Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

Applicazione mobile per l'agricoltura intelligente: progetto TRACE

Relatore:
Prof.
Federico Montori

Presentata da:
Andrea Caselli

Correlatore:
Dott.
Ivan Zyrianoff

II Sessione
Anno Accademico 2024/2025

A mia mamma ...

Sommario

L'agricoltura contemporanea, negli ultimi anni, sta affrontando sfide sempre più critiche legate al cambiamento climatico, alla scarsità delle risorse idriche e alla necessità di incrementare la produttività riducendo l'impatto ambientale. L'Internet of Things (IoT) si posiziona come tecnologia fondamentale per l'evoluzione verso l'agricoltura di precisione, offrendo strumenti avanzati per il monitoraggio continuo delle condizioni ambientali e l'ottimizzazione dei processi produttivi.

Questo lavoro di tesi si inserisce nel progetto TRACE (Traceability and Resources in Agricultural Cultivation with Electronics), focalizzato sull'introduzione dell'Agricoltura 4.0 nel settore delle piante medicinali e aromatiche (MAPs). Nonostante il progetto disponesse di un'infrastruttura IoT operativa con sensori distribuiti nei campi che trasmettono misurazioni via protocollo LoRaWAN, la visualizzazione dei dati era affidata a dashboard web come Grafana, risultando meno immediate per l'utilizzo sul campo.

L'obiettivo principale è stato progettare, sviluppare e validare un'applicazione mobile completa per il monitoraggio dei dati agricoli raccolti dal sistema IoT. L'applicazione, sviluppata con Flutter e Dart, fornisce un'interfaccia ottimizzata per smartphone che permette agli agricoltori di consultare i dati dei sensori direttamente sul campo. Le funzionalità principali includono: visualizzazione interattiva dei dati tramite grafici e tabelle, navigazione di serie temporali con algoritmi di campionamento intelligente per garantire prestazioni fluide anche con migliaia di punti, configurazione di soglie personalizzate con monitoraggio automatico in background e invio di notifiche push al superamento dei valori critici, gestione multi-utente con ruoli differenziati (Admin, Consorzio, User) e sistema di autenticazione JWT.

Introduzione

L'agricoltura contemporanea si trova ad affrontare sfide senza precedenti: il cambiamento climatico, la crescente scarsità delle risorse idriche e la necessità di incrementare la produttività riducendo simultaneamente l'impatto ambientale stanno spingendo il settore verso una trasformazione radicale. In questo contesto, l'Internet of Things (IoT) emerge come tecnologia fondamentale per l'evoluzione verso l'agricoltura di precisione, offrendo agli operatori del settore strumenti avanzati per il monitoraggio continuo delle condizioni ambientali e per l'ottimizzazione dei processi produttivi basata su dati oggettivi.

Il presente lavoro di tesi si inserisce all'interno del progetto TRACE (Traceability and Resources in Agricultural Cultivation with Electronics), e ha come obiettivo l'introduzione dell'Agricoltura 4.0 nel settore MAPs (Medical and Aromatic Plants) attraverso lo sviluppo di un ecosistema digitale completo, che integra reti di sensori IoT, piattaforme di analisi dei dati, sistemi di tracciabilità blockchain e strumenti di supporto decisionale.

Nonostante il progetto TRACE disponesse già di un'infrastruttura IoT operativa per la raccolta dei dati ambientali e pedologici, con sensori distribuiti nei campi che trasmettono misurazioni via protocollo LoRaWAN, la visualizzazione e l'analisi di questi dati erano affidate a dashboard web come Grafana. Sebbene queste piattaforme forniscano funzionalità avanzate di visualizzazione, risultano meno immediate per l'utilizzo sul campo da parte degli operatori agricoli, che necessitano di uno strumento accessibile rapidamente tramite dispositivo mobile.

L'obiettivo principale di questo lavoro è stato progettare, sviluppare e validare un'applicazione mobile completa per il monitoraggio dei dati agricoli raccolti dal sistema IoT del progetto TRACE. L'applicazione doveva rispondere a requisiti identificati attraverso l'analisi delle esigenze degli utenti finali: (i) fornire un'interfaccia mobile ottimizzata per l'utilizzo sul campo, permettendo agli agricoltori di consultare i dati dei sensori direttamente dallo smartphone; (ii) implementare dei grafici interattivi, delle tabelle dettagliate e degli strumenti di analisi che permettano di esplorare serie temporali

anche molto estese, con algoritmi di campionamento intelligente per garantire prestazioni fluide su dispositivi mobili anche con migliaia di punti dati; (iii) consentire agli utenti di configurare soglie personalizzate per i parametri critici, con invio automatico di notifiche push al superamento dei valori impostati, anche quando l'applicazione non è attiva.

Il documento è organizzato come segue: il Capitolo 1 presenta lo stato dell'arte, analizzando il ruolo dell'IoT nell'agricoltura di precisione, le caratteristiche dei time series database e un confronto tra applicazioni mobili esistenti per il settore agricolo, posizionando TRACE Project app nel panorama delle soluzioni disponibili; il Capitolo 2 introduce il progetto TRACE, descrivendo l'architettura dell'infrastruttura IoT esistente, i sensori distribuiti nei campi (Weather Stations e Soil Moisture Stations), e il contesto del settore MAPs che ha motivato lo sviluppo del sistema; il Capitolo 3 affronta la fase di progettazione, illustrando l'architettura dell'applicazione a tre livelli (presentation layer, business logic layer, data layer), la scelta delle tecnologie utilizzate (Flutter, Dart, Shelf, PostgreSQL, InfluxDB), il design del database e l'integrazione con Firebase Cloud Messaging; il Capitolo 4 documenta l'implementazione, analizzando la struttura del backend con la pipeline di middleware, il sistema di autenticazione JWT, l'integrazione con InfluxDB, il servizio di monitoraggio delle soglie, e lato frontend analizzando la gestione dello stato, la comunicazione HTTP e gli algoritmi di campionamento; il Capitolo 5 descrive le validazioni effettuate: la migrazione del backend dall'ambiente di sviluppo a quello di produzione, il processo di pubblicazione su Google Play Store con le diverse fasi di testing, e l'analisi dettagliata dei risultati del questionario PSSUQ somministrato a 11 utenti durante il closed testing; infine, le conclusioni sintetizzano i risultati ottenuti, discutono i limiti del lavoro svolto e propongono direzioni future per l'evoluzione del sistema.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei Codici

Capitolo 1

Stato dell'arte

Negli ultimi anni il settore agricolo è stato oggetto di una trasformazione radicale, tuttora in corso. Come spiegato nell'articolo [4], il surriscaldamento globale, la continua crescita demografica e l'avvento di periodi sempre più frequenti di siccità rappresentano un problema reale e improrogabile per quasi la totalità dei settori, in particolare quello agricolo. È necessario un evoluzione nella metodologia e nella tecnologia applicata all'agricoltura. L'agricoltura di precisione rappresenta una soluzione praticabile, promuovendo pratiche sostenibili, ottimizzando le risorse e migliorando l'efficienza attraverso l'utilizzo di un sistema decisionale basato sull'Internet of Things (IoT).

L'utilizzo dei sistemi IoT sta diventando sempre più frequente all'interno del settore agricolo e rappresenta uno degli strumenti principali per affrontare le sfide poste dai cambiamenti climatici. L'integrazione di sensori intelligenti nei campi, di sistemi di monitoraggio e di piattaforma di analisi di dati avanzate consente agli agricoltori di avere una visione accurata in tempo reale delle condizioni dei campi, facilitando decisioni basate su dati reali. Attraverso delle reti di dispositivi interconnessi, l'IoT permette di raccogliere informazioni fondamentali, migliorando così la capacità di intervenire in modo preciso e tempestivo.

Inoltre, come discusso nell'articolo [2], l'IoT si configura come un'architettura scalabile e flessibile, con la capacità di adattarsi alle esigenze di aziende agricole di ogni dimensione, dai piccoli produttori locali alle realtà industriali. La possibilità di integrazione di tecniche di analisi dei dati, algoritmi predittivi e automazione intelligente permette non solo di ottimizzare i processi produttivi, ma anche di ridurre sprechi e diminuire significativamente l'impatto ambientale. Questa evoluzione tecnologica e la possibilità di applicarla in modo scalabile, rappresenta una condizione fondamentale per lo sviluppo di sistemi agricoli efficienti e sostenibili nel lungo periodo.

1.1 IoT e gestione dei dati

L'Internet of Things si fonda sulla connessione e l'interoperabilità tra oggetti fisici in grado di generare dati in modo continuo. Nell'articolo [2], gli autori forniscono una definizione fondamentale del modello architetturale dei sistemi basati sull'IoT, identificando componenti indispensabili quali sensori, connettività, cloud computing, elaborazione distribuita e applicazioni finali. I sensori rappresentano il punto di partenza del sistema, questi leggono e trasformano in dati i parametri ambientali, e sono progettati per funzionare in modo affidabili anche in condizioni climatiche critiche e variabili. La connettività garantisce il trasferimento di questi dati verso l'infrastruttura centrale tramite tecnologie come Wi-Fi o protocolli a basso consumo con LoRaWAN. Il livello di cloud computing fornisce invece la capacità computazionale e lo spazio di archiviazione e storage necessari per conservare questi dati in grandi quantità. Infine, l'elaborazione distribuita permette al sistema di essere flessibile e scalabile, per ridurre la latenza, mentre le applicazioni finali rappresentano rappresentano l'interfaccia da cui gli utenti accedono ai risultati dell'elaborazione dei dati. Questo schema mostra come la raccolta e la gestione dei dati dei sistemi IoT sia un elemento centrale e significativo.

Con l'aumento esponenziale del numero dei dispositivi IoT, cresce proporzionalmente anche la quantità di dati prodotta, dando origine a flussi di informazioni continui, tipici delle serie temporali. Gli autori di [7] analizzano le caratteristiche e le criticità dei time series database (TSDB) in ambito IoT, evidenziando la necessità di sistemi in grado di sostenere un'acquisizione dei dati ad alta frequenza, che creano flussi continui in entrata che un database relazionale faticherebbe a gestire. A questo si affianca la necessità di una compressione efficiente, necessaria per ottimizzare lo spazio utilizzato nel dataset e per garantire allo stesso tempo un accesso rapido ai dati in lettura. Infine, la scalabilità orizzontale del sistema viene identificata come un requisito indispensabile per distribuire il carico di lavoro su più nodi, mantenendo le prestazioni ad un livello elevato. Questi requisiti risultano fondamentali durante lo sviluppo di dashboard e piattaforme di monitoraggio che integrano l'elaborazione in tempo reale con la visualizzazione storica dei dati.

Oltre ai TSDB, altre tecniche di data processing risultano centrali nei sistemi IoT, come lo stream processing, la window aggregation e il downsampling. L'analisi continua del flusso dei dati è essenziale, permettendo di generare notifiche, attivare automatismi o fare previsioni su fenomeni monitorati.

1.1.1 IoT nell'agricoltura di precisione e smart irrigation

Come detto precedentemente, l'agricoltura è uno dei settori che ha tratto più beneficio dall'evoluzione e dall'implementazione dei sistemi IoT, soprattutto grazie alla possibilità di ottenere informazioni granulari e aggiornate in tempo reale sulle condizioni atmosferiche e del terreno. L'agricoltura di precisione ha come scopo l'ottimizzazione degli input come acqua, fertilizzanti ed energia sulla base di misurazioni e dati oggettivi, riducendo gli sprechi ed aumentando parallelamente l'efficienza e la sostenibilità. Come definito anche dall'articolo [5]: «precision agriculture is the application of technologies and principles to manage spatial and temporal variability associated with all aspects of agricultural production for the purpose of improving crop performance and environmental quality.».

Nell'articolo [4], gli autori presentano un'analisi completa dello stato dell'arte dei sistemi IoT implementati nell'agricoltura, evidenziando come la miniaturizzazione dei sensori, l'integrazione con reti wireless a basso consumo e i progressi nell'elaborazione di sistemi distribuiti abbiano reso l'IoT uno strumento quasi indispensabile per le aziende agricole moderne. Nell'articolo vengono anche evidenziate diversi problemi legati all'applicazione di questi sistemi come la sicurezza dei dati, l'interoperabilità tra dispositivi di marche differenti, robustezza in ambienti rurali difficili e la necessità di modelli standard condivisi.

Tra i molteplici utilizzi dell'IoT in ambito agricolo, l'irrigazione intelligente emerge come una delle più significative dal punto di vista dell'impatto ambientale ed economico. Il poter gestire in modo ottimale la risorsa idrica rappresenta infatti una sfida cruciale nell'agricoltura contemporanea. I sistemi di smart irrigation si basano sull'integrazione di sensori distribuiti capaci di monitorare in modo continuo parametri critici, come l'umidità del suolo e temperatura ambientale. Questi dati, se acquisiti con frequenza eleveta e continua, generano flussi informativi che richiedono infrastrutture dedicate per l'archiviazione, l'elaborazione e la visualizzazione.

Come detto precedentemente, l'architettura di un sistema di irrigazione intelligente prevede diversi strati tecnologici interconnessi. Al livello più basso si collocano i dispositivi sensoriali operanti con protocolli a basso consumo energetico. Questi sensori trasmettono periodicamente misurazioni verso gateway o piattaforme cloud, dove i dati vengono salvati in database e ottimizzati attraverso algoritmi di validazione, che filtrano letture anomale, e procedure di normalizzazione.

In generale, la smart irrigation richiede strumenti per:

- acquisire serie temporali da sensori eterogenei;
- filtrare e normalizzare i dati;
- visualizzare i dati tramite dashboard comprensibili e user-friendly;
- generare notifiche e allarmi in caso di valori critici.

1.2 Applicazioni mobili per l'agricoltura di precisione

Il panorama delle applicazioni mobile applicate all'agricoltura di precisione si è notevolmente arricchito negli ultimi anni, con soluzioni che variano per complessità, target di utenza e funzionalità offerte. Di seguito verrà effettuata un'analisi comparativa tra alcune di queste applicazioni.

SWAMP Farmer App [1] rappresenta un esempio di integrazione tra sensori IoT e interfaccia mobile per il controllo dell'irrigazione. Il sistema si distingue per l'implementazione di un loop di controllo che abbina il monitoraggio continuo dello stato idrico del suolo all'attuazione automatica degli interventi di irrigazione. L'architettura proposta dagli autori è divisa in tre livelli: dispositivi edge per la raccolta dei dati, middleware cloud per l'elaborazione e lo storage, e una applicazione mobile come interfaccia utente. L'applicazione permette agli agricoltori di definire soglie personalizzate per umidità e temperatura del suolo, ricevere notifiche push al superamento dei limiti impostati, e attivare remotamente i sistemi di irrigazione. Questo approccio risulta efficace in contesti di agricoltura intensiva dove il controllo della risorsa idrica impatta direttamente sulla produzione finale. Tuttavia, questo livello di automatizzazione può risultare limitante per le realtà agricole che privilegiano un approccio decisionale più diretto da parte dell'operatore.

Crop Connect [6] è un'applicazione focalizzata sulla dimensione sociale e comunitaria dell'agricoltura digitale. Gli autori evidenziano come la condivisione di conoscenze ed esperienze tra agricoltori rappresenti un asset spesso sottovaluto. L'applicazione integra funzionalità di social networking progettate per il settore agricolo, permettendo agli utenti di pubblicare osservazioni sui propri campi e confrontare strategie agronomiche. Questa dimensione comunitaria risulta rilevante in contesti dove la frammentazione delle

aziende agricole rende difficile l'accesso e l'ottenimento di consulenze specializzate. Tuttavia, l'app presenta limitazioni nell'integrazione dei sistemi IoT, delegando prevalentemente all'utente l'input manuale delle informazioni.

Farm Management System cloud-based[3] rappresenta un approccio più sistemico della gestione aziendale agricola. Gli autori descrivono un'architettura orientata verso i microservizi, che integra moduli per la pianificazione colturale, gestione del magazzino, analisi economica e supporto decisionale. La piattaforma si distingue per l'ampiezza funzionale, proponendosi come soluzione ERP (Enterprise Resource Planning) per il settore agricolo. L'integrazione dei sensori IoT rappresenta una delle funzionalità dell'applicazione ma non costituisce il fulcro del sistema. Questo approccio può risultare utile e adeguato per azienda agricole di dimensioni medio-grandi, ma può risultare complesso per piccoli produttori o consorzi che necessitano principalmente di strumenti per il monitoraggio ambientale.

Come verrà spiegato e analizzato nei capitoli successivi, l'applicazione TRACE Project App si inserisce in questo panorama in modo eterogeneo, proponendo un posizionamento ibrido che integra elementi delle diverse filosofie progettuali elencate sopra. L'applicazione mobile del progetto TRACE condivide con SWAMP Farmer App l'enfasi sul monitoraggio IoT in tempo reale e sulla generazione di notifiche di alert basato su delle soglie configurabili, ma si differenza nell'approccio di controllo: mentre SWAMP privilegia l'automazione, TRACE mantiene l'agricoltore al centro del processo decisionale, fornendo degli strumenti di supporto. Le notifiche ricevute dall'applicazione TRACE segnalano anomalie o superamenti delle soglie, ma la responsabilità di azione resta completamente in mano all'utente.

Della dimensione comunitaria di Crop Connect, TRACE eredita il concetto di consorzio come entità centrale. La gestione multi-utente facilita la condivisione delle risorse sensoriali e di informazioni tra membri dello stesso consorzio. Questa scelta progettuale riflette il contesto operativo italiano dei consorzi e delle cooperative agricole, dove di norma la collaborazione avviene all'interno di strutture formali preesistenti piuttosto che all'interno di community online.

A differenza dell'approccio ERP, TRACE adotta una strategia di specializzazione focalizzata. L'applicazione non punta alla gestione dell'intera filiera operativa, ma si concentra sull'eccellenza nella visualizzazione e analisi dei dati ambientali. Per questo l'applicazione implementa strumenti utili e ottimizzati per la navigazione storica di grandi quantità di dati, con confronto multi-sensore, e algoritmi di campionamento.

In sintesi, l'applicazione TRACE si configura come una soluzione che bilancia la specializzazione funzionale e l'usabilità, privilegiando la visualizzazione analitica dei dati in maniera accurata. Il posizionamento risponde alle esigenze specifiche del contesto italiano dell'agricoltura consorziale, dove la condivisione di infrastrutture sensoriali tra aziende rappresenta un modello organizzativo consolidato e che richiede strumenti digitali adeguati.

Capitolo 2

Progetto TRACE

Un esempio dell'impiego della tecnologia IoT all'interno del settore agricolo è quella del progetto TRACE. Come spiega l'articolo [8], TRACE è un progetto finanziato dalla regione Emilia Romagna all'interno del programma CoPSR e coinvolge un consorzio di istituzioni accademiche e agricole specializzate nella coltivazione di Medical and Aromic Plants (MAPs). Lo scopo del progetto è l'introduzione dell'Agricoltura 4.0 all'interno del settore specializzato, sviluppando un ecosistema digitale avanzato basato sulle infrastrutture IoT.

