composants et la maintenabilité du code.

5.2.1 Principes SOLID

Principe	Application dans PlantNanny
Single Responsibility	Chaque service a une responsabilité unique (ex : <code>Temperature</code> ne fait que lire la température)
Open/Closed	Extension via interfaces (ex : <code>INetworkService</code>) sans modification du code existant
Liskov Substitution	Les mocks de test respectent les contrats des interfaces
Interface Segregation	Interfaces spécifiques et minimales pour chaque service
Dependency Inversion	Injection via le Service Registry, dépendances vers abstractions

TABLE 2 – Application des principes SOLID

5.2.2 Patterns de conception

Le firmware s'appuie sur plusieurs patrons de conception afin de structurer le code, d'améliorer sa lisibilité et de faciliter sa maintenance.

Pattern	Utilisation
Service Registry	Conteneur d'injection de dépendances centralisé
Singleton	Services globaux (Logger, Registry)
Builder	API fluide pour construction des composants UI
Result Type	Gestion fonctionnelle des erreurs (inspiré de Rust)
Observer	Système d'événements pour communication inter-services

TABLE 3 – Patterns de conception utilisés

Service Registry Le Service Registry est le cœur de l'architecture, permettant :

- L'enregistrement des services au démarrage
- L'accès aux services via un Accessor typé
- La gestion de contextes multiples (utile pour les tests)

Service	Responsabilité
Logger	Journalisation (Serial + MQTT distant)
NetworkManager	Connexion WiFi, requêtes HTTP, maintien connexion
BluetoothManager	Gestion BLE, appairage, réception configuration
Humidity	Lecture capteur humidité
Temperature	Lecture capteur température
Luminosity	Lecture capteur luminosité
Pump	Contrôle de la pompe (GPIO)
UpdateOrchestrator	Gestion des mises à jour OTA

TABLE 4 – Services du firmware

Services principaux

5.3 Communication Bluetooth Low Energy

Le service Bluetooth implémente un serveur BLE avec les caractéristiques suivantes :

UUID	Description
4fafc201-...	Service principal PlantNanny
beb5483e-...	Caractéristique RX (Write) - réception données
6e400003-...	Caractéristique TX (Notify) - envoi données

TABLE 5 – Services et caractéristiques BLE

Le Bluetooth est utilisé pour la configuration initiale des identifiants Wi-Fi.

5.4 Cycle d'exécution du firmware

Le firmware exécute un cycle simple et répétitif : il initialise les services au démarrage, puis alterne entre des phases de mesure et de communication.

Concrètement, le contrôleur sort de veille à intervalle régulier, alimente temporairement les capteurs pour effectuer des lectures stables, convertit les valeurs brutes en grandeurs exploitables, décide d'un éventuel arrosage (ou de sa simulation), met à jour l'affichage et publie les données via MQTT avant de retourner en veille.

Lorsque l'utilisateur demande une configuration initiale, l'appareil active la communication BLE pour recevoir les paramètres réseau. En cas d'anomalie (capteur non initialisé, valeurs ADC hors plage, réseau indisponible), le système privilégie une approche robuste basée sur la détection d'erreurs, des tentatives de reconnexion et, si nécessaire, une réinitialisation contrôlée.

En résumé :

1. Réveil (timer ou événement)
2. Allumage des capteurs (GPIO26 HIGH)
3. Attente de stabilisation (50 ms)
4. Lecture des 3 capteurs (10 échantillons moyennés chacun)
5. Extinction des capteurs (GPIO26 LOW)
6. Publication des données via MQTT
7. Retour en veille ou attente du prochain cycle.

5.5 Mise à jour OTA

Le système supporte les mises à jour Over-The-Air via le service `UpdateOrchestrator` donc sans accès physique à l'appareil. Le processus de mise à jour suit les étapes suivantes :

1. Vérification de la connexion réseau
2. Téléchargement de l'image firmware depuis l'URL fournie
3. Vérification de l'intégrité (checksum)
4. Écriture dans la partition OTA secondaire
5. Redémarrage sur la nouvelle partition

5.6 Gestion des erreurs

Pour ce qui concerne la gestion d'erreurs, nous avons implementé des solutions à divers niveaux.