2.1 Settore MAPs e applicazione di sistemi IoT

Il settore MAPs negli ultimi anni ha registrato una crescita significativa, guidata dall'aumento della domanda di questa tipologia di piante all'interno del settore medico, farmaceutico, nutraceutico e cosmetico. Le piantagioni MAPs sono caratterizzate da un elevato contenuto di composti bioattivi, la cui qualità dipende fortemente dalle condizioni di crescita. A differenza delle altre colture le MAPs necessitano di condizioni atmosferiche precise e regolate e richiedono quindi un controllo rigoroso dei paramentri ambientali, come temperatura, umidità del suolo e dell'aria e qualità del terreno.

Le caratteristiche e le precise condizioni necessarie per la crescita di una piantagione MAPs rendono questo settore adatto all'adozione di sistemi IoT. L'applicazione di sensori distribuiti permette infatti il monitoriaggio continuo e in tempo reale delle condizioni di crescita della pianta, offrendo

agli agricoltori la possibilità di individuare e correggere situazioni critiche. Grazie alla raccolta sistematica dei dati, l'operatore agricolo può agire tempestivamente su eventuali problemi e, ad esempio, regolare l'irrigazione, prevenire l'incidenza di malattie e molto altro.

Un ulteriore elemento che distingue il settore MAPs dalle altre colture è la crescente richiesta di tracciabilità, certificazioni biologiche e conformità agli standard qualitativi sempre più rigidi richiesti dal settore medico-farmaceutico. La sensibilità di questa tipologia di piante alle condizioni di crescita rende necessario la documentazione delle pratiche agronomiche e la qualità dell'ambiente produttivo. L'integrazione di sistemi IoT, in parallelo a tecnologie blockchain, consente la registrazione automatica e verificabile dei dati raccolti dai sensori. Questo approccio garantisce, quindi, trasparenza e tracciabilità lungo tutta la catena di produzione.

L'utilizzo e l'implementazione di queste tecnologie unisce quindi il monitoraggio avanzato, il supporto agli agricoltori nelle decisioni da adottare e la certificazione digitale tutto in un'unica infrastruttura. Questo rende l'IoT un elemento di innovazione tecnica e un fattore abilitante per la sostenibilità del settore MAPs, aiutando a migliorare la qualità del prodotto e a soddisfare i requisiti necessari per la trasparenza.

2.2 Architettura del progetto TRACE

Come spiegato precedentemente, TRACE è un'iniziativa finalizzata all'innovazione del settore agricolo delle MAPs. In particolare TRACE mira a conseguire tre principali obiettivi: (i) garantire la qualità delle coltivazioni attraverso un sistema di monitoraggio in tempo reale, permettendo e abilitando un approccio di agricoltura di precisione; (ii) supportare la certificazione e la tracciabilità dei prodotti tramite sistemi digitali di registrazione sicura dei dati; (iii) migliorare la produttività e la sostenibilità attraverso decisioni strategiche data-driven, ottenute tramite l'integrazione dei dati IoT, informazioni agricole già esistenti e tecniche di elaborazione e analisi avanzate.

Per soddisfare questi requisiti è stata implementata l'architettura illustrata nella Figura 2.1, composta da tre macro-componenti.

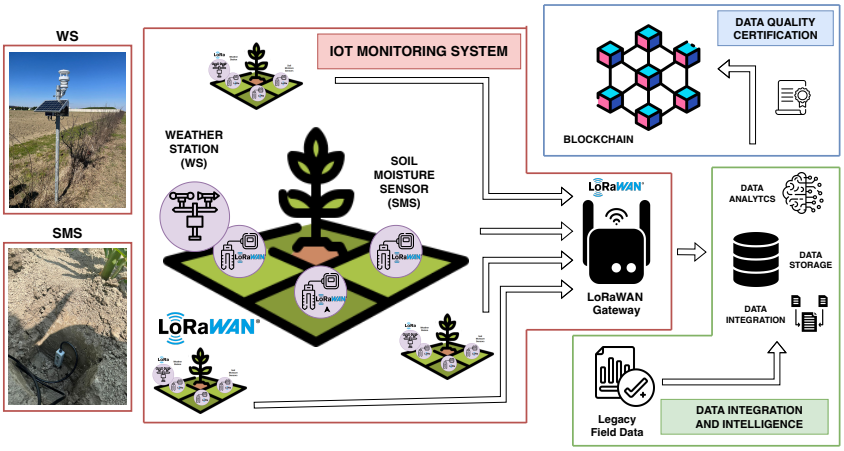


Figura 2.1: Architettura del sistema TRACE.

Sistema IoT di monitoraggio Il primo livello comprende l'infrastruttura di prelievo, comunicazione e gestione dei dati necessaria a raccogliere misurazioni ambientali e del suolo in modo dettagliato direttamente nelle zone agricole. Il prelievo dei dati è stato implementato tramite l'impiego di due tipologie di dispositivi: Weather Station (WS), adibita al prelievo dei parametri atmosferici, e le Soil Moisture Stations (SMS), specializzate nel raccoglimento di dati pedologici.

L'implementazione di questo sistema basato sulle tecnologie IoT risponde a vincoli molto rigidi dello smart farming:

- efficienza energetica, poiché i dispositivi sono installati in ambienti privi di alimentazione stabile e devono operare lungo periodi estesi di tempo come le stagioni colturali;
- affidabilità della comunicazione, di cui è necessario il funzionamento anche in condizioni atmosferiche rigide, in caso di interferenze radio o di malfunzionamenti locali delle unità;
- robustezza nel lungo periodo, fondamentale per ottenere dataset completi, robusti e continuativi.

Integrazione dei dati e intelligenza Il secondo livello integra i dati provenienti dal sistema IoT con altre fonti informative presenti all'interno delle aziende agricole, come statistiche di resa, analisi del suolo e registri delle pratiche agronomiche. Questa componente implementa anche l'utilizzo di strumenti di data analytics e l'utilizzo di intelligenza artificiale volti allo

studio e alla ricerca di correlazioni tra parametri ambientali e del terreno con l'output produttivo. Un esempio è la definizione di strategie di irrigazione ottimizzate che combinano i dati provenienti dal sistema IoT, dati storici e informazioni agronomiche specifiche.

Tracciabilità e certificazione Il terzo livello riguarda la tracciabilità lungo tutta la catena di produzione delle MAPs, partendo dalla fase di coltivazione fino ad arrivare a quella di trasformazione ed elaborazione del prodotto. In questo contesto , l'implementazione di strumenti come i Digital Field Notebook (DFN), in alcuni casi già richiesti da normative nazionali, rappresentano una delle migliori soluzioni per registrare e consultare tutte le attività svolte durante la fase di coltivazione.

TRACE integra il DFN con tecnologie blockchain, con l'obiettivo di ottenere registri immutabili e verificabili delle operazioni agricole, permettendo una maggiore trasparenza e fiducia tra tutti gli attori della supply chain del prodotto.

2.3 Implementazione del progetto TRACE

Uno degli elementi centrali del progetto TRACE riguarda l'implementazione del sistema IoT sviluppato sulla base dell'architettura spiegata nella sezione 2.0.2. Questa implementazione è stata svolta seguendo i tre requisiti fondamentali elencati precedentemente. Per soddisfare questi vincoli sono state adottate alcune scelte progettuali chiave: (i) utilizzo di dispositivi IoT commerciali; (ii) adozione della tecnologia LoRa per l'invio e la comunicazione a lungo raggio a basso consumo energetico; (iii) impiego della infrastruttura LoRaWAN che opera tramite server open-source ChirpStack; (iv) integrazione di software custom per la gestione dei dati e della loro visualizzazione.

L'utilizzo delle tecnologie LoRa e LoRaWAN è ormai comune all'interno dell'ambito agricolo, grazie alle loro capacità di coprire grandi distanze a basso consumo energetico e di trasmettere a più gateaway senza ulteriori procedure di associazione. Un esempio di implementazione di queste tecnologie è il progetto SWAMP [1]. SWAMP ha dimostrato come l'utilizzo di messaggi downlink LoRaWAN nella riconfigurazione dinamica della frequenza di campionamento dei sensori possa estendere in modo significativo la durata delle batterie e diminuire al minimo la necessità di interventi di

manutenzione.

Nel contesto TRACE, ogni sensore WS è responsabile della rilevazione di dati ambientali quali temperatura, umidità, pressione atmosferica, precipitazioni e dati relativi alla direzione e velocità del vento. Nel complesso sono state installate otto WS, una per ogni azienda agricola aderente al progetto. Parallelamente sono stati distribuiti venti sensori SMS in aree strategiche per ottenere una copertura massimale e strategica. I sensori SMS sono stati installati a circa 30 cm di profondità e vengono utilizzati per il prelievo di parametri del suolo come umidità, temperatura e conducibilità elettrica.

L'area agricola monitorata si estende ad una distanza massima di 10 km tra i punti più distanti. Per garantire una copertura LoRaWAN più adeguata e ottimale e per massimizzare la ridondanza sono stati installati tre LoRaWAN gateway sui tetti di edifici in posizioni strategiche. Ogni gateway è connesso alla rete tramite rete 5G o collegamento tramite cavo ethernet.

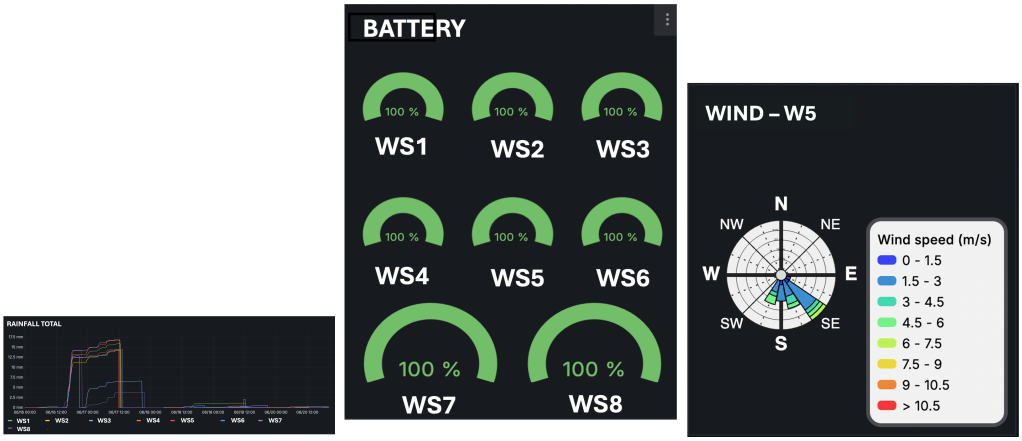
Dal punto di vista software è stato utilizzata la piattaforma cloud open-source ChipStark, utilizzata sia per il Network Server (NS) che per l'Application Server (AS). Viene poi utilizzata un'applicazione personalizzata, sviluppata all'interno del progetto che si occupa di

- recuperare messaggi inviati tramite connessione MQTT;
- decodificare i payload;
- filtrare campioni errati o rumorosi;
- memorizzare i dati validati all'interno di un database InfluxDB [1], scelto per l'efficienza nella gestione di serie temporali.

Infine, per consentire agli utenti un accesso semplice e immediato ai dati è stata realizzata una dashboard Grafana. Questa fornisce tre funzionalità semplici: (i) monitoraggio di dati ambientali e del suolo con la possibilità di visualizzare dati aggregati come media, minimo e massimo; (ii) monitoraggio dei dispositivi con informazioni come carica delle batterie dei sensori; (iii) analisi geospaziale tramite mappe interattive. Le figure 2.2a 2.2b 2.2c mostrano alcuni esempi delle interfacce implementate su Grafana.

[1] https://www.influxdata.com/

Nonostante l'efficacia della dashboard, l'utilizzo di Grafana presenta dei limiti per gli utenti sul campo, in particolare per gli agricoltori che necessitano di uno strumento rapido, intuitivo e direttamente accessibile da dispositivo mobile. Per questo, per rispondere alle esigenze, è stata sviluppata l'applicazione TRACE Project App. L'app fornisce un accesso immediato ai dati raccolti dai sensori e permette un controllo costante delle condizioni atmosferiche e del suolo, rendendo così il sistema IoT del progetto ancora più efficace e utilizzabile nelle attività quotidiane svolte sul campo.



(a) (b) (c)

Figura 2.2: Esempi di visualizzazioni dei dati nella dashboard Grafana

Capitolo 3

Progettazione

L'applicazione TRACE è stata creata con lo scopo di fornire agli utenti finali uno strumento intuitivo e accessibile per il monitoraggio dei dati agricoli raccolti dai sensori presenti nei campi. Come detto precedentemente, la visualizzazione e l'analisi dei dati erano prima affidate a piattaforme web come Grafana, che, nonostante rispettino tutti i requisiti richiesti, risultano meno immediate per un utilizzo sul campo. L'applicazione si pone quindi come alternativa user-friendly e più facilmente accessibile, offrendo funzionalità avanzate come notifiche push-up in tempo reale, gestione multi-utente e un'interfaccia grafica ottimizzata per dispositivi mobili.

3.1 Panoramica generale

L'applicazione è strutturata secondo un'architettura a tre livelli per garantire la separazione delle responsabilità, scalabilità e manutenibilità del sistema. L'architettura dell'app separa la logica di presentazione, la logica di business e la gestione dei dati in layer distinti e indipendenti l'uno dall'altro, permettendo lo sviluppo e il testing di ciascun componente in modo isolato.

Il primo livello, il Presentation Layer, è rappresentato dall'applicazione mobile sviluppata in Flutter [1], che fornisce l'interfaccia utente e gestisce l'interazione con l'utente. Questo layer ha come scopo la visualizzazione dei dati e la raccolta degli input dell'utente, delegando la logica di business al livello successivo.

[1] https://flutter.dev/

Il secondo livello, il Business Logic Layer, è implementato tramite backend sviluppato in Dart [2] che espone API REST. Questo layer gestisce e coordina le operazione tra database e frontend, implementa le regole di business, gestisce l'autenticazione degli utenti e i loro permessi, e amministra i servizi in background.

Il terzo livello, il Data Layer, è costituito da due database: InfluxDB per le soglie temporali prelevate dai sensori IoT, e PostgresSQL per la gestione dei dati strutturati e relazionali come utenti, note di irrigazione e associazioni dei sensori.

Trasversalmente, il sistema implementa un servizio cloud esterno, Firebase Cloud Messaging (FCM)[3], per l'invio di notifiche push ai dispositivi. Questo servizio viene gestito completamente dal backend ma ha un impatto significativo sull'esperienza dell'utente, creando un canale di comunicazione tra il sistema e gli utenti. L'utilizzo di FCM è indispensabile perchè i sistemi operativi mobili (iOS e Android) non permettono ai server backend di inviare direttamente le notifiche push. FCM ha la funzione di intermediario che autentica il mittente, mantiene un canale di comunicazione affidabile anche quando l'app è in background, e ottimizza la consegna dei messaggi riducendo il consumo di batteria e dei dati internet. Senza l'utilizzo del servizio FCM, non sarebbe possibile garantire che le notifiche raggiungano gli utenti in modo sicuro, affidabile e scalabile.

Il flusso di dati tipico attraversa l'architettura secondo questo ciclo: i sensori IoT trasmettono i dati prelevati tramite LoRaWAN ad un gateway centrale che li salva sul database InfluxDB. All'apertura dell'app da parte di un utente viene mandata una richiesta HTTP al backend specificando quali dati si vuole visualizzare. Il backend, dopo aver verificato l'identità e i permessi dell'utente, interroga InfluxDB per recuperare i dati time-series richiesti e PostegresSQL per ottenere informazioni sui sensori e sulle configurazioni. I dati vengono poi aggregati, trasformati in JSON e inviati all'applicazione che li visualizza tramite grafici e tabelle.

L'applicazione offre una funzionalità che permette agli utenti di creare delle soglie personalizzabili per tenere traccia dei valori critici prelevati dai sensori. Questo avviene tramite un servizio in background, gestito dal backend, monitora in loop i valori dei sensori rispetto alle soglie configurate dagli utenti. Quando una soglia viene superata e sono trascorsi gli intervalli minimi configurati, il servizio invia una notifica push all'utente tramite FCM, anche se l'applicazione è chiusa.

[2] https://dart.dev/

[3] https://firebase.google.com/docs/cloud-messaging

3.1.1 Frontend Mobile: Presentation Layer

Il frontend dell'applicazione è stato sviluppato utilizzando Flutter, framework open-source di Google per lo sviluppo di applicazioni mobili cross-platform. Flutter permette la creazione di applicazioni native per Android e iOS da un'unica codebase scritta in linguaggio Dart.

La scelta di Flutter come framework per il frontend è motivata da diversi fattori tecnici:

Cross-Platform Nativo Flutter compila il codice Dart in codice macchina nativo per ciascuna piattaforma, garantendo performance paragonabili ad applicazioni sviluppate con SDK nativi. A differenza dei framework basati su WebView, Flutter non soffre di overhead di runtime.

Consistent UI il sistema di rendering di Flutter disegna ogni pixel dello schermo, garantendo un'interfaccia identica su tutte le piattaforme e versioni di sistema operativo, eliminando i problemi di inconsistenza visiva e comportamentale.

Funzione di Hot Reload la funzionalità di hot reload permette di vedere le modifiche applicate al codice immediatamente nell'app senza perdere lo stato corrente, accelerando il ciclo di sviluppo e facilitando l'iterazione sul design dell'interfaccia.

Widget-Based Architecture Flutter adotta un approccio dichiarativo alla costruzione dell'UI, dove l'interfaccia è composta da widget riusabili e componibili, facilitando la manutenzione del codice e la creazione di componenti personalizzabili.

Il Presentation Level gestisce tutte le interazioni con l'utente e si occupa di:

- Rendering dell'interfaccia utente: visualizzazione di schermate, widget, grafici e tabelle. L'interfaccia è ottimizzata per dispositivi di diverse dimensioni, sia smartphone che tablet;
- Gestione input dell'utente: cattura le interazioni touch dell'utente (tap, swipe ecc), input da tastiera, selezioni da dropdown e date picker, con validazione client-side immediata per migliorare la user-experience;
- Comunicazione con il backend: esecuzione delle chiamate HTTP alle API REST del backend e gestione dell'autenticazione tramite token

JWT. Quest'ultimo permette al backend di identificare gli utenti e i loro permessi, vietando ad utenti non loggati o con ruoli non autorizzati di effettuare azioni e chiamate al backend non permessi;

- State management locale: gestione dello stato dell'applicazione (dati visualizzati, configurazioni utente, cache temporanea) utilizzando diversi pattern. Per lo stato locale delle singole schermate viene utilizzato StatefulWidget, un tipo di widget in Flutter che mantiene i valori in modo persistente all'interno della schermata e che aggiorna l'interfaccia ogni volta che lo stato cambio. Per gli stati condivisi tra più schermate, o per servizi globali, viene utilizzato il pattern Singleton, che garantisce l'esistenza di una solo istanza dell'oggetto condivisa all'interno di tutta l'applicazione, che permette una gestione facilitata delle preferenze utente e della cache globale;

- Caching e persistenza locale: salvataggio di configurazioni, token di autenticazione e preferenze dell'utente;

- Notifiche push: integrazione con FCM per ricevere e visualizzare notifiche push anche quando l'applicazione non è aperta:

Ogni operazione effettuata dal frontend che richiede elaborazione o accesso ai dati passa obbligatoriamente attraverso il backend, garantendo sicurezza e consistenza.

3.1.2 Backend API: Business Logic Layer

Il backend è implementato tramite server API REST sviluppato in Dart utilizzando il framework Shelf [4]. Questa scelta tecnologica garantisce coerenza linguistica con il frontend e permette di condividere i modelli di dati e la logica di business tra gli altri due livelli.

Dart, tipicamente utilizzato per lo sviluppo del fronted, si è dimostrato un linguaggio efficace anche per le applicaziioni server-side:

Type Safety Dart è un linguaggio fortemente tipizzato con type inference che permette di ridurre gli errori a runtime e facilità l refactorin. Il compilatore Dart cattura errori comuni durante la fase di sviluppo anziché in produzione.

[4] https://pub.dev/packages/shelf

Perfomance la Dart VM (Virtual Machine) e il compilatore AOT (Ahead-of-time) producono codice ottimizzato. Per applicazione server, Dart offre performance comparabii a Node.js e altri linguaggi come Python per operazioni intensive.

Asynchronous Programming Dart ha support nativo per la programmazione asincrona tramite async/await e Future/Stream, essenziale per la gestione della concorrenza in applicazioni server che devono gestire richieste simultanee. Un Future rappresenta un valore che sarà disponibile in futuro, per esempio il risultato di una chiamata al backend o di un operazione I/O (Input Output). Le keywors async/await permettono di scrivere codice asincron: dichiarando una funziona async e usando await davanti ad un Future, il programma attende il completamento dell'operazione senza bloccare il thread principale. Gli Stream, invece, rappresenta una sequenza di valori che arrivano nel tempo, utilizzati per la gestione di eventi multipli o flussi di dati con aggiornamento in tempo reale.

Condivisione di codice utilizzare Dart sia per il frontend che per il backend permette di condividere definizioni di modelli di dati, algoritmi di validazione e logica di business, riducendo la duplicazione di codice.

Ecosistema Server l'ecosistema Dart include package per lo sviluppo server come Shelf (web server), PostgresSQL (driver del database), JWT (autenticazione), BCrypt (hashing delle password) e logging.

Il livello di business logic è il cuore del sistema e si occupa di:

- Autenticazione e autorizzazione: gestione dell'intero ciclo di autenticazione dell'utente, che comprende registrazione, login, refresh dei token JWT e logout. Inoltre è stato implementato il Role-Based Access Control (RBAC) per verificare che ogni utente possa accedere solo alle risorse autorizzate in base al proprio ruolo;

- Validazione: ogni dato ricevuto dal frontend viene validato per tipo, formato, range di valori e vincoli di business prima di essere processato o salvato. Questo per prevenire attacchi di tipo injection e garantire integrità dei dati all'interno dei database;

- Business logic: implementazione delle regole di business complesse come il workflow di approvazione (consorzio si registra → admin approva → consorzio operativo), calcolo statistiche aggregate e gestione delle associazioni sensori-consorzi-utenti;

- Accesso ai database: il backend è l'unico componente che ha accesso diretto ai database. Esegue query SQL su PostgresSQL per i dati relazionali e query temporali su InfluxDB per i dati time-series dei sensori, aggrega i risultati e restituisce dati strutturati al frontend;
- Servizi background: oltre a gestire richieste HTTP sincrone, il backend esegue processi asincroni in background, come il monitoraggio delle soglie personalizzate degli utenti, l'invio di notifiche quando tali soglie vengono superate, e la sincronizzazione periodica giornaliera della lista dei sensori.

Il backend implementa una pipeline di elaborazione delle richieste HTTP in middleware che processa sequenzialmente ogni richiesta, separando le responsabilità come logging, gestione CORS, autenticazione JWT e rate limiting delle richieste.

3.1.3 Database: Data Layer

L'architettura dell'applicazione utilizza due database specializzati per rispondere a due diverse esigenze: InfluxDB per i dati time-series ad alta frequenza, provenienti dai sensori sul campo, e PostgresSQL per dati relazionali strutturati.

La scelta dell'utilizzo di due database anziché uno solo è stata dettata dalle caratteristiche diverse dei dati gestisti dal sistema. I dati provenienti dai sensori IoT sono serie temporali: sequenze di valori numerici associati a timestamp precisi. Questi dati hanno pattern di scrittura ad alta frequenza (migliaia di punti al giorno) e pattern di lettura aggregati (medie orarie, trend giornalieri). Al contrario, i dati degli utenti e delle associazioni dei sensori sono relazionali, con pattern di lettura e scrittura bilanciati di cui è necessario garantirne l'integrità referenziale.

InfluxDB InfluxDB è un database open-source specializzato nella gestione di dati time-series, progettato per carichi di avoro IoT e monitoring. InfluxDB organizza i dati in una gerarchia a quattro livelli. Il livello più alto è il bucket, ovvero un contenitore logico per dati correlati. All'interno del bucket i dati sono organizzati in measurements, che rappresentano le tipologie di misurazioni raccolte dai sensori (temperatura, umidità, pressione, ecc.). Ogni punto è caratterizzato da un timestamp, da un insieme di tags (coppie chiave-valore per i metadati indicizzati come device_name o location) e da un insieme di fields (coppie chiave-valore per i valori numerici effettivi).

I measurements implementati all'interno del sistema TRACE sono otto:

- Temperature
- Humidity
- Pressure
- Rainfall
- Wind_speed
- Wind_direction
- Soil_moisture
- Soil_conductivity

InfluxDB utilizza Flux, un linguaggio di query funzionale e ottimizzato per la manipolazione di dati time-series. Flux permette operazioni complesse come il filtraggio temporale, aggregazione su finestre temporali, calcolo di derivate e integrali, e join tra measurements diversi. Il backend dell'applicazione costruisce dinamicamente query Flux in base ai parametri ricevuti dal frontend (sensori selezionati, range temporale, ecc.).

PostgresSQL PostgresSQL è un database relazionale open-source, utilizzato per la gestione di tutti i dati strutturati dell'applicazione che richiedono relazione complesse. Il database PostegresSQL implementa uno schema normalizzato con sette tabelle principali. Le tabelle sono collegate tra loro tramite chiavi esterne (foreign key) con vincoli di integrità referenziale per garantire la consistenza dei dati.

Il database implementa vincoli di CHECK per valori ammissibili (es. role IN ('admin','consorzio','user')), vincoli UNIQUE per garantire unicità (email, nome dei sensori), e vincoli di foreign key con azioni CASCADE per mantenere consistenza durante l'eliminazione di un record all'interno del database. Vengono inoltre implementati dei trigger su tutte le tabella per aggiornare automaticamente il campo updated_at ogni volta che un record viene modificato.

Figura 3.1: Diagramma di dominio

Tabella	Descrizione
users	Contiene l'anagrafica completa degli utenti del sistema: credenziali di autenticazione (email e password hash), ruolo (admin/consorzio/user), stato di approvazione, token FCM per le notifiche push e preferenze personalizzate come l'intervallo di auto-refresh e la modalità di campionamento. La password è salvata esclusivamente come hash BCrypt.
consorzi	Rappresenta i consorzi agricoli registrati nel sistema. Ogni consorzio ha un proprietario (owner_id), un nome descrittivo e uno stato di approvazione. I consorzi non approvati esistono nel database ma non possono effettuare il login finché l'admin non li abilita.
consorzio_members	Implementa la relazione many-to-many tra utenti e consorzi. Ogni richiesta di adesione crea un record con `is_approved = false`. Il responsabile del consorzio gestisce approvazioni e rifiuti.
sensori	Contiene tutti i dispositivi IoT registrati. Ogni sensore è identificato univocamente dal nome dispositivo e può essere assegnato a un consorzio tramite `consorzio_id`. I sensori con `consorzio_id = NULL` sono non allocati e visibili solo all'admin.
thresholds	Memorizza le soglie configurate dagli utenti, con tipo di misurazione, valore critico, operatore matematico, lista di sensori associati e intervalli di controllo/-notifica. Il campo `last_triggered_at` implementa un cooldown per evitare spam di notifiche. Il campo `use_average`, di default impostato come false, serve per decidere se utilizzare la media dei valori o controllarli singolarmente durante il controllo della soglia.
irrigation_notes	Storico delle irrigazioni registrate dagli utenti: quantità d'acqua (in mm), data/ora, e timestamp di creazione/modifica e campo di appunti facoltativo.

Tabella 3.1: Descrizione delle tabelle del database PostgreSQL

3.1.4 Servizio Cloud: Firebase Cloud Messaging

L'applicazione integra FCM, un servizio di Google per l'invio di notifiche push multipiattaforma, che permette la comunicazione asincrona dal backend verso i dispositivi mobili degli utenti anche quando l'app non è attiva.

Il sistema FCM segue un flusso di quattro fasi:

Registrazione del dispositivo quando un utente effettua il login per la prima volta su un dispositivo, l'applicazione richiede a Firebase un token FCM univoco per quel dispositivo. Questo token, una stringa alfanumerica, serve per identificare univocamente l'istanza dell'app su quello specifico dispositivo. Il token viene inviato al backend che lo salva nella tabella users associato all'utente che ha effettuato il login.

Trigger notifica le notifiche possono essere triggerate da due eventi distinti. Il primo è il superamento di una soglia personalizzata: il servizio di monitoring in background rileva che un valore del sensore ha superato una soglia, verifica che sia trascorso il periodo di cooldown dall'ultima notifica inviata controllando il campo notification_interval_minutes della soglia, e invia la notifica. Il secondo è una comunicazione broadcast da parte dell'admin a tutti gli utenti, sia standard che consorzi.

Invio notifica trame FCM il backend costruisce un paylod JSON che contiene titolo, messaggio, dati aggiuntivi (tipo soglia, valore rilevato, sensore, ecc.) e token FCM del destinatario. Firebase riceve la richiesta, valida il token e inoltra la notifica.

Ricezione e visualizzazione il sistema operativo del dispositivo riceve la notifica push e risveglia l'applicazione TRACE in background. L'app processa il paylod, estrae i dati e visualizza la notifica nel notification tray del dispositivo. Se l'utente clicca sulla notifica l'applicazione si apre.

3.2 Workflow e funzionalità dell'applicazione

Questa sezione illustra il flusso operativo dell'applicazione attraverso l'analisi dettagliata delle interfacce utente e delle funzionalità principali. Il workflow è organizzato seguendo il percorso naturale dell'utente, partendo dalla fase di autenticazione fino alla gestione delle soglie personalizzate.

3.2.1 Autenticazione

Il sistema di autenticazione gestisce tre ruoli: **User**, **Consorzio** e **Admin**.

User Registrazione: l'utente *User* si registra fornendo i propri dati personali (nome, cognome, email e password) e selezionando il consorzio al quale desidera iscriversi, tra la lista dei consorzi disponibili. Dovrà attendere di essere approvato all'interno del consorzio selezionato prima di poter eseguire il *login*.
Permessi e funzionalità: dopo l'approvazione, l'utente *User* può visualizzare esclusivamente i sensori associati dall'amministratore del consorzio di appartenenza.

Consorzio Registrazione: l'utente *Consorzio* si registra fornendo dati personali (nome, cognome, email e password) e indicando il nome del consorzio che si vuole registrare. L'utente *Consorzio* dovrà poi attendere di essere accettato dall'utente *Admin* prima di eseguire il *login* come amministratore del consorzio.
Permessi e funzionalità: una volta approvato, l'utente *Consorzio* riceve dall'*Admin* l'associazione dei sensori appartenenti al consorzio. L'amministratore del consorzio può (i) assegnare sensori agli utenti iscritti; (ii) accettare o rifiutare richieste di iscrizione; (iii) visualizzare i dati di tutti i sensori assegnati.

Admin Registrazione: la registrazione dell'*Admin* non è possibile dall'applicazione, ma viene eseguita durante il primo avvio del *backend*, con credenziali modificabili nel file `.env`.
Permessi e funzionalità: l'amministratore può (i) accettare o rifiutare le richieste di registrazione dei consorzi; (ii) inviare notifiche a tutti gli utenti; (iii) gestire l'associazione dei sensori ai consorzi; (iv) visualizzare i dati di tutti i sensori disponibili.

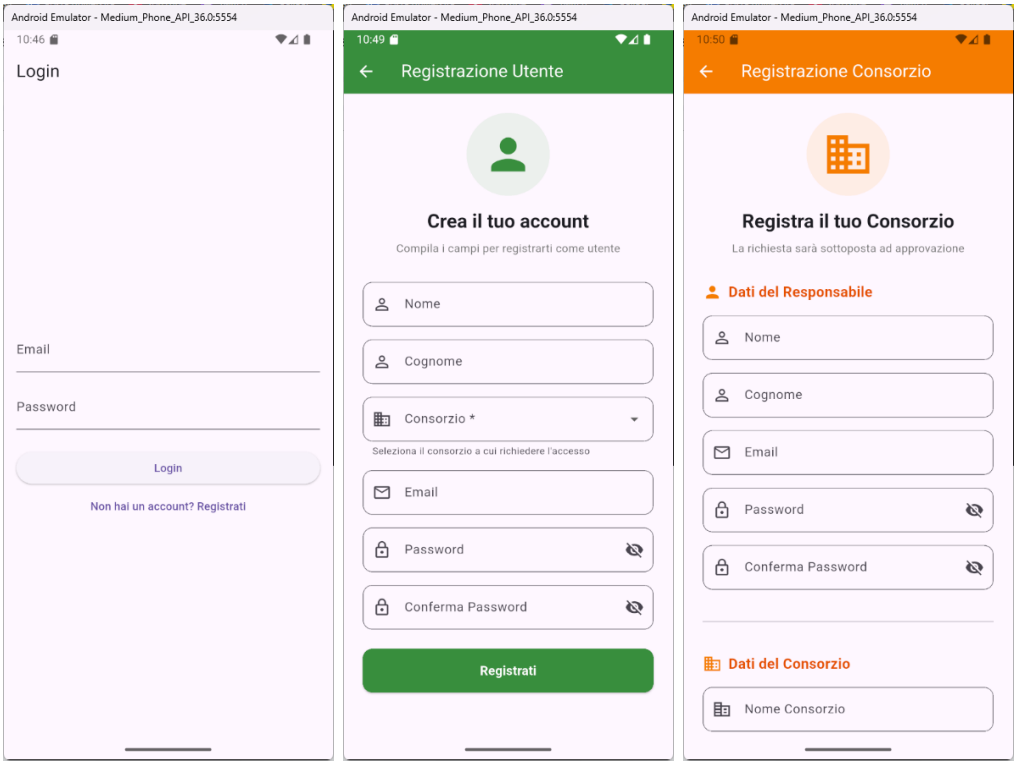


Figura 3.2: Da sinistra: *Login*, *Registrazione User*, *Registrazione Consorzio*

3.2.2 Interfaccia Dashboard

La *dashboard*, navigabile attraverso una *headbar*, permette la visualizzazione di sette viste principali:

- **Schermata Home**: vista principale dopo il login; permette l'aggiunta di note di irrigazione e la visualizzazione delle ultime tre create.

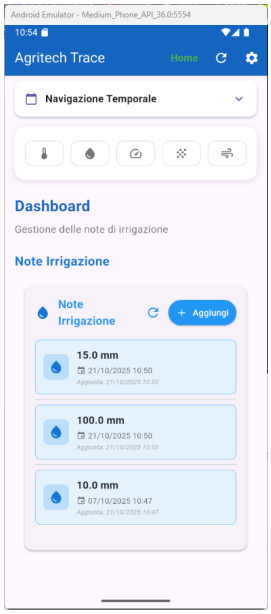


Figura 3.3: *Schermata Home*

- **Schermata Temperatura**: composta da quattro *collapsible menu* (Tabella, Grafico Completo, Grafico WS, Grafico EM) per visualizzare i dati dei sensori EM e WS.

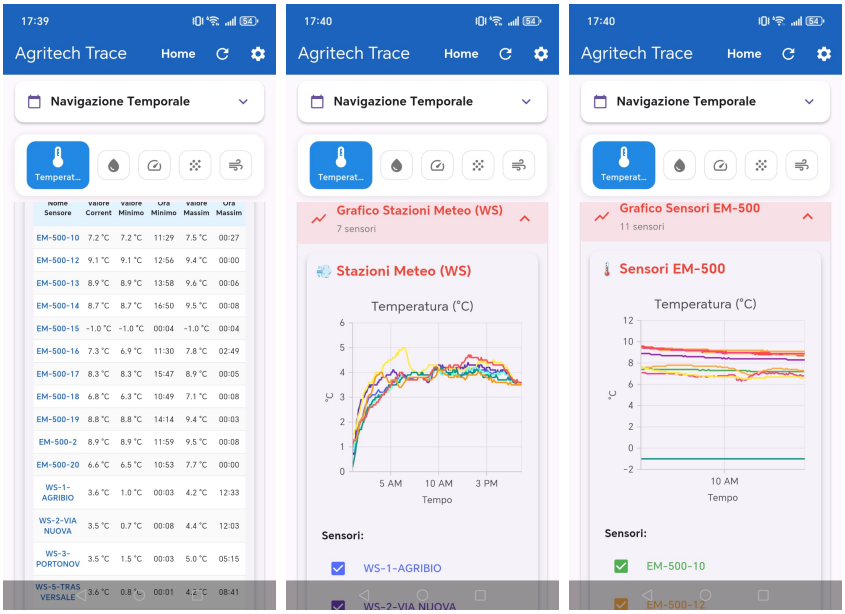


Figura 3.4: Da sinistra: *Tabella Temperatura*, *Grafico WS Temperatura*, *Grafico EM Temperatura*

- **Schermata Umidità**: suddivisa in tre sezioni (Umidità dell'aria, Umidità del suolo, Conducibilità del suolo), ciascuna con Tabella e Grafico EM/WS.