Gestion des erreurs capteurs Les capteurs retournent `NaN` (Not a Number) en cas d'erreur de lecture :

- Capteur non initialisé
- Valeur ADC hors plage (court-circuit ou circuit ouvert)
- Erreur de communication

Stratégies de récupération en cas d'erreur Pour garantir un fonctionnement stable, plusieurs stratégies de reprise sont mises en œuvre. Les erreurs non critiques (perte Wi-Fi, MQTT indisponible, échec temporaire de téléchargement) déclenchent des tentatives de reconnexion et un logging détaillé. Les erreurs liées aux capteurs (valeurs incohérentes ou hors plage) sont traitées en invalidant la mesure (retour `NaN`) afin d'éviter toute décision d'arrosage basée sur une donnée incorrecte.

Enfin, en cas de problème persistant, l'appareil peut redémarrer proprement pour réinitialiser les composants logiciels et retrouver un état nominal, sans bloquer l'utilisateur.

6 Communication et protocoles

6.1 Architecture de communication

Le système utilise une architecture de communication hybride combinant MQTT et REST API. Cette approche permet de découpler les différents composants et d'éviter une connexion directe au microcontrôleur pour la récupération des données.

La figure 6 illustre cette architecture.

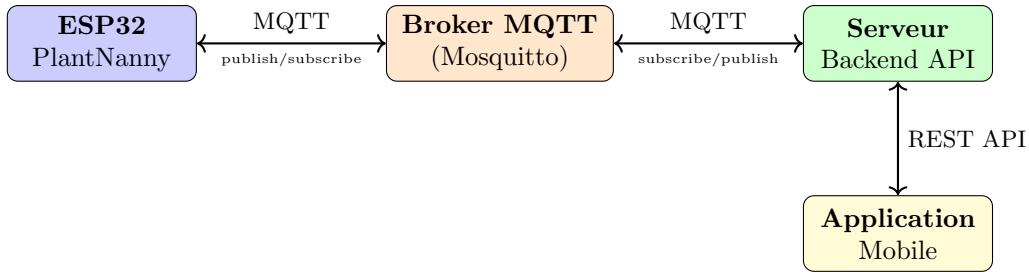


FIGURE 6 – Architecture de communication MQTT

6.2 Protocole MQTT

MQTT (Message Queuing Telemetry Transport) est utilisé comme protocole principal de communication entre l'ESP32 et le backend, via un **broker** intermédiaire.

Dans notre contexte MQTT, un **broker** est un serveur central qui reçoit les messages publiés par les clients (publishers) et les redistribue aux clients abonnés (subscribers) selon les **topics**. Il joue ainsi le rôle de « routeur » applicatif et peut également gérer l'authentification, les ACL et, selon la configuration, la rétention de messages.

6.2.1 Avantages de l'architecture MQTT

Cette architecture présente plusieurs avantages :

- **Découplage** : L'ESP32 n'a pas besoin d'exposer de serveur HTTP, évitant les problèmes de NAT et de pare-feu
- **Légèreté** : MQTT est un protocole léger, adapté aux microcontrôleurs avec ressources limitées
- **Temps réel** : Les données sont transmises dès leur disponibilité via le mécanisme publish/subscribe
- **Fiabilité** : Le broker assure la persistance des messages en cas de déconnexion temporaire
- **Scalabilité** : Plusieurs contrôleurs peuvent se connecter au même broker sans modification du serveur

6.2.2 Topics MQTT

La communication se fait via les topics suivants

Topic	Description
plantnanny/{device_id}/sensors	Publication des données capteurs (ESP32 → Serveur)
plantnanny/{device_id}/status	Publication de l'état du système (ESP32 → Serveur)
plantnanny/{device_id}/logs	Publication des logs pour debugging distant
plantnanny/{device_id}/cmd/refresh	Commande de forçage de mise à jour des capteurs (Serveur → ESP32)

TABLE 6 – Topics MQTT du système

6.2.3 Commande de forçage de mise à jour

La seule commande disponible via MQTT est le **forçage de mise à jour des données capteurs**. Lorsque le serveur (ou l'application via le serveur) souhaite obtenir des données fraîches, il publie un message sur le topic `plantnanny/{device_id}/cmd/refresh`.