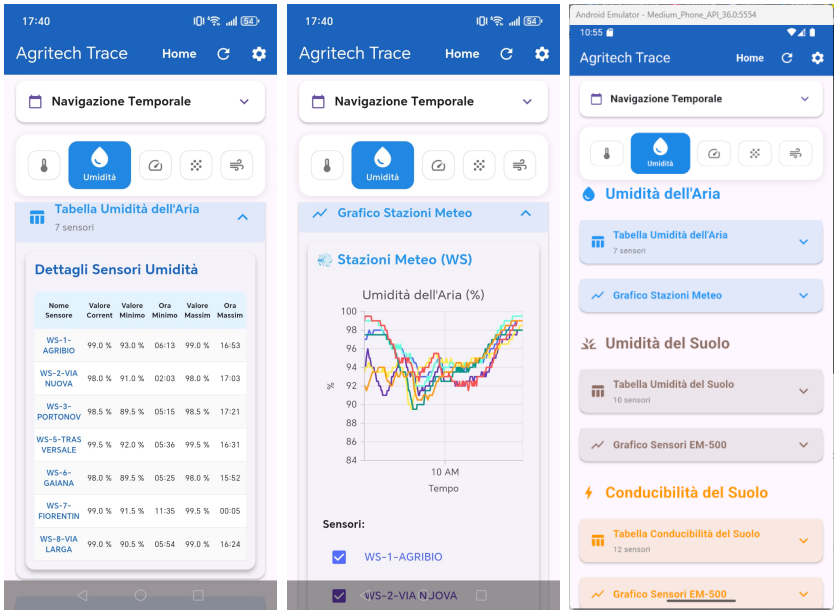


Figura 3.5: Da sinistra: *Tabella Umidità*, *Grafico Umidità*, *Schermata Umidità con menu chiusi*

- **Schermata Pressione**: due *collapsible menu* per visualizzare Tabella e Grafico WS della pressione.

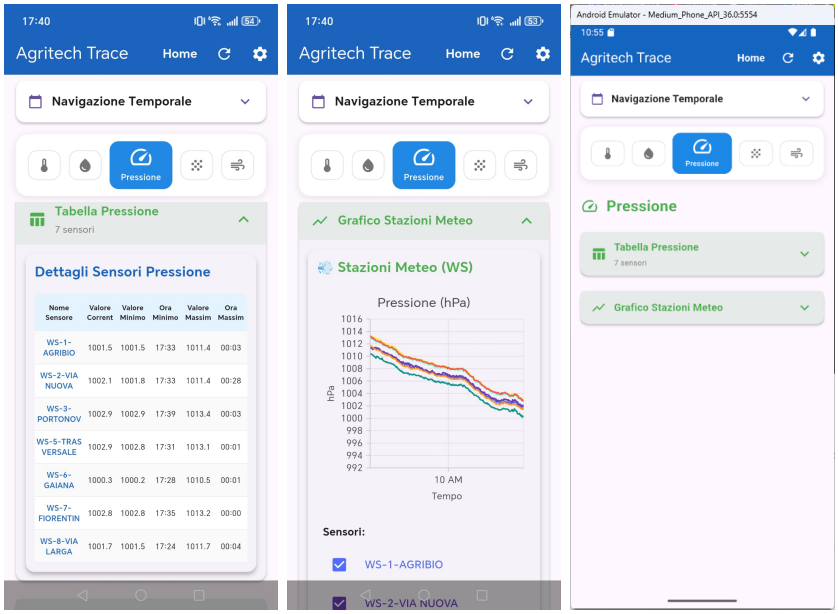


Figura 3.6: Da sinistra: *Tabella Pressione*, *Grafico Pressione*, *Schermata Pressione con menu chiusi*

- **Schermata Precipitazioni**: tre *collapsible menu* (Riepilogo precipitazioni, Grafico andamento, Tabella) per visualizzare totale, media e andamento delle precipitazioni.

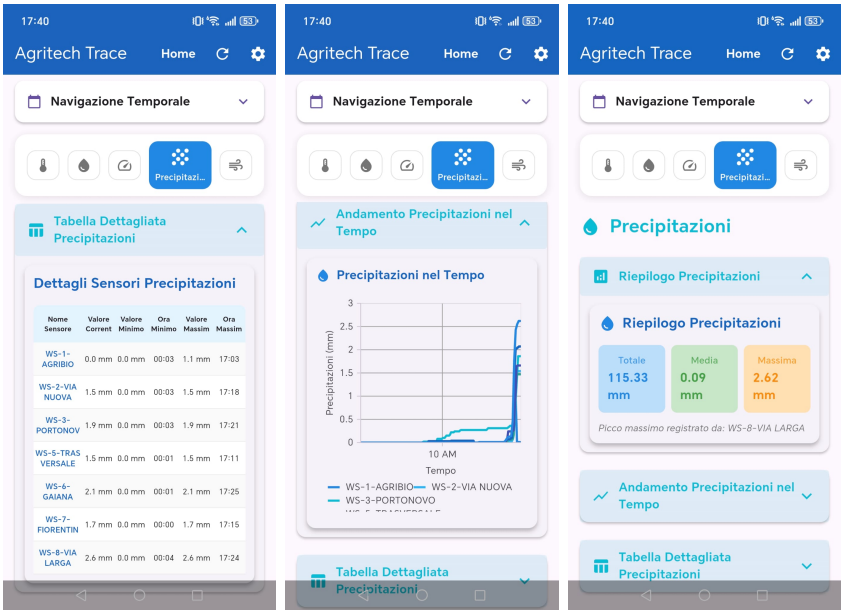


Figura 3.7: Da sinistra: *Tabella Precipitazioni*, *Grafico Precipitazioni*, *Riepilogo Precipitazioni*

- **Schermata Vento**: cinque *collapsible menu* per visualizzare Tabella e Grafico di velocità e direzione vento, oltre alla Rosa dei venti.

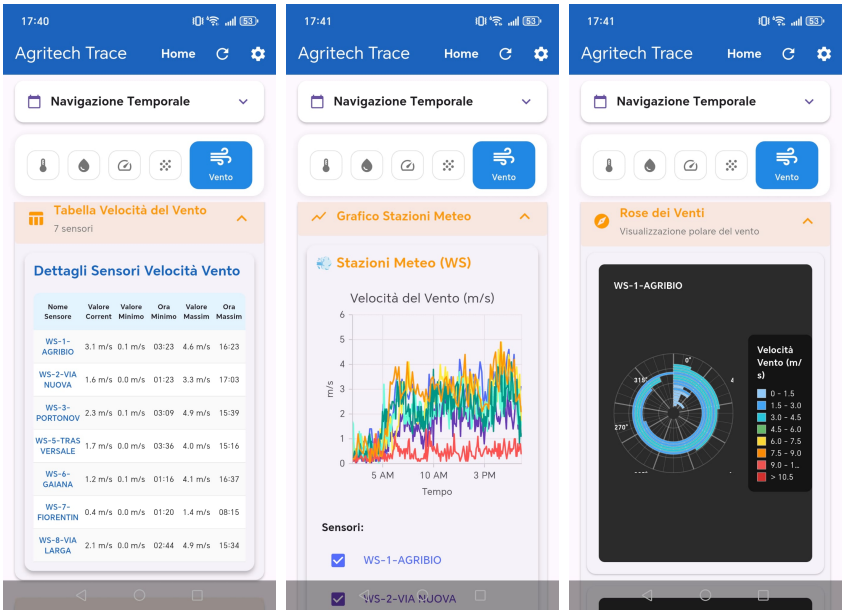


Figura 3.8: Da sinistra: *Tabella Velocità Vento*, *Grafico Velocità Vento*, *Rosa dei venti*

- **Schermata Impostazioni**: vista con opzioni avanzate variabili in base al ruolo dell'utente.

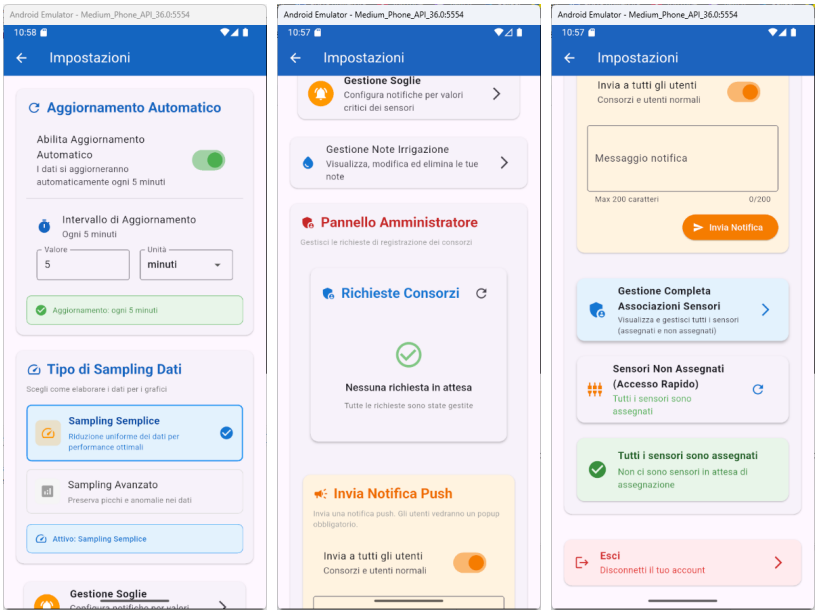


Figura 3.9: Da sinistra: *Schermata Impostazioni Consorzio*, *Schermata Impostazioni Admin pt1*, *Schermata Impostazioni Admin pt2*

3.2.3 Navigazione Temporale

L'applicazione implementa un *collapsible menu*, visibile in tutte le viste, che permette la visualizzazione dei dati di un range temporale diverso da quello di default (dalla mezzanotte del giorno corrente). Il menù presenta diverse opzioni di default, ma è possibile selezionare un range personalizzato, con un massimo di 90 giorni consecutivi selezionabili per evitare un sovraccarico di dati.

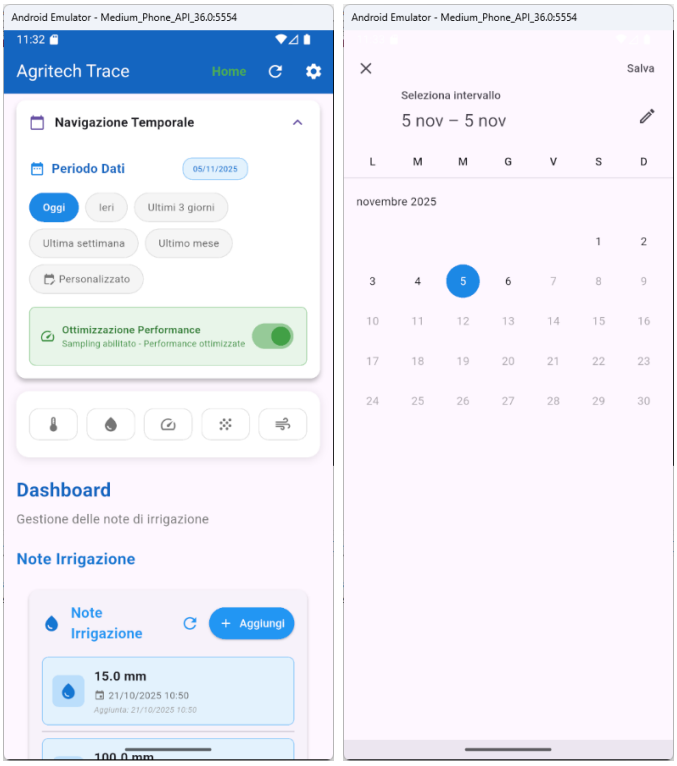


Figura 3.10: Da sinistra: *Menù navigazione temporale*, *Menù selezione range personalizzato*

3.2.4 Note di Irrigazione

L'applicazione implementa la possibilità di aggiungere note di irrigazione. È possibile creare delle nuove note dall'apposito menù accessibile dalla schermata delle impostazioni, dove si può modificare o eliminare note già esistenti, o dalla schermata Home tramite l'apposito bottone. Le note sono composte da data e ora di irrigazione e millimetri di acqua irrigata (con un massimo di 100 mm).

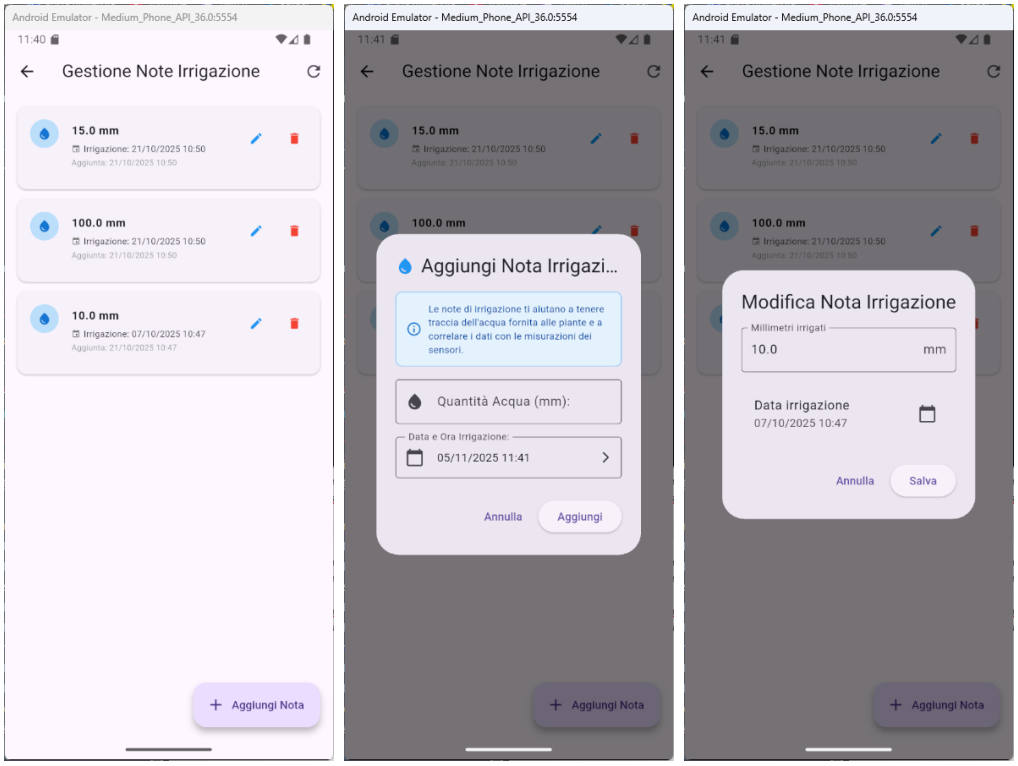


Figura 3.11: Da sinistra: *Schermata Note*, *Aggiunta Nuova Nota*, *Modifica Nota Esistente*

3.2.5 Soglie personalizzabili

L'applicazione permette di controllare e tenere sotto osservazione possibili valori critici prelevati dai sensori. Questo è possibile tramite la creazione di *thresholds* personalizzabili, tramite i quali l'utente può selezionare uno o più sensori target (se sono selezionati più sensori è possibile scegliere se controllare i singoli valori o controllare la media) e impostare un valore critico, un range di *check time* e uno di notifica.

In base al valore di *check* impostato, il *backend* effettua chiamate al database *InfluxDB* e controlla se il valore corrente del sensore o della media dei sensori risulta critico. Se il *backend* trova che una soglia è stata superata, viene mandata una notifica push all'utente, in base al *time range* di notifica selezionato durante la creazione della soglia.

La creazione delle soglie e la gestione di quelle già esistenti è possibile dall'apposita sezione all'interno delle impostazioni.

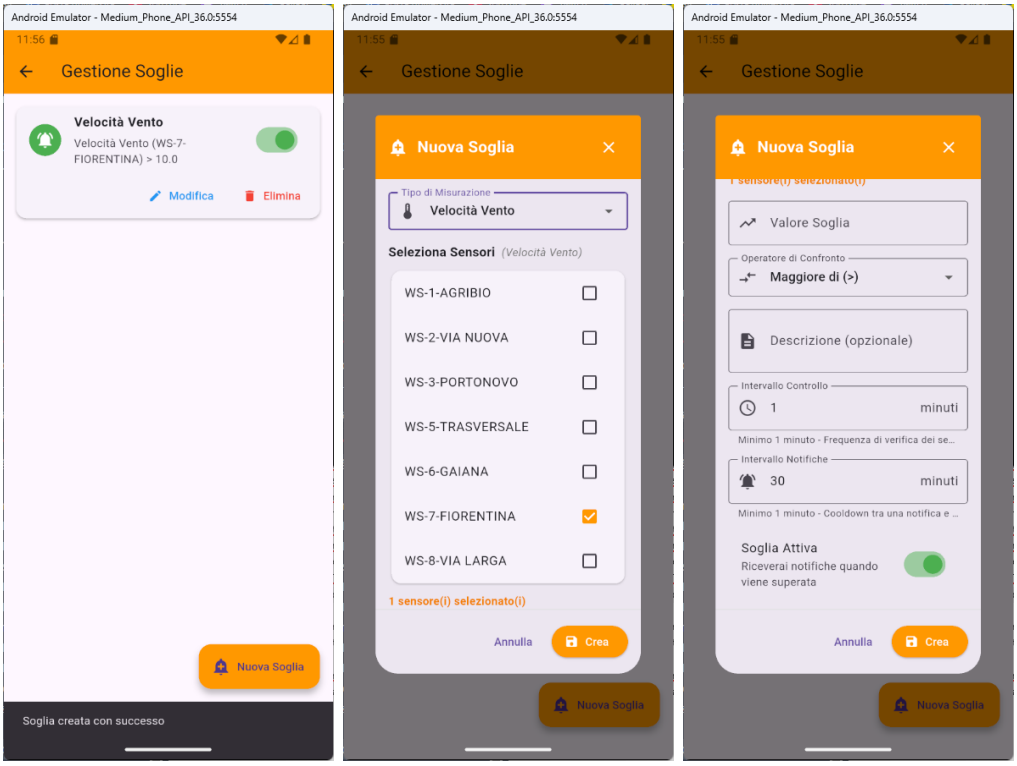


Figura 3.12: Da sinistra: *Gestione Soglie*, *Creazione nuova soglia (1)*, *Creazione nuova soglia (2)*

Capitolo 4

Implementazione

L'analisi dell'implementazione dell'applicazione procede dal backend verso il frontend, concentrandosi sui meccanismi chiave che permettono il corretto funzionamento del sistema: l'autenticazione JWT, il monitoraggio delle soglie, l'integrazione del servizio cloud FCM e gli algoritmi di campionamento.

4.1 Implementazione del backend

Il backend costituisce il cuore dell'applicazione, implementando tutta la logica di business, la gestione della sicurezza e i servizi in background che operano indipendentemente dalle richieste degli utenti. La scelta di Dart come linguaggio si è rivelata strategica per mantenere coerenza tecnologica con il frontend.

4.1.1 Framework Shelf e architettura modulare

Il backend utilizza Shelf, un framework web per Dart che implementa il pattern middleware/handler. A differenza dei framework come Express.js o Spring Boot, Shelf adotta un sistema dove ogni funzionalità è un middleware indipendente che può essere assemblato in una pipeline.

La struttura del progetto riflette questa filosofia modulare:

```
backend/
|- bin/
|   |- server.dart              # Entry point principale
```

```
|   |- init_admin.dart          # Script inizializzazione admin
|   |- healthcheck.dart         # Health check per container orchestration
|- lib/
|   |- middleware/              # Componenti pipeline HTTP
|   |   |- auth_middleware.dart
|   |   |- logging_middleware.dart
|   |   |- rate_limit_middleware.dart
|   |- routes/                  # Handler endpoint REST
|   |   |- auth_routes.dart
|   |   |- influx_routes.dart
|   |   |- threshold_routes.dart
|   |   |- sensor_routes.dart
|   |   |- irrigation_note_routes.dart
|   |   |- notification_routes.dart
|   |- services/                # Logica business e servizi background
|   |   |- jwt_service.dart
|   |   |- sql_service.dart
|   |   |- influxdb_service.dart
|   |   |- threshold_monitoring_service.dart
|   |   |- fcm_service.dart
|   |   |- sensor_sync_service.dart
|   |- models/                  # Rappresentazioni dati
|       |- user.dart
|       |- threshold.dart
|       |- sensor.dart
|       |- irrigation_note.dart
|- Dockerfile                   # Containerizzazione multi-stage
|- pubspec.yaml                 # Dependency management
```

Questa organizzazione permette di mantenere un'alta coesione (ogni modulo ha una responsabilità ben definita) e basso l'accoppiamento (i moduli comunicano attraverso delle interfacce chiare).

4.1.2 Pipeline middleware e gestione richieste HTTP

Quando arriva una richiesta HTTP al server, questa attraversa una pipeline di middleware che implementano cross-cutting-concerns (preoccupazioni trasversali che riguardono tutte le richieste). La pipeline viene costruita all'interno di server.dart seguendo un ordine preciso:

- CORS Middleware: gestisce le politiche di Cross-Origin Resource Sharing, indispensabile per permettere al frontend di comunicare con il backend anche quando eseguito su domini o porte differenti.

- Logging Middleware: registra ogni richiesta con timestamp, metodo HTTP, endpoint, IP sorgente, durata dell'elaborazione e status code della risposta. I log vengono scritti sulla console.

- Rate Limiting Middleware: implementa la protezione contro attacchi DDoS e abuso dell'API limitando il numero di richieste per IP/Token in una finestra temporale (di default impostato a 200 richieste ogni 15 minuti). Utilizza una mappa con timestamp con scadenza automatica.

- Error Handler Middleware: cattura tutte le eccezioni non gestite dai middleware successivi e all'interno degli handler, restituendo risposte JSON strutturate. Previene crash del server.

- Auth Middleware: valida i token JWT per gli endpoint protetti. Opera in due modalità: requireAuth(), che blocca le richieste che non possiedono un token valido, e optionalAuth(), che processa il token se presente ma permette l'accesso anche senza. Dopo la validazione estrae dal paylod le informazioni utente e le inserisce all'interno del contesto della richiesta, rendendole accessibili agli handler successivi senza dover ricodificare il token.

- Router: direziona le richieste agli handler specifici in base al metodo HTTP e al percorso URL. Utilizza shelf_router che supporta i path parameters (/users/:id) e query parameters. Il router è organizzato in una struttura gerarchica con mount points per ogni dominio funzionale (/api/v1/auth, /api/v1/influx, /api/v1/thresholds, ecc.)

Questa architettura a pipeline garantisce separation of concerns, testabilità (ogni middleware può essere testato isolatamente) e flessibilità (possono essere aggiunti nuovi middleware senza dover modificare il codice già esistente).

4.1.3 Sistema di autenticazione JWT

Il sistema di autenticazione è implementato con JSON Web Tokens, una modalità standard per l'autenticazione stateless in architetture distribuite. Il sistema genera due tipi di token:

Access Token : token a breve scadenza (1 ora) che l'utente include all'interno degli header Authorization: Bearer <token> di ogni richiesta API protetta. All'interno del paylod JSON contiene:

- sub (subject): id univoco dell'utente
- role: ruolo per RBAC (per determinare i permessi)
- email, firstName, lastName: informazioni dell'utente
- iat (issued at): timestamp di emissione del token
- exp (expiration): timestamp di scadenza del token
- type: "access", per distinguerlo dal refresh token

Il token è firmato utilizzando una chiave segreta condivisa. La firma permette e garantisce integrità (il paylod non può essere modificato senza invalidare la firma) e autenticità (solo chi possiede la chiave segreta può generare dei token validi)

Refresh Token : token a lunga scadenza (7 giorni) usato per ottenere nuovi access token senza dover richiedere nuovamente le credenziali. Contiene paylod minimale (sub, email, type:"refresh"). Quando l'access token scade, il frontend può fare richiesta all'endpoint /api/v1/auth/refresh inviando il refresh token e ottenere una nuova coppia di token.

Questo schema a doppio token serve per bilanciare la sicurezza e l'usabilità dell'applicazione: access token brevi limitano la finestra di vulnerabilità se rubati; refresh token lunghi evitano login ripetuti per migliorare l'esperienza utente; in caso di compromissione, l'invalidazione dei refresh token forza ad effettuare nuovamente il login globale.

La validazione del token all'interno del AuthMiddleware esegue tre controlli: (i) verifica la firma, (ii) verifica la scadenza controllando che il timestamp corrente sia minore dell'exp del paylod, (iii) parsa il paylod decodificandolo in Base64URL.

Se la validazione fallisce (firma invalida o token scaduto), la richiesta viene respinta con 401 Unauthorized. Se ha successo tutte le informazioni precedentemente elencate vengono iniettate nel context della richiesta e l'handler può accedervi tramite request.context['userId'] e request.context['role'].

4.1.4 Integrazione InfluxDB e query Flux

Come spiegato nel capitolo precedente, InfluxDB è un database specializzato per i dati time-series, ottimizzato per query temporali complesse. Il backend funge da proxy intelligente tra il frontend e InfluxDB, implementando due responabilità critiche:

1. Security Layer : InfluxDB non possiede un sistema di autenticazione granulare per gli utenti. Il backend traduce l'autenticazione JWT in query Flux che filtrano i dati in base ai sensori autorizzati per l'utente che sta effettuando la richiesta. Questo garantisce che utenti non autorizzati non possano accedere a dati di sensori di altri consorzi anche se riuscissero a bypassare il frontend.

2. Data Trasformation : InfluxDN restituisce dati in formato CSV annotato. Il backend parsa questo file, lo converte in formato JSON leggibile dal frontend e arricchisce i dati (aggiungendo per esempio: nome del sensore, unità di misura, statistiche aggregate, ecc.).

Le query Flux utilizzano un modello funzionale a pipeline. Esempio di query:

Codice 4.1: Esempio query Flux

```
from(bucket: "trace_project")
  |>range(start: 2025-11-24T00:00:00Z, stop: 2025-11-25T14:30:00Z)
  |>filter(fn: (r) => r._measurement == "
     device_frmpayload_data_temperature" and r._field == "value")
  |>filter(fn: (r) => contains(value: r.device_name, set: ["EM-500-1
     ", "EM-500-2", "EM-500-3"]))
  |>filter(fn: (r) => r._value > -40.0 and r._value < 80.0 and r.
     _value != -0.100)
  |>keep(columns: ["_time", "_value", "device_name"])
```

Analisi della pipeline della query Flux:

1. from(bucket: "trace_project"): seleziona il bucket InfluxDB contenente i dati. È l'equivalente di un database nei sistemi SQL

2. range(start:..., stop: ...): filtra temporalmente i dati. I timestamp sono in formato UTC, vengono convertiti dal backend nella timezone italiana corrente. Il backend costruisce questi valori dinamicamente, convertendo i dati temporali ricevuti dal frontend.

3. filter(fn: (r) => r._measurement == "device_frmpayload_data_temperature" and r._field == "value"): doppio filtro che seleziona:

   - _measurement: equivale al nome della tabella in SQL. InfluxDb organizza i dati per tipologia di misurazione (temperature, humidity, pressure, ecc.)
   - _field == "value": ogni measurement può avere diversi field (es: value, quality, status) e nella richiesta viene selezionato quello che si vuole ricevere, in questo caso solo il valore numero effettivo.

4. filter(fn: (r) => contains(value: r.device_name, set: [...])): rappresenta il filtro di sicurezza critico. La funzionalità contains() verifica che il device_name sia presente nella lista dei sensori autorizzati per l'utente. Il backend costruisce questo array in modo dinamico, interrogando PostgresSQL. Questo impedisce data leakage.

5. filter(fn: (r) => r._value > -40.0 and r._value < 80.0 and r._value != -0.100): rappresenta il filtro per la qualità dei dati. Elimina valori outlier non plausibili: temperature sotto i -40°C o sopra i 80°C. Il valore -0.1°C è un error code documentato dal produttore che indica un malfunzionamento temporaneo del sensore. Questi filtri prevengono che errori di lettura o malfunzionamenti corrompano le statistiche aggregate (media, min, max).

6. keep(columns: ["_time", "_value", "device_name"]): proiezione finale che mantiene solo le colonne necessarie, riducendo il peso e la dimensione del CSV restituito. InfluxDB di default include metadati inutili al frontend.

Risultato della query in CSV annotato:

Codice 4.2: Esempio risultato CSV annotato delle query Flux

```
#group,false,false,true,true
```

```
#datatype,string,long,dateTime:RFC3339,string,double
#default,_result,,,
,result,table,_time,device_name,_value
,_result,0,2025-11-24T06:15:00.000Z,EM-500-1,18.5
,_result,0,2025-11-24T06:20:00.000Z,EM-500-1,18.7
,_result,1,2025-11-24T06:15:00.000Z,EM-500-2,18.2
,_result,1,2025-11-24T06:20:00.000Z,EM-500-2,18.4
```

Le prime righe (quelle con #) contengono i metadati del formato. Il backend parsa le righe dei dati, converte i timestamp da UTC alla timezone italiana e trasforma il CSV in array JSON utilizzati poi dal frontend per la costruzione delle tabelle e dei grafici:

Codice 4.3: Esempio JSON creato dal parsing del CSV

```
[
  {"time": "2025-11-24T08:15:00+02:00", "device": "EM-500-1", "value
      ": 18.5},
  {"time": "2025-11-24T08:20:00+02:00", "device": "EM-500-1", "value
      ": 18.7},
  {"time": "2025-11-24T08:15:00+02:00", "device": "EM-500-2", "value
      ": 18.2},
  {"time": "2025-11-24T08:20:00+02:00", "device": "EM-500-2", "value
      ": 18.4}
]
```

4.1.5 Servizio di monitoraggio delle soglie

Il ThresholdMonitoringService è uno dei componenti più critici del backend e implenta un sistema di alerting che opera in background 24/7 indipendentemente dall'interazione utente. Il servizio si avvia in automatico all'inizializzazione del server e continua a funzionare fino all'arresto.

Architettura del servizio : il servizio utilizza un Timer.periodic di Dart per eseguire controlli a cadenza regolare (1 minuto). A ogni tick del timer, il servizio:

1. Recupera le soglie attive: esegue una query PostgresSQL che seleziona tutte le soglie con valore is_active = true all'interno della tabella thresholds;

2. Filtra per l'intervallo di controllo: per ogni soglia verifica se è trascorso il tempo minimo necessario dall'ultimo controllo, confrontando il *timestamp* corrente con il parametro `lastTriggeredAt + checkIntervalMinutes`;

3. Query InfluxDB: per ogni soglia da controllare esegue una query Flux per ottenere gli ultimi valori dei sensori specificati nella soglia in questione;

4. Calcolo dei valori da confrontare: se la soglia ha come attributo use_average = true, calcola la media dei valori ricevuti; altrimenti utilizza i singoli valori;

5. Valutazione della condizione: confronta il valore calcolato nel punto precedente con il threshold_value usando l'operatore specificato nella soglia;

6. Invio della notifica: se la condizione è soddisfatta, il servizio verifica se è trascorso abbastanza tempo dall'ultima notifica confrontando `lastTriggeredAt + notificationIntervalMinutes` con il timestamp corrente. Se il *cooldown* è scaduto, il servizio invia la notifica; altrimenti viene soppressa per evitare spam;

7. Aggiornamento dei timestamp: aggiorna i timestamp last_triggered_at e notification_interval_minute con i valori aggiornati.

Gestione degli intervalli Il sistema implementa due intervalli indipendenti per ottimizzare le risorse e l'esperienza utente:

- check_interval_minutes: determina ogni quanto interrogare il database InfluxDB. Questo valore è personalizzabile durante la creazione della soglia ed è necessario perché ogni tipologia di dato (temperatura, umidità, pioggia, ecc.) ha valori critici differenti e richiede un diverso livello di rapidità nella rilevazione e nella risposta agli eventuali superamenti della soglia.

- notification_interval_minutes: implementa il cooldown tra notifiche successive di una stessa soglia. È necessario per evitare lo spam di

notifiche, permettendo però al sistema di tenere traccia comunque della condizione in modo regolare.

Integrazione FCM L'invio delle notifiche push utilizza FCM V1 API con sistema di autenticazione OAuth2. Il processo di invio si divide in quattro fasi: (i) il backend carica il file firebase-service-account.json, che contiene le credenziali del service account Google; (ii) genera un OAuth2 access token temporaneo usando le credenziali del service account; (iii) recupera dalla tabella users il fcm_device_token dell'utente target; (iv) costruisce il paylod JSON della notifica; (v) invia una richiesta POST HTTPS a https://fcm.googleapis.com/v1/projects/project_id/messages:send con l'access token all'interno dell'header Authorization: Bearer.

Esempio di paylod JSON per l'invio della notifica:

Codice 4.4: Esempio paylod JSON per l'invio di una notifica

```
{
  "message": {
    "token": "<fcm_device_token_utente>",
    "notification": {
      "title": "Soglia Superata: Temperatura",
      "body": "Temperatura media 35.2C supera soglia di 35C"
    },
    "data": {
      "threshold_id": "123",
      "measurement_type": "temperature",
      "current_value": "35.2"
    }
  }
}
```

La sezione notification crea l'alert visivo che comparirà sul dispositivo, mentre data contiene il paylod personalizzato che sarà processato dall'applicazione (es: aprire direttamente il grafico della temperatura quando l'utente clicca sulla notifica).

4.1.6 Servizio di sincronizzazione dei sensori

Il SensorSyncService risolve il problema di coerenza dei dati: i sensori IoT sono configurati in InfluxDB (che riceve i dati) ma devono essere registrati anche in PostgresSQL per l'assegnazione ai consorzi. Senza sincronizzazione automatica, i nuovi sensori dovrebbero essere aggiunti manualmente dall'admin o risulterebbero non esistenti al sistema.

Il servizio esegue una sincronizzazione periodica, ogni 24 ore, con il seguente workflow:

1. Query InfluxDB: esegue una query FLux per estrarre l'elenco di tutti i device_name unici presenti all'interno del bucket;
2. Query PostgreSQL: esegue una query SQL per estrarre tutti i nome_dispositivo presenti all'interno della tabella sensori;
3. Controllo differenze: calcola la differenza insiemistica confrontando i sensori presenti in InfuxDB ma assenti in PostgreSQL;
4. Inserimento batch: inserisce i nuovi sensori in PostgreSQL con parametro consorzio_id = NULL (non assegnato ancora a nessun consorzio);
5. Logging: registra il numero di sensori sincronizzati.

4.1.7 Routing e API REST

Il backend esone un'API RESTful:

Endpoint di Autenticazione (/api/v1/auth)

Tabella 4.1: Endpoint di Autenticazione

Metodo	Endpoint	Descrizione
POST	/login	Autentica l'utente e restituisce la coppia di token (access + refresh).
POST	/register	Registra un nuovo utente (user o consorzio).
POST	/refresh	Rinnova l'access token usando il refresh token.
GET	/me	Ottiene il profilo dell'utente corrente (protetto).
PUT	/update-fcm-token	Aggiorna il token FCM del dispositivo (protetto).

Endpoint dati InfluxDB (/api/v1/influx) – Protetti

Tabella 4.2: Endpoint dati InfluxDB

Metodo	Endpoint	Descrizione
POST	/temperature/table	Ottiene dati di temperatura per range temporale.
POST	/air-humidity/table	Ottiene dati di umidità dell'aria.
POST	/pressure/table	Ottiene dati di pressione atmosferica.
POST	/rainfall/table	Ottiene dati di precipitazioni.
POST	/soil-moisture/table	Ottiene dati di umidità del suolo.
POST	/wind-speed/table	Ottiene dati di velocità del vento.
POST	/wind-direction/table	Ottiene dati di direzione del vento.

Endpoint soglie personalizzate (/api/v1/thresholds) – Protetti

Tabella 4.3: Endpoint soglie personalizzate

Metodo	Endpoint	Descrizione
GET	/	Lista tutte le soglie dell'utente corrente.
GET	/active	Lista solo le soglie attive.
POST	/	Crea una nuova soglia.
PUT	/:id	Aggiorna una soglia esistente.
DELETE	/:id	Elimina una soglia.
POST	/:id/toggle	Attiva o disattiva rapidamente una soglia.

Endpoint sensori (/api/v1/sensori) – Protetti

Tabella 4.4: Endpoint sensori

Metodo	Endpoint	Descrizione
GET	/available	Lista i sensori disponibili per l'utente corrente.
GET	/all	Lista tutti i sensori (solo admin).
POST	/:id/assign	Assegna un sensore a un consorzio (solo admin).
POST	/sync	Forza la sincronizzazione da InfluxDB (solo admin).

Endpoint note di irrigazione (/api/v1/irrigation-notes) – Protetti

Tabella 4.5: Endpoint note di irrigazione

Metodo	Endpoint	Descrizione
GET	/	Lista tutte le note dell'utente.
GET	/range	Filtra le note in base a un range temporale.
POST	/	Crea una nuova nota.
PUT	/:id	Aggiorna una nota esistente.
DELETE	/:id	Elimina una nota.

Endpoint notifiche (/api/v1/influx) – Solo admin

Tabella 4.6: Endpoint notifiche

Metodo	Endpoint	Descrizione
POST	/broadcast	Invia una notifica push a tutti gli utenti.
POST	/send	Invia una notifica push a un utente specifico.

Le risposte seguono il formato JSON con gestione degli errori standardizzata. Errori 4xx indicano problemi dal lato client e errori 5xx dal lato server.

4.2 Implementazione del frontend

4.2.1 Architettura e gestione dello stato

Flutter utilizza il pattern Widget Tree dove ogni elemento dell'interfaccia è un widget immutabile. L'applicazione è basata principalmente su StatefulWidget per schermate con stato mutabile (schermate dei grafici, schermata delle impostazioni) e StatelessWidget per i componenti puri (card, dialog). La gestione dello stato avviene in modo ibrido:

- Stato locale: i widget con State object preservano lo stato locale (es: _isLoading, _selectedDateRange). Quando lo stato cambia, il metodo setState() chiama la ricostruzione del widget.