L'ESP32, abonné à ce topic, déclenche alors immédiatement une lecture de tous les capteurs et publie les nouvelles valeurs sur le topic `plantnanny/{device_id}/sensors`.

6.2.4 Bibliothèque utilisée

La communication MQTT est gérée par la bibliothèque **PubSubClient** (v2.8), qui offre une API simple pour la connexion au broker et la gestion des messages.

6.3 API REST

En complément de MQTT, le système expose des APIs REST documentées avec OpenAPI 3.0 :

- **API Serveur** : Backend cloud pour persistance et gestion multi-utilisateurs ;
- **API Contrôleur** : Endpoints locaux exposés par l'ESP32 (usage en mode dégradé).

Méthode	Endpoint	Description
POST	/auth/register	Création d'un compte utilisateur
POST	/auth/login	Authentification (retourne JWT)
POST	/auth/logout	Déconnexion
POST	/auth/refresh	Renouvellement du token

TABLE 7 – Endpoints d'authentification

Authentification

Méthode	Endpoint	Description
GET	/plants	Liste des plantes de l'utilisateur
POST	/plants	Ajout d'une nouvelle plante
GET	/plants/{id}	Détails d'une plante
PUT	/plants/{id}	Mise à jour de la configuration
DELETE	/plants/{id}	Suppression d'une plante
POST	/plants/{id}/water	Déclenchement arrosage

TABLE 8 – Endpoints de gestion des capteurs

Gestion des capteurs

Méthode	Endpoint	Description
GET	/measurements	Historique des mesures (filtrable)
GET	/measurements/plants/{id}	Historique d'une plante
POST	/measurements	Upload de mesures (contrôleur)

TABLE 9 – Endpoints de mesures

Mesures et historique Le contrôleur ESP32 expose également une API REST locale :

Méthode	Endpoint	Description
GET	/sensors	Liste des capteurs disponibles
GET	/sensors/data	Lectures actuelles de tous les capteurs
POST	/watering/trigger	Déclenchement manuel de l'arrosage
GET	/watering/status	État du système d'arrosage

TABLE 10 – Endpoints du contrôleur

6.4 Sécurité du système embarqué

L'API implémente plusieurs mécanismes de sécurité :

- **Authentification via Firebase** : gestion sécurisée des comptes utilisateurs et émission de jetons JWT avec expiration
- **HTTPS** : chiffrement des communications entre l'application, l'API et les services externes (en environnement de production)
- **RBAC (Role-Based Access Control)** : contrôle des accès aux ressources en fonction des rôles utilisateurs
- **Rate limiting** : limitation du nombre de requêtes afin de prévenir les abus et les attaques par déni de service

6.4.1 Sécurité MQTT

La sécurité des échanges MQTT a suscité notre attention lors du développement, dans la mesure où les données environnementales ainsi que les commandes d'arrosage transmettent par le réseau. Le broker MQTT a donc été configuré afin de garantir un niveau de protection adapté à un contexte IoT.

L'accès au broker est protégé par un mécanisme d'authentification reposant sur des identifiants utilisateur et des mots de passe. Ces informations sont gérées et stockées de manière centralisée dans **Firebase**, ce qui permet de simplifier la gestion des comptes tout en évitant le stockage de secrets sensibles directement sur les dispositifs embarqués.

En complément de l'authentification, un contrôle d'accès fin est mis en place au moyen de listes de contrôle d'accès (*Access Control Lists*). Ces règles permettent de restreindre les droits de publication et d'abonnement des clients MQTT en fonction des topics, garantissant ainsi qu'un client ne puisse accéder qu'aux flux de données qui lui sont explicitement autorisés.

Enfin, le broker supporte le chiffrement des communications via le protocole TLS. Celui-ci permet de protéger les échanges contre l'interception ou la modification des messages sur le réseau.

6.4.2 Sécurité Bluetooth

La phase de configuration initiale du dispositif repose sur une communication Bluetooth Low Energy (BLE). Afin d'éviter toute configuration non autorisée, un mécanisme de sécurisation spécifique a été mis en place lors du processus d'appairage.