- Stato globale attraverso Singleton Services: i dati condivisi all'interno di più schermate (token di autenticazione, preferenze dell'utente) sono gestiti tramite servizi Singleton che permettono ai dati di persistere oltre il lifecycle dei widget. AuthService,SamplingPreferenceService, AutoRefreshService sono tutte istanze globali accessibili da ovunque all'interno dell'app.

- Persistenza con SharedPreferences: tutti i dati che devono sopravvivere alla chiusura o al restart dell'applicazione (token JWT, ruolo utente, preferenze) vengono salvati all'interno di SharedPreferences. Lettura asincrona all'avvio e scrittura asincrona ad ogni modifica apportata.

4.2.2 Comunicazione con il backend

BackendService è l'unico punto di accesso per tutte le chiamata HTTP. Questa centralizzazione ha diversi vantaggi.

Gestione dei token automatica : ogni richiesta effettuata verso un endpoint protetto include automaticamente il token JWT nell'header, evitando la duplicazione in ogni schermata.

Timeout differenziati : le chiamate agli endpoint di InfluxDB, che utilizzano query pesanti, hanno un timeout maggiore rispetto alle chiamate standard.

Retry logic : tutti gli errori transienti (problemi temporanei causati da condizioni ambientali o momentanee che si risolvono da soli), come timeout network, 502/503 temporary unavailable, attivano un processo di retry automatico, con exponential backoff (strategia di gestione degli errori in informatica in cui un client ritenta una richiesta non riuscita dopo un intervallo di tempo che aumenta esponenzialmente con ogni tentativo successivo) prima di mostrare l'errore all'utente.

Error handling centralizzato : gestione di tutte le risposte HTTP provenienti dal backend. Le eccezioni principali e più frequenti vengono trasformate in formato user-friendly e più leggibile dall'utente.

4.2.3 Ottimizzazione di performance: algoritmi di sampling

Il rendering di grafici con migliaia di punti su dispositivi mobili potrebbe causare diversi problemi, i principali sono:

- Frame drop e lag dell'interfaccia durante l'utilizzo dell'applicazione;
- Sovraccarico della memoria che potrebbe causare crash dell'app su alcuni dispositivi:
- Trasferimento di dati eccessivo su connessioni cellulari.

Gli algoritmi di campionamento implementati all'interno dell'applicazione risolvono questi problemi, riducendo le dimensioni del dataset visualizzato mantenendo comunque alta la qualità dei dati. All'interno del progetto sono stati implementati due algoritmi di sampling:

Campionamento uniforme : approccio basico che, dato un dataset con N punti e un target di T punti da ottenere, calcola degli step = N/T e seleziona un punto ogni step posizioni. Per mantenere l'intero range temporale, il primo e l'ultimo punto del dataset vengono sempre inclusi. Questo algoritmo garantisce una distribuzione temporale uniforme. Se il dataset contiene i dati delle ultime 72 ore, i punti campionati saranno uniformemente distribuiti all'interno di questa finestra temporale. Lo svantaggio principale di questo algoritmo è la possibilità di perdita di eventi critici se posizionati tra due step, come uno sbalzo di temperatura per un periodo di tempo molto breve. Questo algoritmo ha complessità O(T).

Codice 4.5: Algoritmo di sampling uniforme

```
// **** Sampling uniforme semplice ****
static List<SensorData> _sampleUniform(List<SensorData> data, int
    targetPoints) {
  final step = data.length / targetPoints;
  final sampledData = <SensorData>[];

  // Mantieni sempre il primo punto
  sampledData.add(data.first);

  // Campiona uniformemente
  for (int i = 1; i < targetPoints - 1; i++) {
    final index = (i * step).round();
    if (index < data.length) {
      sampledData.add(data[index]);
    }
  }

  // Mantieni sempre l'ultimo punto
  if (data.length > 1) {
    sampledData.add(data.last);
  }

  return sampledData;
}
```

Campionamento con preservazione dei picchi : algoritmo sofisticato che assegna degli score di importanza ad ogni punto basato sulla curvatura locale (seconda derivata discreta). I punti dove la pendenza della curva cambia bruscamente ricevono uno score alto. Il processo di questo algoritmo è il seguente:

1. Per ogni punto i, estremi esclusi, viene calcolato lo slope prima del punto(pendenza, varazione tra due punti consecutivi). slope_prima=value[i]-value[i-1]

2. Per ogni punto i, estremi esclusi, viene calcolato slope dopo del punto. slope_dopo=value[i+1]-value[i];

3. Per ogni punto i, estremi esclusi, viene calcolato lo score di curvatura in valore assoluto. score[i]= | slope_dopo - slope_prima |;

4. Assegna score infinito al primo e all'ultimo punto del dataset, per mantenerli sempre;

5. Ordina i punti per score decrescente;

6. Seleziona i migliori T punti in base allo score;

7. Riordina i punti cronologicamente per mantenere la sequenza temporale.

Questo algoritmo ha complessità O(N log(N)), è più costoso del sampling uniforme ma permette di preservare picchi e anomalie anche con una riduzione del numero di punti elevata. È adatto per dati con variabilità alta e per eventi rari critici (allerte meteo).

Il sampling si attiva solo quando il numero dei punti ricevuti da InfluxDB è maggiore di 3000 e il numero di punti target è fissato a 500 per garantire un rendering fluido su tutti i dispostivi. L'utente può selezionare quale algoritmo di sampling utilizzare tramite la schermata delle impostazioni e la preferenza viene salvata in SharedPreferences e applicata globalmente a tutti i grafici. È possibile anche disabilitare il sampling tramite l'apposito bottone all'interno del menu collapsible di navigazione temporale.

Codice 4.6: Algoritmo di sampling con preservazione dei picchi

```
/// **** Sampling avanzato con preservazione dei picchi ****
static List<SensorData> _sampleWithPeakPreservation(List<SensorData
    > data, int targetPoints) {
  // Non campionare se sotto la soglia minima (3000 punti)
  if (data.length <= minPointsForSampling) return data;

  // Non campionare se gia sotto il target
  if (data.length <= targetPoints) return data;

  // Ordina per timestamp
  final sortedData = List<SensorData>.from(data);
```

```
  sortedData.sort((a, b) =>a.timestamp.compareTo(b.timestamp));

  // Calcola importanza di ogni punto
  final importanceScores = _calculateImportanceScores(sortedData);

  // Crea lista di punti con i loro score di importanza
  final pointsWithScores = <MapEntry<SensorData, double>>[];
  for (int i = 0; i < sortedData.length; i++) {
    pointsWithScores.add(MapEntry(sortedData[i], importanceScores[i]
        ));
  }

  // Ordina per importanza (maggiore = piu importante)
  pointsWithScores.sort((a, b) =>b.value.compareTo(a.value));

  // Prendi i punti piu importanti
  final selectedPoints = pointsWithScores
      .take(targetPoints)
      .map((entry) =>entry.key)
      .toList();

  // Riordina per timestamp
  selectedPoints.sort((a, b) =>a.timestamp.compareTo(b.timestamp));

  return selectedPoints;
}

/// **** Calcola score di importanza per ogni punto ****
static List<double> _calculateImportanceScores(List<SensorData>
    data) {
  final scores = List<double>.filled(data.length, 0.0);

  // Primi e ultimi punti sono sempre importanti
```

```
  if (data.isNotEmpty) {
    scores[0] = 100.0;
    if (data.length > 1) {
      scores[data.length - 1] = 100.0;
    }
  }

  // Calcola importanza basata su variazione locale
  for (int i = 1; i < data.length - 1; i++) {
    final prev = data[i - 1];
    final current = data[i];
    final next = data[i + 1];

    // Calcola variazione (derivata seconda approssimata - curvatura
        )
    final leftSlope = current.value - prev.value;
    final rightSlope = next.value - current.value;
    final curvature = (rightSlope - leftSlope).abs();

    // Calcola distanza temporale (punti piu distanziati sono piu
        importanti)
    final timeSpan = next.timestamp.difference(prev.timestamp).
        inMilliseconds;
    final timeWeight = timeSpan / 1000.0;

    // Score finale combina curvatura e peso temporale
    scores[i] = curvature * 10 + timeWeight * 0.1;
  }

  return scores;
}
```

4.2.4 Auto-Refresh intelligente

Il sistema di aggiornamento automatico implementa una logica context-aware. Un timer periodico, configurabile attraverso la schermata delle impostazioni, ricarica i dati, verificando però prima il contesto temporale:

- Se l'utente sta visualizzando i dati nel range temporale di default (dati a partire dalla mezzanotte del giorno corrente) il refresh avviene in automatico;
- Se l'utente sta visualizzando i dati di un range temporale diverso da quello di default viene mostrato un dialog. Questo pop-up avvisa l'utente che in quel momento dovrebbe avvenire un refresh dei dati ma questo comporterebbe il reset del range temporale in visualizzazione. L'utente, tramite due bottoni, può decidere se procedere con il refresh dei dati o saltare questo ciclo e aspettare il prossimo.

4.2.5 Visualizzazione dei dati con fl_chart e widget personalizzati

I grafici all'interno dell'applicazione utilizzano principalmente la libreria fl_chart per il rendering, integrata con widget personalizzati dove necessario. Questa libreria permette di creare grafici LineChart multi-serie (una linea per sensore) per dati come temperatura, umidità e pressione e BarChart con aggregazone automatica oraria e giornaliera per i dati relativi alle precipitazioni.

Tuttavia la libreria non fornisce supporto nativo per diagrammi polari e quindi è stato implementato un widget personalizzato WindRoseChart che utilizza syncfusion_flutter_charts per il rendering dei grafici radiali e CustomPainter per la griglia di sfondo. Il widget implementa una rosa dei venti completa con: (i) griglia radiale con cerchi concentrici e 36 linee radiali da 10° l'una; (ii) RadialBarSeries per visualizzare la frequenza e l'intensità del vento in ogni direzione; (iii) elaborazione dei dati complessa che combina i dati della velocità con quelli della direzione, matchando i timestamp, calcola la media per ognuno dei 36 settori e genera barre colorate in base all'intensità; (iv) legenda custom con 8 fasce di velocità ((0-1.5 m/s fino a >10.5 m/s) mappate su scala colori dal blu (calma) al rosso (tempesta)); (v) etichette dei gradi posizionate radialmente ogni 45° per l'orientamento cardinale.

L'algoritmo di processing del widget custom è particolarmente sofisticato:

1. Crea una map timestamp-indexed per velocità e direzione;

2. Per ogni timestamp in comune normalizza la direzione in gradi e la raggruppa in uno dei 36 settori;
3. Calcola la frequenza (numero di occorrenze) e velocità media per ogni settore;
4. Genera WindRoseData solo per i settori con dei dati (i settori che non presentano dati non vengono visualizzati);
5. Colora dinamicamente le barre usando pointColorMapper, per applicare il gradiante basato sulla velocità media del settore.

Codice 4.7: Funzione _processWindData() per rose dei venti

```
List<WindRoseData> _processWindData() {
  // Usa direttamente le liste invece di combinare sensori multipli
  List<SensorData> allSpeedData = windSpeedData;
  List<SensorData> allDirectionData = windDirectionData;

  // Crea mappa per combinare velocita e direzione per timestamp
  Map<DateTime, double> speedMap = {};
  Map<DateTime, double> directionMap = {};

  for (var speed in allSpeedData) {
    speedMap[speed.timestamp] = speed.value;
  }

  for (var direction in allDirectionData) {
    directionMap[direction.timestamp] = direction.value;
  }

  // Crea 36 settori per i gradi (ogni 10 gradi: 0deg, 10deg, 20deg,
      ..., 350deg)
  List<String> directions = [];
  for (int i = 0; i < 36; i++) {
    directions.add('${i * 10}');
  }
```

```
Map<String, List<double>> directionBins = {};
for (String dir in directions) {
  directionBins[dir] = [];
}

// Raggruppa i dati per direzione (ogni 10 gradi)
speedMap.forEach((timestamp, speed) {
  if (directionMap.containsKey(timestamp)) {
    double directionDegrees = directionMap[timestamp]!;
    // Normalizza la direzione e raggruppala in settori di 10
        gradi
    int sectorIndex = (directionDegrees / 10).round() % 36;
    String direction = '${sectorIndex * 10}';
    directionBins[direction]!.add(speed);
  }
});

// Crea dati per la rosa dei venti - SOLO settori con vento
List<WindRoseData> windRoseData = [];

for (String direction in directions) {
  List<double> speeds = directionBins[direction]!;
  if (speeds.isNotEmpty) {
    double avgSpeed = speeds.reduce((a, b) =>a + b) / speeds.
        length;
    double frequency = speeds.length.toDouble();

    windRoseData.add(WindRoseData(
      direction: direction,
      frequency: frequency,
      avgSpeed: avgSpeed,
    ));
```

```
    }
    // Non aggiungere settori vuoti - RadialBarSeries gestira
        automaticamente
  }

  return windRoseData;
}
```

4.3 Workflow del backend

Il backend dell'applicazione è un server implementato in Dart utilizzando il framework Shelf. All'avvio, il sistema carica il file .env per accedere alle variabili di configurazione, come le chiavi API di Firebase e le credenziali dei database. Subito dopo il caricamento delle variabili d'ambiente vengono inizializzati i service core dell'applicazione. SqlService viene istanziato come Singleton e crea e gestisce la connessione al database PostgreSQL. Parallelamente viene inizializzato InfluxDbService, sempre come Singleton, che configura la connessione al database InfluxDB. Entrambe le connessioni ai database richiedono parametri privati e sensibili che vengono prelevati dal file .env del backend.

Una volta che le connessioni ai database sono state stabilite, viene inizializzato il FirebaseAdminService, che carica il file firebase-service-account.json contenente tutte le credenziali del service account Firebase. Questo servizio viene utilizzato per l'invio di notifiche push tramite FCM e per verificare i token FCM.

Dopo l'inizializzazione dei servizi di base, il server procede con la configurazione della pipeline HTTP. Viene creata una pipeline che concatena una serie di middleware eseguiti in sequenza per ogni richiesta che il backend riceve dal frontend. In ordine:

- logRequests(): registra ogni richiesta HTTP con timestamp, path, status code della risposta e metodo.
- createCorsHeadersMiddleware(): aggiunge gli header necessari per permettere al frontend, in esecuzione su origini diverse, di effettuare richieste al backend.
- jwt_middleware.dart: intercetta ogni richiesta proveniente dal frontend e verifica la presenza dell'header Authorization. Se è pre-

sente, estrae il token JWT e lo decodifica utilizzando la chiave JWT_SECRET presente all'interno del .env, estraendo userId, role e email.

Una volta completata la configurazione della pipeline, vengono registrate le routes dell'applicazione. Ogni area funzionale dell'app ha la propria classe di routes:

- AuthRoutes: autenticazione e registrazione.
- SensorRoutes: gestione dei sensori e recupero dei dati.
- ThresholdsRoutes: gestione delle soglie personalizzabili.
- ConsorzioRoutes: gestione dei consorzi.
- IrrigationNoteRoutes: gestione delle note di irrigazione.

Nel momento in cui tutte le routes vengono registrate, il server si avvia e rimane in ascolto sulla porta 3000 (o quella specificata all'interno del .env) per tutte le richieste TCP in entrata. Quando una richiesta HTTP arriva da un client, come una chiamata POST alla route /auth/login con allegato un file JSON contenente le informazioni dell'utente, la richiesta attraversa la pipeline dei middleware. Il middleware di logging registra la richiesta, quello di CORS aggiunge gli header necessari e il middleware JWT verifica se l'endpoint richiede l'autenticazione. In questo caso, l'endpoint di login è pubblico, quindi il middleware JWT lo lascia passare senza verificare il token e la richiesta viene passata alla route della classe AuthRoutes, che invoca il metodo _login().

Il metodo _login() legge il body della richiesta estraendo email e password, e invoca lo SqlService per controllare il database PostgreSQL. Se l'utente esiste all'interno del database, viene recuperato l'hash bcrypt della password e confrontato con la password fornita nella richiesta di login. Se i due hash coincidono, viene generato un token JWT, firmato con la chiave segreta JWT_SECRET, e inserito, insieme alle altre informazioni sull'utente, in un oggetto JSON che viene restituito al frontend come Response.ok().

Per richieste ad endpoint protetti, come /sensor-data, il flusso varia leggermente. Quando arriva al backend una richiesta GET per /sensor-data, la richiesta attraversa la pipeline, ma invece di continuare normalmente, il middleware JWT ferma il flusso e verifica il token. Se il token della richiesta risulta valido, vengono estratti userId e role dell'utente e inseriti nel context della richiesta, che viene poi inoltrata a SensorRoutes per l'esecuzione del metodo _getSensorData().

Il metodo _getSensorData() estrae i parametri richiesti dalla query, come sensor, start e end (che indicano rispettivamente il sensore da cui prelevare i dati, la data di partenza e quella di fine) e, prima di procedere, verifica che l'utente che ha effettuato la richiesta possieda l'accesso al sensore richiesto interrogando la tabella user_sensors di PostgreSQL. Se l'utente ha accesso, il flusso procede interrogando InfluxDB. InfluxDB elabora la query, recupera i dati e li restituisce in formato CSV annotato. Il CSV contiene header con metadati e righe con i dati effettivi. Il file viene poi parsato da InfluxDbService riga per riga, estraendo i campi _time, _value, _field e altri metadati utili. Per ogni riga viene creato un oggetto Map con le chiavi timestamp, value, sensorName, measurementType, che vengono aggiunte a una lista. Una volta ottenuta la lista, il metodo applica eventuali filtri aggiuntivi e, se il frontend ha richiesto un campionamento tramite il parametro sample=true, viene applicato l'algoritmo di sampling.

I dati processati vengono quindi serializzati in un oggetto JSON e inviati al frontend passando per la pipeline in ordine inverso, dove il middleware di logging registra la risposta con status code 200, che viene infine inviata al client attraverso una connessione TCP.

Parallelamente ai processi delle richieste HTTP, il backend esegue servizi in background. Il più importante è il ThresholdMonitoringService, che viene avviato in modo isolato tramite una richiesta utente oppure come timer periodico nel file server.dart. Questo servizio esegue un ciclo infinito con intervalli configurabili (impostato a 60 secondi) in cui, a ogni iterazione, interroga il database PostgreSQL per recuperare tutte le soglie della tabella thresholds dove is_active=true. Per ogni soglia, il servizio legge i parametri e verifica se è il momento di controllarla confrontando il timestamp corrente con il parametro lastTriggeredAt + checkIntervalMinutes. Se non è ancora il momento, passa alla soglia successiva, altrimenti procede con la verifica.

Durante la verifica, il servizio interroga InfluxDB per ottenere gli ultimi valori dei sensori specificati nel parametro sensorNames e, se useAverage=true, calcola la media aritmetica tra i valori recuperati; altrimenti controlla i valori singolarmente. Il valore ottenuto (singolo o medio) viene confrontato con il thresholdValue utilizzando l'operatore aritmetico specificato nella soglia. Se la condizione è soddisfatta, il servizio verifica se è trascorso abbastanza tempo dall'ultima notifica confrontando lastTriggeredAt + notificationIntervalMinutes con il timestamp corrente. Se il cooldown è scaduto, il servizio invia la notifica; altrimenti viene soppressa per evitare spam.

Il messaggio FCM viene inviato attraverso FirebaseAdminService.sendNotification(), che effettua una richiesta POST all'API di FCM utilizzando il service account per il login. Firebase riceve la richiesta, verifica le credenziali e invia la notifica al dispositivo identificato dal token FCM registrato

nella tabella users. Dopo l'invio, il servizio aggiorna lastTriggeredAt con il timestamp corrente e procede con la soglia successiva. Una volta che il servizio ha processato tutte le soglie, rimane in attesa dell'intervallo configurato e, allo scadere, ricomincia il ciclo.

Un altro componente del backend è la gestione delle associazioni tra sensori, consorzi e utenti. Quando l'admin effettua una richiesta POST per assegnare un sensore a un consorzio all'endpoint /sensors/assign, la richiesta, contenente sensor_id e consorzio_id, viene inviata a SensorRoutes.assignSensorToConsorzio(). Questo metodo verifica che l'utente che ha effettuato la richiesta abbia il ruolo di admin e, se confermato, aggiorna il database PostgreSQL eseguendo una query UPDATE sulla tabella sensors, aggiornando il campo consorzio_id con quello contenuto nella richiesta.

Il flusso per l'associazione di un sensore a un utente da parte di un consorzio è simile. L'utente consorzio effettua una richiesta POST all'endpoint /sensors/user-associations, contenente user_id e un array di sensor_ids. La richiesta viene poi inviata a SensorRoutes.saveSensorAssociation(). Questo metodo verifica se l'utente ha il ruolo di amministratore e l'accesso ai sensori indicati. Se tutte le verifiche passano, il metodo esegue prima un'operazione DELETE sulla tabella associazione_sensori per rimuovere le associazioni precedenti relative all'userId indicato, e poi un'operazione INSERT per inserire tutte le nuove associazioni.

Il backend gestisce anche i processi di registrazione di nuovi utenti tramite gli endpoint /auth/register/user e /auth/register/consorzio. Quando arriva una richiesta POST a /auth/register/user contenente nome, cognome, email, password e consorzio_id, il metodo AuthRoutes._registerUser() verifica che l'email non sia già presente all'interno della tabella users, e che il consorzio indicato esista e sia approvato. Se le validazioni passano, il metodo esegue l'hashing della password e inserisce nella tabella users un nuovo record che avrà di default il campo is_approved=false, poiché deve essere approvato dall'amministratore del consorzio.

Per la registrazione di un consorzio, il flusso è simile ma coinvolge due tabelle. Il metodo AuthRoutes._registerConsorzio() riceve nome, cognome, email, password e il nome del consorzio, crea un record nella tabella consorzi con il campo is_approved=false e crea un altro record nella tabella users con campo consorzioId (intero incrementale) che punta al campo consorzioId della tabella consorzi.

Il backend implementa anche gli endpoint per la gestione delle note di irrigazione. Quando un utente effettua una richiesta POST all'endpoint /irrigation-notes, contenente tutti i dati inseriti nel form, il metodo IrrigationNoteRoutes.createNote() estrae lo userId dal context, verifica e valida i dati e inserisce un nuovo record nella tabella irrigation_notes del database

PostgreSQL. La nota viene associata all'utente tramite lo userId e può essere recuperata, eliminata o modificata da quest'ultimo.

L'intero flusso del backend ruota attorno alle richieste e risposte HTTP con processi asincroni, all'esecuzione di query sui database, alla trasformazione e all'elaborazione dei dati e alla restituzione di risposte sotto forma di oggetti strutturati di tipo JSON, il tutto gestito da middleware e servizi background che eseguono task periodici indipendentemente dalle richieste HTTP in entrata.

4.4 Workflow del frontend

L'applicazione inizia l'esecuzione dal file main.dart. Al momento dell'avvio viene caricato il file .env per accedere alle variabili di configurazione come URL del backend e credenziali per l'inizializzazione del servizio Firebase. L'applicazione costruisce poi il widget principale MyApp e inizializza gli screens. L'app determina dinamicamente quale vista mostrare: se l'utente ha già effettuato il login e il token JWT, carica la MainDashboard, altrimenti viene renderizzata la LoginScreen.

Quando l'utente tenta di effettuare l'accesso per la prima volta la schermata LoginScreen gestisce il processo di autenticazione. L'utente compila il form con email e password e al momento del submit viene chiamato il metodo _login() che inizializza il BackendService. Questo metodo estrae la configurazione dell'URL del backend da ApiConfig. Dopo aver determinato l'URL appropriato in base alla piattaforma, il BackendService effettua una richiesta POST all'endpoint /auth/login inviando le credenziali. Se l'autenticazione ha successo, il backend invia in risposta un oggetto JSON contenente il token JWT e le informazioni sull'utente che ha appena effettuato il login. Alla ricezione del JSON la LoginScreen estrae questi dati e li passa all'AuthService che salva il token, il ruolo e l'email dell'utente nelle SharedPreferences per avere persistenza all'interno di tutta la sessione.

Dopo il salvataggio dei dati di autenticazione viene inizializzato il FirebaseService che richiede il token FCM al sistema Firebase, gestisce i permessi di notifica e invia il token al backend utilizzando l'endpoint /users/fcm-token. Il backend salva il token nel database PostgreSQL nella tabella users, associandolo all'utente corrente.

Completata l'autenticazione e l'inizializzazione, l'applicazione carica la MainDashboard utilizzando Navigator.pushAndRemoveUntil() che rimuove tutte le schermate precedenti dallo stack di navigazione, impedendo all'utente di tornare alla schermata di login utilizzando il pulsante back. La MainDa-

shboard implementa una navigazione a tab, permettendo di navigare tra le diverse viste principali dell'applicazione.

All'utente, una volta loggato, verrà mostrata la HomeScreen. All'inizializzazione, la HomeScreen carica i sensori associati all'utente tramite una richiesta GET al backend passando il token JWT nell'header. Il backend verifica il token tramite il middleware JWT e, tramite l'ID univoco dell'utente, interroga il database PostgreSQL per recuperare tutti i sensori associati a quell'utente all'interno della tabella user_sensors.

Alla ricezione della lista dei sensori viene invocato il metodo _loadSensorData() che effettua una richiesta GET all'endpoint del backend /sensor-data passando come parametri i nomi dei sensori, data di inizio e di fine del range temporale. Il backend, alla ricezione della richiesta, interroga InfluxDB per recuperare i dati dei sensori specificati nella richiesta. I dati restituiti da InfluxDB sono in formato CSV annotato, che viene parsato dal backend e trasformato in JSON. Il frontend riceve questo JSON e lo deserializza creando oggetti SensorData.

La visualizzazione dei dati avviene tramite i widget. Il widget SensorChart riceve in input una lista di oggetti SensorData e SensorsVisibilityController che gestisce la visibilità delle serie nel grafico attraverso il Singleton. L'utente può così interagire con la legenda dei grafici e lo stato viene mantenuto all'interno del controller.

L'applicazione implementa anche un sistema di aggiornamento automatico dei dati tramite l'AutoRefreshService, un timer che esegue un refresh dei dati a intervalli regolari configurabili dall'utente all'interno della schermata delle impostazioni. Quando il timer scatta, l'applicazione esegue un callback che ricarica i dati dei sensori aggiornando i grafici e le tabelle.

Un aspetto fondamentale dell'app è la possibilità di creare delle soglie di monitoraggio. L'utente può accedere all'apposita sezione cliccando il pulsante "Gestione Soglie" all'interno delle impostazioni. All'apertura della vista viene effettuata una richiesta GET al backend che interroga PostgreSQL per individuare nella tabella thresholds le soglie appartenenti a quel user_id. Le soglie ricevute in risposta dal backend vengono poi visualizzate sotto forma di card contenenti informazioni sul tipo di misurazione, quali sensori sono monitorati, valore della soglia e stato di attivazione. L'utente può modificare soglie esistenti o crearne di nuove attraverso il pulsante "Nuova Soglia". Questo apre il ThresholdDialog che permette all'utente di creare una nuova soglia inserendo tutte le informazioni necessarie. Al salvataggio della nuova soglia, il ThresholdDialog crea un oggetto Threshold e lo invia tramite ThresholdService.createThreshold() o ThresholdService.updateThreshold() a seconda dell'azione effettuata dall'utente. Il backend salva la soglia nel database Post-

greSQL e il ThresholdMonitoringService inizia il ciclo di controllo periodico secondo l'intervallo inserito dall'utente.

In base al ruolo dell'utente che ha effettuato il login, le viste all'interno della schermata delle impostazioni variano. Per gli utenti Consorzio sarà mostrata una tab aggiuntiva "Gestione Consorzi". Questa schermata permette di gestire gli utenti, accettare o rifiutare users che effettuano richiesta per entrare nel consorzio, e gestire l'associazione dei sensori appartenenti al proprio consorzio. Per gli utenti Admin invece, sono presenti viste simili a quelle dell'utente Consorzio, solo a un livello superiore. L'Admin può accettare o rifiutare nuovi consorzi che hanno fatto richiesta e gestire l'associazione dei sensori ai consorzi. L'Admin visualizza inoltre un widget di input text che gli permette di mandare una notifica push a tutti gli utenti.

L'applicazione gestisce anche le note di irrigazione tramite la IrrigationNotesManagementScreen. Gli utenti possono registrare eventi di irrigazione manuale inserendo data di irrigazione e quantità d'acqua utilizzata. Al salvataggio, queste informazioni vengono inviate al backend tramite IrrigationNoteService.createNote() e salvate all'interno della tabella irrigation_notes.

L'intero flusso dell'applicazione ruota attorno a un ciclo di autenticazione–autorizzazione–richiesta–risposta. Ogni azione dell'utente invoca metodi dei services, che effettuano richieste HTTP al backend, ricevono risposte JSON e le passano ai widget per la visualizzazione.

Capitolo 5

Validazioni

Dopo una fase iniziale di sviluppo e progettazione svolta in locale, è stato necessario effettuare la migrazione ad un infrastruttura di produzione stabile, sicura e accessibile 24 ore su 24. Parallelamente, l'applicazione è stata preparata per la pubblicazione sul Google Play Store, passando attraverso le diverse fasi di testing. Inoltre, l sistema è stato sottoposto ad una validazione dell'esperienza utente tramite un questionario Post-Study System Usability Questionnaire (PSSUQ), mirato a valutare la percezione degli utenti nelle tre dimensioni chiave dell'esperienza utente: utilità del sistema, qualità delle informazioni e qualità dell'interfaccia.

5.1 Migrazione del backend

La migrazione del backend dall'ambiente di sviluppo in locale ad un'infrastruttura di produzione rappresenta una delle fasi più importanti e significative dell'intero progetto, poiché ha permesso di trasformare un prototipo funzionante in un servizio stabile e continuamente operativo. Durante la fase di sviluppo, l'intero sistema era eseguito su un'architettura Docker [1] locale, composta da tre elementi principali: un container dedicato al backend API, un'istanza PostgreSQL e un collegamente esterno all'InfluxDB già attivo all'interno del progetto TRACE.

Tuttavia, quest'approccio non era sufficiente per supportare le esigenze del progetto, che richiedeva un backend costantemente online per gestire le normali task degli utenti e gli eventi background dell'applicazione. Per questo motivo è stato necessario migrare l'ambiente di sviluppo in un ambiente di

[1] https://www.docker.com/

produzione con requisiti specifici: disponibilità 24/7, scalabilità per sostenere un carico crescente e configurazioni di sicurezza rafforzate.

La migrazione è avvenuta attraverso un processo strutturato:

1. Predisposizione dell'ambiente di produzione: è stato predisposto l'ambiente di produzione, basato su un server dedicato con Docker Engine e Docker Compose. Sono stati configurati i parametri di rete, il firewall e i certificati SSL necessari per la comunicazione sicura tra client e server.

2. Hardening dell'infrastruttura: è stata effettuata un'operazione di miglioramento e hardening dell'infrastruttura riconfigurando PostgreSQL seguendo parametri ottimizzati per la produzione, sono state rimosse quasi tutte le parti di codice relative al debug con print a console, utilizzati per il testing durante lo sviluppo dell'applicazione e sono stati configurati i volumi persistenti per garantire la conservazione dei dati durante gli aggiornamenti.

3. Deploy: è stata migrata l'intera infrastruttura ed il sistema è stato distribuito adottando una procedura a basso downtime. Sono inoltre stati eseguiti test end-to-end per verificare la piena operatività dello stack: connettività tra backend e database, accessibilità dell'API dall'esterno e funzionamento dell'app in ambiente reale.

5.2 Pubblicazione su Google Play Store

Parallelamente alla migrazione del backend, l'applicazione è stata preparata per la distribuzione attraverso il Google Play Store. Questo processo ha richiesto una serie di configurazione tecniche, come la generazione della build di produzione tramite Flutter e la pulizia degli asset non necessari.

Una volta ottenuto il bundle di produzione (.aab), è stata configurata la Google Play Console, inserendo le informazioni richieste per la pubblicazione: la scheda descrittiva, gli screenshot dell'app in esecuzione su diversi dispositivi di varie dimensioni, l'icona, la classificazione dei contenuti e la definizione delle autorizzazioni richieste. Particolare attenzione è stata posta alla sezione di "Data Safety", che richiede una descrizione dettagliata e precisa delle modalità di raccolta, gestione e trattamento di tutti i dati dell'utente.

Il percorso di pubblicazione prevede diverse fasi di testing progressive:

1. Fase di internal testing: questa fase ha coinvolto un gruppo ristretto di persone, composto principalmente dagli sviluppatori e dai responsabili

del progetto. Ha permesso di individuare bug relativi alla visualizzazione dei grafici, alla gestione delle notifiche e alla compatibilità tra i dispositivi, che sono stati corretti rapidamente prima di passare alla fase di testing successiva.

2. Fase di closed testing: questa è la fase tuttora attiva, che coinvolge un gruppo più ampio di utenti tester, coinvolgendo gli agricoltori e il consorzio associati al progetto TRACE. Questa fase sta permettendo di osservare il comportamento dell'applicazione applicata nei contesti reali: utilizzo nei campi, connessioni instabili o deboli, modelli differenti di smartphone e interazione quotidiane. Queste prove stanno fornendo feedback utile per il miglioramento futuro della stabilità, delle prestazione e della chiarezza dell'interfaccia dell'applicazione.

3. Fase di open testing: rappresenterà un test pubblico con un numero di utenti molto più ampio rispetto alle precedenti, finalizzato a valutare la scalabilità, del carico di rete, della compatibilità e delle qualità complessive del sistema prima della pubblicazione definitiva.

5.3 Validazione dell'usabilità tramite questionario PSSUQ

5.3.1 Metodologia e struttura del questionario

La valutazione dell'esperienza utente è stata condotta coinvolgendo il gruppo di utenti partecipanti alla fase di closed testing. Questi si dividono in tre macro-categorie: agricoltori, programmatori o sviluppatori e tester con nessuna conoscenza nell'ambito dell'agricoltura o della programmazione. Ogni partecipante, durante il testing dell'applicazione, ha dovuto svolgere due task: (i) controllare la temperatura di una WS relativa a 7 giorni fa; (ii) impostare un allarme che si attivi quando la velocità del vento di una WS supera un valore a tua scelta. Al completamento delle task è stato somministrato il questionario Post-Study System Usability Questionnaire (PSSUQ) [2] , uno strumento standardizzato che consente di misurare in modo oggettivo la percezione di usabilità del sistema.

Il PSSUQ si compone di 16 domande suddivise in tre dimensioni principali: System Usefulness (SU, domande 1–6), Information Quality (IQ, domande 7–12) e Interface Quality (IQ, domande 13–16). I partecipanti hanno risposto

[2]https://uiuxtrend.com/pssuq-post-study-system-usability-questionnaire/

esprimendo il loro grado di accordo con ciascuna affermazione su una scala da 1 a 5, dove il punteggio 1 corrisponde a "Completamente d'accordo" e il 5 corrisponde a "Completamente in disaccordo".

5.3.2 Risultati del questionario

La valutazione è stata condotta somministrando il questionario ad un totale di 11 tester, che si dividono in un tre macro-categorie: agricoltori, informatici e nessuna delle due. Di seguito verranno analizzati, tramite tre grafici, i risultati del questionario.

Valutazione PSSUQ per categoria di domanda e gruppo di utenti

Il grafico nella Fig. 5.1 presenta un confronto sistematico dei punteggi medi all'interno delle quattro dimensioni fondamentali del questionario PSSUQ. La visualizzazione è organizzata in modo da evidenziare le differenze di percezione tra i diversi gruppi di utenti che hanno partecipato alla valutazione.

L'asse x del grafico identifica le categorie elencate precedentemente (System Usefulness, Information Quality , Interface Quality) e in aggiunta anche la categoria Overall che rappresenta la media complessiva di tutte le 16 domande del questionario. L'asse y, invece, riporta il punteggio medio secondo la scala Likert utilizzata nel PSSUQ, che va da 1 a 5, dove 1 corrisponde a "completamente d'accordo" (massima soddisfazione), mentre il valore 5 indica "completamente in disaccordo" (minima soddisfazione).