L'appairage BLE s'effectue à l'aide d'un code PIN à six chiffres, généré par le contrôleur et affiché à l'utilisateur. Ce code peut être saisi manuellement dans l'application mobile.

Une fois le code PIN transmis, celui-ci est vérifié directement par le contrôleur. Aucune information sensible n'est échangée tant que cette étape de validation n'a pas été correctement réalisée. Ce n'est qu'après confirmation de l'authenticité de l'utilisateur que les identifiants Wi-Fi peuvent être envoyés au dispositif.

6.4.3 Stockage sécurisé des identifiants

Les identifiants sensibles utilisés par le système, tels que les paramètres de connexion au réseau Wi-Fi et au broker MQTT, doivent être stockés de manière persistante tout en garantissant leur confidentialité. Pour répondre à cette contrainte, le dispositif embarqué s'appuie sur la mémoire *Non-Volatile Storage* (NVS) fournie par l'ESP32.

Les données sont enregistrées dans une partition dédiée de la mémoire flash, configurée de manière à empêcher un accès direct non autorisé. Lorsque cela est possible, les informations stockées bénéficient d'un mécanisme de chiffrement, assurant qu'aucun identifiant sensible n'est conservé en clair sur le microcontrôleur.

Par ailleurs, aucun identifiant n'est directement intégré dans le code source du projet, ce qui limite les risques de fuite en cas de diffusion ou de partage du système. Enfin, le système propose une fonctionnalité de réinitialisation aux paramètres d'usine, permettant d'effacer l'ensemble des identifiants stockés. Cette option facilite la reconfiguration du dispositif et constitue une mesure de sécurité supplémentaire en cas de perte ou de changement de propriétaire.

6.4.4 Persistance de la configuration

Afin d'assurer un fonctionnement stable et cohérent du système après un redémarrage ou une coupure d'alimentation, la configuration du dispositif est stockée de manière persistante dans la mémoire *Non-Volatile Storage* (NVS) de l'ESP32. Cette approche permet de conserver l'ensemble des paramètres essentiels sans nécessiter de reconfiguration à chaque mise sous tension.

La gestion de cette persistance est assurée par le service **ConfigManager**, qui centralise l'accès aux paramètres de configuration et encapsule les opérations de lecture et d'écriture dans la mémoire NVS.

En isolant la gestion de la configuration dans un service dédié, l'architecture logicielle bénéficie d'une meilleure modularité et d'une maintenance beaucoup plus simple.

7 Tests et validation

7.1 Stratégie de test

La validation du système repose sur une stratégie de tests multi-niveaux, inspirée de la pyramide des tests. Cette approche vise à privilégier les tests les plus simples et les moins coûteux à exécuter, tout en complétant la validation par des tests plus complexes lorsque cela est nécessaire. Elle permet ainsi de garantir un bon compromis entre couverture fonctionnelle, rapidité d'exécution et fiabilité globale.

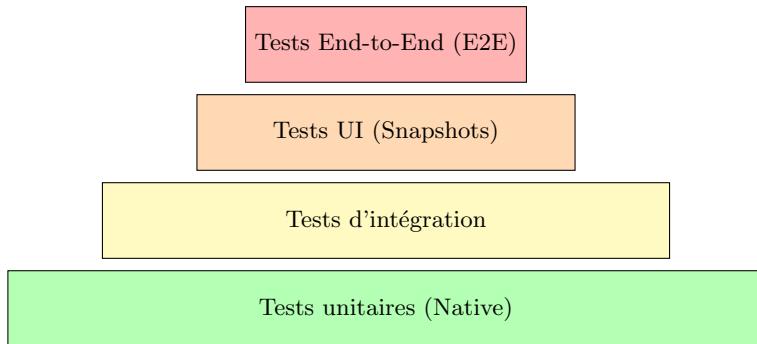


FIGURE 7 – Pyramide des tests du projet *PlantNanny*

Les tests unitaires constituent la base de cette pyramide, tandis que les tests d'intégration, les tests d'interface utilisateur et les tests de bout en bout viennent progressivement compléter la validation du système.