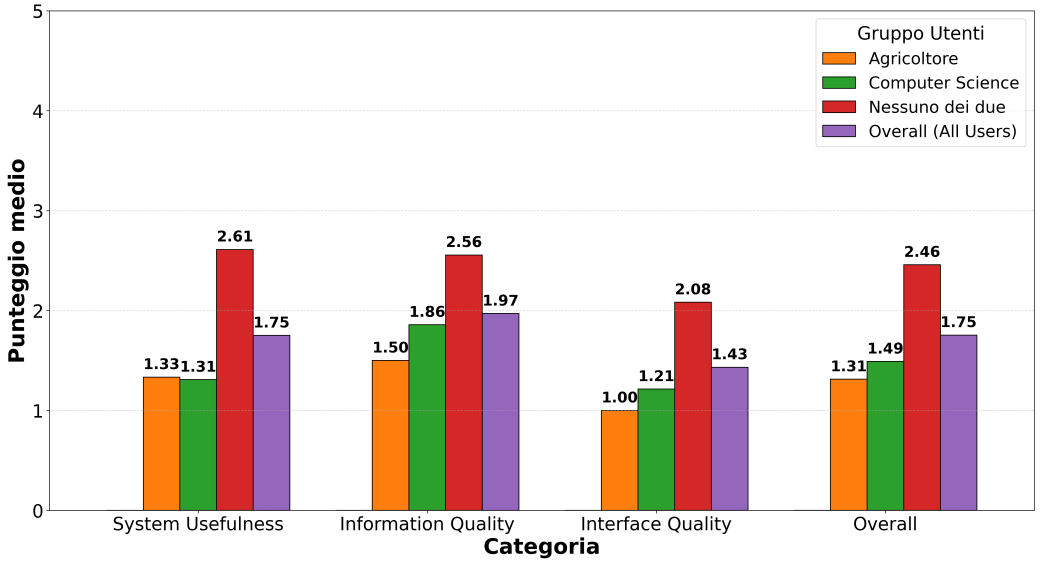


Figura 5.1: Valutazione PSSUQ del Sistema TRACE per Categoria e Gruppo di Utenti

Analisi dei risultati

- System Usefulness: gli agricoltori hanno espresso un punteggio medio di 1.33, indicando un grado di soddisfazione elevato. Analogamente gli utenti con background in Computer Science hanno attribuito un punteggio medio di 1.31. Questa vicinanza tra le due categorie suggerisce che il sistema risponde in modo efficace alle esigenze pratiche degli agricoltori ma anche ai criteri di valutazione standard degli utenti esperti nella programmazione. Gli utenti classificati come "Nessuno dei due" hanno assegnato un punteggio medio di 2.61, che, nonostante sia nella zona positiva della scala di valutazione, evidenzia una minore immediatezza nell'apprezzamento dell'utilità del sistema. La media overall ha un punteggio medio di 1.75, confermando un giudizio favorevole sull'utilità generale dell'applicazione;

- Information Quality: gli agricoltori hanno valutato questa dimensione con un punteggio medio di 1.50, gli utenti Computer Science con un punteggio di 1.86, il terzo gruppo con un punteggio di 2.56. La media overall di 1.97 indica che, nonostante la qualità delle informazioni fornite dall'applicazione sia apprezzata, esistono margini di miglioramento. Migliori messaggi di errori, maggiore documentazione e

informazioni testuali potrebbero aiutare a migliorare l'aspetto generale dell'applicazione;

- Interface Quality: gli agricoltori hanno attribuito un punteggio medio di 1.00, indicando una soddisfazione unanime rispetto all'interfaccia utente. Gli utenti Computer Science hanno assegnato un punteggio medio di 1.21, mentre il terzo gruppo ha espresso un giudizio di 2.08. La media complessiva risulta 1.43 e posiziona questa categoria come la migliore tra le tre.

- Overall: gli agricoltori hanno valutato complessivamente l'applicazione con un punteggio di 1.31, gli utenti Computer Science con un punteggio di 1.49, e il gruppo "Nessuno dei due" conferma il pattern osservato assegnando un punteggio più elevato rispetto agli altri gruppi con un valore di 2.46. La media generale è di 1.75 e conferma che l'applicazione rispecchia i criteri standard del questionario PSSUQ, specialmente per il pubblico target primario dell'app.

Distribuzione dei Punteggi PSSUQ per Domanda

Descrizione del grafico Il grafico nella Fig. 5.2 rappresenta uno strumento statistico per la visualizzazione della distribuzione delle risposte per ciascuna delle domande.

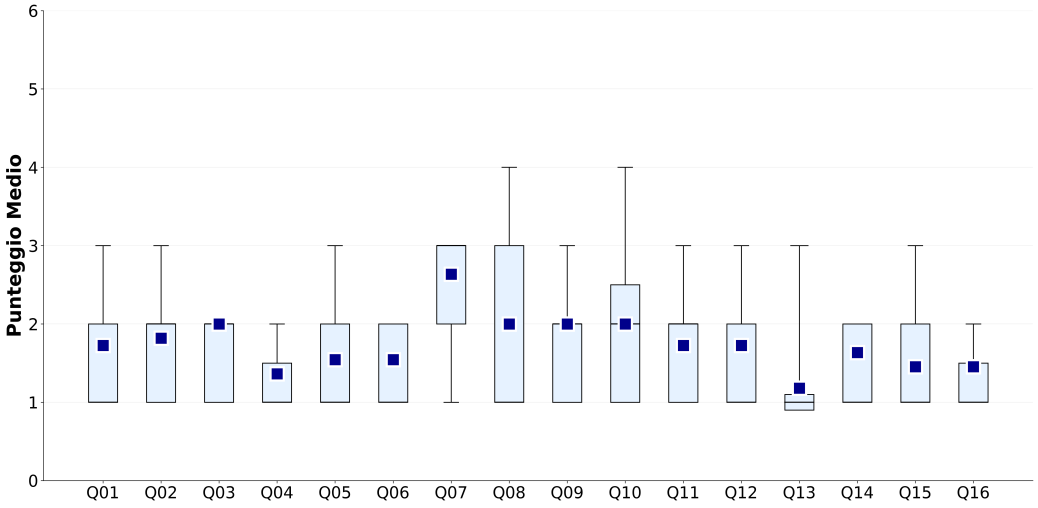


Figura 5.2: Distribuzione dei Punteggi PSSUQ per Domanda (Q01-Q16)

Analisi dei risultati

- System Usefulness: queste domande presentano una distribuzione generalmente compatta e centrata nella zona bassa della scala, indicando un livello elevato di soddisfazione. La domanda Q01 ha ottenuto una media di 1.73 con un box stretto posizionato tra i valori 1 e 2; la Q02 ha ottenuto una media di 1.82; Q03 ha ottenuto una media leggermente superiore di 2.00 con valori outliers che raggiungono il punteggio di 4.00; la Q04 ha ottenuto il risultato migliore dell'intera sezione con una media di 1.36; le domande Q05 e Q06 hanno ottenuto entrambe una media di 1.55. Questi valori confermano che la curva di apprendimento del sistema è percepita come ragionevole.

- Information Quality: queste domande hanno ottenuto una maggiore variabilità nelle risposte e punteggi mediamente più elevati rispetto alle altre due sezioni. La domanda Q07 rappresenta il punteggio più critico della sezione con una media di 2.64, con una distribuzione ampia dal valore 1 fino al 5 e con una mediana posizionata sul 3; le domande Q08, Q09 e Q10 presentano una media di 2.00 e una distribuzione più ampia rispetto alle domande della categoria precedente; le ultime due domande della categoria, Q11 e Q12, hanno ottenuto una media di 1.73. Questi valori indicano che, nonostante la media generale delle valutazioni di questa sezione siano positive, c'è possibilità di un margine di miglioramento.

- Interface Quality: questa sezione di domande rappresenta il punto di forza dell'applicazione. La domanda Q13 ha ottenuto il risultato migliore dell'intero questionario con una media di 1.18 e un box praticamente collassato sul valore 1, questo vuol dire che sul totale degli utenti che hanno votato a questionario, solo uno non ha assegnato il punteggio massimo, assegnando 3; la Q14 conferma il trend positivo con un punteggio medio di 1.64; la Q15 ha ottenuto una media di 1.45; la Q16 ha ottenuto un punteggio medio di 1.45. Questi valori confermano la valutazione positiva dell'intera applicazione insieme alle precedenti sezioni, confermando anche la sezione sulla qualità dell'interfaccia come punto di forza ottenendo i punteggi più elevati.

L'analisi della larghezza dei box e della posizione dei whiskers permette di individuare le domande che hanno generato maggiore o minore consenso tra gli utenti. Le domande Q04 e Q13 si distinguono per box estremamente stressi, indicando che quasi la totalità degli utenti hanno espresso valutazioni molto simili e, in questo caso, positive. Al contrario, la Q07 mostra la

distribuzione più ampia, segnalando delle esperienze molto differenziate tra i partecipanti al questionario. Gli outliers presenti nel grafico non devono essere interpretati come problematici o punti critici dell'applicazione, ma richiedono un'analisi qualitativa più approfondita per comprendere se derivino da casi d'uso particolari, aspettative iniziali diverse dell'utente o da lacune del sistema che si sono manifestate in condizioni specifiche.

Distribuzione delle Risposte per Domanda

Descrizione del grafico Il grafico nella Fig. 5.3 fornisce una rappresentazione immediata e dettagliata della composizione esatta delle risposte su scala Likert per ciascuna delle domande. A differenza del boxplot, che sintetizza la distribuzione tramite indicatori statistici, questa visualizzazione mostra il numero di utenti che hanno scelto ciascun livello della scala, permetto di cogliere pattern e tendenze.

La scelta cromatica effettuata all'interno del grafico, con transizione da colori caldi a colori freddi, facilita l'identificazione delle domande che hanno ricevuto valutazioni positive (dominanza di rosso e arancione) rispetto a quelle più problematiche (presenza significativa di giallo, azzurro o blu).

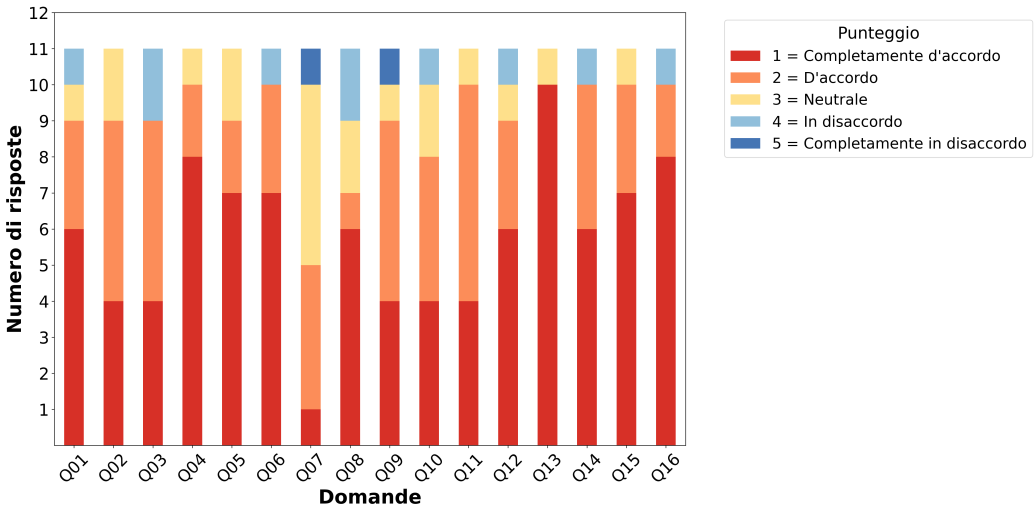


Figura 5.3: Distribuzione delle Risposte per Domanda - Scala Likert PSSUQ

Analisi dei risultati

- System Usefulness: si può osservare una composizione cromatica generalmente favorevole, con una predominanza di risposte nei livelli 1

e 2 (colori rosso e arancione). Le risposte neutrali (giallo) e negative (azzurro) sono presenti ma in quantità contenute.

- Information Quality: come evidenziato anche dai grafici precedenti questa categoria di domande presenta una composizione cromatica più variegata e meno uniforme. Questo indica che una porzione di utenti ha espresso valutazione neutre o tendenti al negativo rispetto alla qualità, accessibilità e chiarezza delle informazioni fornite dal sistema. Tuttavia, è possibile notare che le ultime due domande della sezione, Q11 e Q12, hanno un pattern cromatico più simile a quello osservato nella prima categoria. Questo suggerisce che il problema principale riscontrato dagli utenti non risiede tanto nel contenuto informativo in sé, quanto piuttosto nella modalità in cui le informazioni vengono comunicate, specialmente durante errori nell'applicazione.

- Interface Quality: conferma il ruolo di punto di forza del sistema, attraverso una composizione cromatica orientata verso i colori caldi. La domanda Q13, come evidenziato nell'analisi degli altri grafici, si configura come la domanda con il più alto livello di consenso dell'intero questionario, con una quasi totalità di risposte rosse. Le altre domande della sezione mantengono tutte una forte predominanza di colori caldi.

Analisi finale dei risultati

In sintesi, l'applicazione dimostra un livello di usabilità complessivamente elevato secondo i criteri standardizzati del questionario PSSUQ, con una media generale di 1.75 su una scala dove valori inferiori di 2.0 sono considerati indicatori di buona usabilità. L'interfaccia grafica emerge come il punto di eccellenza, con punteggi che rasentano la perfezione, specialmente da parte del pubblico target. L'utilità del sistema è ampiamente riconosciuta, con gli utenti che apprezzano la facilità d'uso e la rapidità con cui possono diventare produttivi. L'area che invece presenta maggiori margini di miglioramento riguarda la qualità delle informazioni, in particolare la chiarezza dei messaggi di errore a schermo e dell'help contestuale, aspetti che se migliorati potrebbero elevare ulteriormente la valutazione complessiva del sistema.

Conclusioni

Il presente lavoro di tesi ha affrontato la progettazione, lo sviluppo e la validazione di un'applicazione mobile completa per il monitoraggio e la gestione dei dati agricoli provenienti dal sistema IoT del progetto TRACE. L'obiettivo principale era fornire agli operatori agricoli, in particolare agli agricoltori specializzati nella coltivazione di piante officinali e aromatiche, uno strumento pratico, accessibile e utilizzabile sul campo per la consultazione dei dati ambientali raccolti dai sensori.

Un contributo significativo del lavoro riguarda l'implementazione degli algoritmi di campionamento intelligente dei dati. La possibilità di ridurre dataset da migliaia di punti a 500 punti strategicamente selezionati, mantenendo i picchi e le anomalie rilevanti, ha permesso di garantire prestazioni ottimali anche su dispositivi mobili di fascia media, senza compromettere la qualità delle informazioni visualizzate.

Il sistema di notifiche push basato su soglie personalizzabili rappresenta un altro elemento distintivo dell'applicazione. Gli utenti possono definire valori critici per ciascun parametro ambientale monitorato, selezionare i sensori da controllare, specificare l'intervallo di controllo e configurare la frequenza delle notifiche. Il servizio backend opera in background 24/7, interrogando periodicamente InfluxDB e inviando notifiche tramite Firebase Cloud Messaging quando le soglie vengono superate, permettendo agli agricoltori di intervenire tempestivamente anche quando non stanno utilizzando l'applicazione.

L'applicazione sviluppata ha raggiunto gli obiettivi prefissati, come dimostrato dai risultati della validazione con utenti reali attraverso il questionario PSSUQ. Con un punteggio medio complessivo di 1.75 su una scala dove valori inferiori a 2.0 indicano buona usabilità, il sistema ha ottenuto valutazioni particolarmente positive nelle tre dimensioni chiave analizzate, con margine di miglioramento specialmente nella sezione delle qualità delle informazioni fornite dal sistema.

Nonostante i risultati positivi, il lavoro presenta alcune limitazioni: (i) la valutazione di usabilità ha coinvolto 11 utenti, un numero relativamente li-

mitato che, sebbene sufficiente per identificare problemi di usabilità maggiori secondo le linee guida standard, potrebbe non catturare problematiche che emergerebbero con una base utenti più ampia e diversificata; (ii) attualmente l'applicazione è disponibile solo per dispositivi Android attraverso il Google Play Store, escludendo gli utenti iOS. Sebbene l'architettura Flutter consenta la compilazione anche per iOS con modifiche minime, le limitazioni temporali del progetto hanno impedito di completare il processo di pubblicazione sull'Apple App Store; (iii) sebbene l'applicazione memorizzi localmente alcune informazioni (token di autenticazione, preferenze utente), la maggior parte delle funzionalità richiede una connessione attiva al backend, e in contesti rurali con copertura cellulare limitata o assente, questo può rappresentare un ostacolo significativo all'utilizzo dell'applicazione; (iv) nonostante l'interfaccia sia stata valutata positivamente, l'applicazione offre opzioni limitate di personalizzazione dell'esperienza utente. Gli utenti non possono, per esempio, riorganizzare l'ordine delle sezioni nella dashboard, nascondere parametri non rilevanti o salvare configurazioni preferite di visualizzazione dei grafici.

Sulla base dell'esperienza maturata durante lo sviluppo e dei feedback raccolti durante la fase di testing, è possibile identificare diverse direzioni per l'evoluzione dell'applicazione: (i) la pubblicazione dell'applicazione anche su iOS rappresenta la priorità immediata per garantire la massima accessibilità; (ii) L'arricchimento dell'applicazione con modelli di machine learning addestrati sui dati storici potrebbe fornire previsioni sulle condizioni atmosferiche future e alert proattivi su potenziali situazioni critiche prima che queste si verifichino effettivamente; (iii) l'introduzione di strumenti di comunicazione tra membri dello stesso consorzio potrebbe trasformare l'applicazione da strumento di monitoraggio individuale a piattaforma collaborativa per la gestione condivisa delle risorse agricole.

Bibliografia

[1] Ramide Augusto Sales Dantas, Milton Vasconcelos da Gama Neto, Ivan Dimitry Zyrianoff, and Carlos Alberto Kamienski. The swamp farmer app for iot-based smart water status monitoring and irrigation control. In *2020 IEEE International Workshop on Metrology for Agriculture and Forestry (MetroAgriFor)*, pages 109–113. IEEE, 2020.

[2] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[3] Alexandros Kaloxylos, Aggelos Groumas, Vassilis Sarris, Lampros Katsikas, Panagis Magdalinos, Eleni Antoniou, Zoi Politopoulou, Sjaak Wolfert, Christopher Brewster, Robert Eigenmann, et al. A cloud-based farm management system: Architecture and implementation. *Computers and electronics in agriculture*, 100:168–179, 2014.

[4] Sheikh Mansoor, Shahzad Iqbal, Simona M Popescu, Song Lim Kim, Yong Suk Chung, and Jeong-Ho Baek. Integration of smart sensors and iot in precision agriculture: trends, challenges and future prospectives. *Frontiers in Plant Science*, 16:1587869, 2025.

[5] Francis J Pierce and Peter Nowak. Aspects of precision agriculture. *Advances in agronomy*, 67:1–85, 1999.

[6] KP Vyshali Rao, EK Trisha Morey, Vanipenta Bhanuprakash Reddy, and M Vamshi. Crop connect: Empowering farmers through digital community. In *International Conference on Emerging Research in Computing, Information, Communication and Applications*, pages 151–168. Springer, 2024.

[7] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. Apache iotdb:

A time series database for iot applications. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023.

[8] Ivan Zyrianoff, Andrea Iannoli, Federico Montori, Luca Sciullo, Luciano Bononi, Alice Baldissara, Beatrice Brintazzoli, Enrico Dall'Olio, Mattia Alpi, Rocco Enrico Sferrazza, Giovanni Dinelli, Ilaria Marotti, and Marco Di Felice. Traceability and research in the agricultural chain of medicinal and aromatic plants (maps): An iot-based approach. In *2025 IEEE International Workshop on Metrology for Agriculture and Forestry (MetroAgriFor)*, 2025. Proceedings not yet published.