7.2 Framework de test

Nous avons utiliser le framework **Unity** pour la mise en œuvre des tests unitaires. Ce framework est particulièrement adapté aux systèmes embarqués et offre une compatibilité à la fois avec une exécution native sur poste de développement et avec une exécution directe sur le matériel embarqué, en l'occurrence l'ESP32.

Unity fournit un ensemble d'assertions simples et efficaces permettant de vérifier les résultats attendus, tout en facilitant l'automatisation des tests et leur intégration dans le cycle de développement.

Tests unitaires La couverture des tests unitaires porte essentiellement sur les modules génériques du projet, tels que les utilitaires, les patrons de conception utilisés et les fonctions de conversion et de traitement des données issues des capteurs. Afin de garantir une isolation complète des composants testés, les dépendances externes sont remplacées par des implémentations simulées via des interfaces.

Tests d'interface utilisateur Nous avons aussi mis en place des tests spécifiques pour valider le rendu de l'interface utilisateur affichée sur l'écran TFT. Ces tests, appelés tests de régression visuelle (tests *snapshot*) nous ont permis de nous assurer que les composants graphiques conservaient un rendu cohérent au fil des évolutions du code.

Ils portent notamment sur la disposition des éléments, leurs dimensions, leurs positions ainsi que leurs propriétés visuelles, telles que les couleurs, les marges ou les tailles de police. Cette approche permet de détecter rapidement toute régression graphique introduite lors de modifications du code de l'interface.

Tests des services réseau Les services réseau ont fait l'objet de tests dédiés visant à valider les mécanismes de connexion, de gestion des identifiants réseau et de récupération des informations système. Ces tests sont réalisés à l'aide de services simulés, permettant de couvrir différents scénarios, qu'il s'agisse de connexions réussies ou d'échecs liés à l'indisponibilité du réseau.

Tests sur matériel réel De plus, certains comportements du système ne pouvant être évalués correctement par simulation, une partie de la validation repose sur des tests exécutés directement sur le matériel physique. Ces tests concernent notamment la connexion Wi-Fi réelle, le processus de mise à jour du firmware à distance (*Over-The-Air*) ainsi que l'appairage Bluetooth Low Energy.

7.3 Exécution des tests

Nous avons organisé les tests de manière structurée au sein du projet, avec une séparation entre les tests unitaires, les tests simulés et les tests nécessitant l'utilisation du matériel réel.

L'exécution des tests a été automatisé à l'aide d'outils fournis par l'environnement de développement. Les tests peuvent être lancés aussi bien sur la plateforme native que sur le matériel embarqué, garantissant une validation complète du système tout au long du cycle de développement.

7.4 Résultats et limites

Les résultats obtenus ont montré que les principales fonctionnalités du système sont opérationnelles. L'initialisation des services, la communication Bluetooth, la connexion au réseau Wi-Fi, l'affichage sur l'écran TFT ainsi que la mise à jour OTA ont été validés avec succès. Ces résultats confirment la stabilité globale de l'architecture logicielle et la pertinence des choix techniques effectués.

Néanmoins certaines limites ont été identifiées. La précision des mesures dépend d'une calibration initiale des capteurs, l'autonomie du système reste conditionnée à une alimentation externe, et la portée du Bluetooth est limitée dans un environnement intérieur. De plus, le dispositif n'étant pas conçu pour résister à des conditions extérieures difficiles, son usage est recommandé en intérieur.

8 Outils de développement et infrastructure

Le développement du projet s'appuie sur un ensemble d'outils (Github, VSCode) destinés à faciliter la mise en place de l'environnement, la compilation du firmware, l'exécution des tests et le déploiement des services backend.

8.1 Scripts de développement

Plusieurs scripts ont été regroupés dans le répertoire `Devtools/`. Ils nous ont permis à automatiser les principales étapes du développement, depuis l'installation des dépendances jusqu'à l'exécution des tests, en passant par la compilation et le déploiement du code sur le microcontrôleur.

Script	Description
<code>setup.sh</code>	Installation des dépendances et configuration initiale de l'environnement de développement.
<code>build.sh</code>	Compilation du firmware embarqué sans déploiement sur le microcontrôleur.
<code>make.sh</code>	Compilation du firmware suivie de son téléchargement sur le microcontrôleur ESP32.
<code>run.sh</code>	Lancement du programme et affichage des logs via le port série pour le suivi de l'exécution.
<code>test.sh</code>	Exécution des tests unitaires sur la plateforme native ou embarquée.
<code>test.ui.snapshots.sh</code>	Lancement des tests de régression visuelle pour l'interface utilisateur.
<code>generate.sh</code>	Génération automatique de code à partir de modèles prédefinis.
<code>help.sh</code>	Affichage d'une aide synthétique présentant l'ensemble des scripts disponibles.

TABLE 11 – Scripts de développement

Ces scripts contribuent à uniformiser les pratiques de développement et à faciliter la prise en main du projet, tout en limitant les erreurs liées à des commandes manuelles.

8.2 Infrastructure Docker

Afin de garantir une infrastructure backend reproductible, et simple à déployer nous avons utiliser un conteneur Docker. Cette approche nous a permis de standardiser l'environnement d'exécution des services backend, d'éviter les conflits de dépendances entre systèmes et de simplifier les phases de développement.

L'infrastructure est composée de plusieurs services distincts, chacun encapsulé dans son propre conteneur et dédié à une responsabilité.

TABLE 12 – Conteneurs Docker utilisés dans le projet

Service	Description
<code>mqtt_broker/</code>	Broker MQTT Mosquitto configuré avec authentification et règles de contrôle d'accès, chargé de la communication entre les dispositifs embarqués et le backend.
<code>logger_broker/</code>	Service de journalisation distant permettant de centraliser les logs du système et de faciliter le diagnostic des erreurs.
<code>postgres/</code>	Base de données PostgreSQL utilisée pour la persistance des données applicatives, notamment l'historique des mesures et les informations de configuration.

Chaque service dispose de sa propre configuration Docker, incluant les paramètres réseau, les volumes persistants et les variables d'environnement nécessaires à son bon fonctionnement.

Le démarrage des services backend est automatisé à l'aide de scripts et de fichiers de configuration Docker, ce qui permet de lancer rapidement l'ensemble des composants nécessaires au fonctionnement du système.

Grâce à cette combinaison d'outils de développement et d'une infrastructure conteneurisée, le projet bénéficie d'un environnement de travail reproductible et facilement extensible.

9 Conclusion

Ce projet a permis de concevoir et expérimenter un système intelligent et embarqué reposant sur un microcontrôleur **ESP32 LilyGo T-Display**, intégrant à la fois des mesures de données d'environnement, une interface utilisateur embarquée et une architecture de communication adaptée aux contraintes de l'Internet des objets.

Les objectifs initiaux ont été atteints. Le dispositif assure une **surveillance continue de l'environnement de la plante** grâce à la lecture de trois capteurs : la température (thermistance NTC), l'humidité du sol (capteur capacitif) et la luminosité (photorésistance LDR). Les mesures sont converties à partir des valeurs brutes de l'ADC à l'aide de formules mathématiques documentées, incluant un moyennage des échantillons et des mécanismes de calibration. Les données ainsi obtenues sont publiées via le protocole **MQTT** sur un broker Mosquitto, garantissant une communication légère, temps réel et découpée entre le microcontrôleur, le backend et l'application mobile.

Sur le plan logiciel, le firmware s'appuie sur une architecture **orientée services**, favorisant une séparation claire des responsabilités et une meilleure maintenabilité. L'interface utilisateur embarquée, affichée sur l'écran TFT, constitue un apport significatif en termes d'ergonomie et de diagnostic, en fournissant à l'utilisateur une visualisation immédiate des données capteurs et de l'état du système.

Par l'allumage de la LED rouge, le projet démontre pleinement la faisabilité d'un système IoT complet, allant de l'électronique embarquée jusqu'aux services applicatifs et à l'interface utilisateur.

Dans son ensemble, cela a été une expérience formatrice dans laquelle nous avons pu nous confronter aux enjeux techniques, organisationnels et méthodologiques liés à la conception d'un système IoT complet. Il a permis de développer une approche rigoureuse de la conception embarquée, tout en soulignant l'importance de l'intégration entre matériel, logiciel et communication dans la réussite d'une solution utilisable en conditions réelles.