POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN
INGEGNERIA INFORMATICA

SUBJECT : SOFTWARE ARCHITECTURE AND PATTERN DESIGN

PROJECT REPORT

GREENFIELD ADVISOR – AGRICULTURE SUSTAINABILITY

PROFESSOR:
MARINA MONGIELLO

STUDENTS: FRANCESCO COLELLA – ZAIDI SYED SAMAR UI HASSAN – ALBERTO VOX

AY 2025–2026

Contents

List of Tables

Listings

List of Figures

1 Introduction

Agriculture has long served as the foundation of human civilization and currently faces mounting pressures from climate change, soil degradation, and resource constraints, while needing to deliver higher productivity with reduced environmental footprints. Challenges include extreme weather, market shocks, and biodiversity loss, alongside ambitious policy targets such as the Kunming-Montreal Global Biodiversity Framework and the EU Farm to Fork Strategy [1].

With global population projections exceeding 9 billion by 2050, the demand for sustainable food production is escalating rapidly [2]. To address these growing needs, the agricultural sector is undergoing a profound transformation driven by technological innovation such as precision farming, IoT, AI, robotics, and remote sensing that are transforming production systems and food value chains, enabling improved decision-making and reducing trade-offs. These innovations are key to achieve sustainability, resilience, and animal welfare in modern agriculture [1, 2].

2 State of the Art

Green agriculture represents a paradigm shift toward sustainable, resource-efficient, and climate-resilient farming systems. Recent advancements combine ecological principles with digital technologies to optimize productivity while minimizing environmental impact.

Precision and Smart Agriculture: Modern green agriculture leverages IoT sensors, AI, and machine learning to monitor soil moisture, nutrient levels, and microclimatic conditions in real time. Predictive analytics enable precise irrigation and fertilization, reducing resource waste and improving yields.
Drones and autonomous robots equipped with multispectral cameras support early detection of crop stress and disease, while edge computing ensures rapid decision-making in the field. Sensors play a crucial role in agriculture, detecting environmental changes and transmitting information to processors; smart sensors integrate onboard computing capabilities, allowing them to process and analyze data independently [3].

Climate-Smart and Regenerative Practices: Climate-smart agriculture integrates agroforestry, cover cropping, and conservation tillage with digital advisory systems to enhance resilience against climate variability. Regenerative approaches focus on soil carbon restoration and biodiversity, increasingly supported by AI-driven modeling and digital twins for ecosystem simulation.
"A Digital Regenerative Agriculture would prioritise farm environmental performance as a driver of productivity and provide the means of effectively quantify system capacity and condition" [4].
Digital twins represent the virtual version of the corresponding farm, where real-time data collected are integrated to improve decision-making and productivity [5, 6].

Controlled-Environment and Soilless Systems: Vertical farming and hydroponics have emerged as key innovations, utilizing LED lighting, automated nutrient delivery, and AI-based growth optimization. These systems enable year-round production with minimal land and water use, and blockchain technologies are being adopted for traceability and food safety.
They allow intensive production in less space strongly reducing land and water usage. Hydroponic farming itself could increase the yield per area up to 11 times while consuming until 13 times less water [7, 8].

Sustainable Inputs and Soil Health Monitoring: Advances in biofertilizers, biostimulants, and nanotechnology improve nutrient efficiency and soil microbiome health. Sensor-based soil analysis combined with AI achieves near-perfect classification accuracy, enabling site-specific nutrient management [9].

Cross-Cutting Digital Trends: Green agriculture increasingly relies on cloud-edge architectures, robotics, and blockchain for transparency and automation. Sustainability metrics now extend beyond carbon to include biodiversity and ecosystem services, aligning with global policy frameworks [10].

3 Methodology

GreenField Advisor is a platform designed to acquire data from heterogeneous sources (e.g. field sensors, meteorological datasets, camera images, etc.) and transform them into operational recommendations aimed at reducing the waste of water, energy, and fertilizers. Its primary objective is to foster more sustainable agricultural production through advanced analysis and intelligent decision-support mechanisms.

3.1 System Architecture

The system is organized into three main components that work together to deliver actionable insights. The first component, **Data Collection**, focuses on gathering information from multiple sources, ensuring a comprehensive view of the agricultural environment. Once collected, these data are processed through the **Data Processing Pipeline**, which handles tasks like cleaning, feature engineering, and parameter estimation to guarantee accuracy and reliability. Finally, the **Recommendation Module** leverages artificial intelligence models and complementary strategies to generate operational suggestions aimed at optimizing resource use and promoting sustainable agricultural practices.

3.1.1 Functional Requirements

The **Functional Requirements** describe what the system must do, namely the functionalities and behaviors to be implemented in order to meet the needs of the user or the process. It refers to the operations, services, and interactions that the system is expected to support. They can be grouped into the following main areas:

- **Data Utilization & Simulation** ensures reproducibility in controlled environments by leveraging existing datasets and a *Digital Twin* approach. The system simulates real-time sensor inputs (*IoT Ingestion*) to mimic field conditions for development and testing purposes.
- **Real-Time Dashboard** provides an intuitive visualization layer, enabling users to monitor the state of key resources—such as water, energy, and fertilizers—alongside environmental parameter trends via dynamic charts and KPI cards. It also serves as the interface for system interactions.
- **Hybrid Operative Suggestions** core logic module delivers optimized recommendations for irrigation and fertilizer dosage. Unlike traditional systems, it employs a *Hybrid Intelligence* approach, combining deterministic rule-based controls with AI-driven predictions (Logistic Regression) to issue alerts in cases of potential waste or plant stress.
- **Computer Vision Diagnosis** allows users to upload imagery of crop leaves to detect pathologies on-demand. The system utilizes a Deep Learning model (CNN) to classify plant health status (e.g., Healthy vs. Powdery Mildew) and provides immediate visual feedback.
- **Smart Notification & Reporting** that, instead of passive manual exports, implements an *Event-Driven Reporting* mechanism. Upon detecting critical

anomalies (e.g., simultaneous water and nutrient stress), it automatically aggregates the data from the last 15 monitoring cycles and dispatches a detailed Email Report containing the incident history and actuator status.

- **User Profiling & Dynamic Configuration** ensures secure access via JWT authentication and allows users to manage their profiles. Crucially, it provides a *Dynamic Settings* interface where users can adjust agronomic thresholds (e.g., max temperature, humidity limits) in real-time, synchronizing these preferences instantly with the backend logic.

3.1.2 Non-Functional Requirements

The **Non-Functional Requirements** define how the system should operate, focusing on qualitative attributes and constraints that influence performance rather than core functionalities. These requirements typically encompass several key aspects:

- **Scalability** leverages an *Event-Driven Architecture*, allowing producer and consumer microservices to scale independently to handle increasing data loads without architectural bottlenecks.
- **Reliability** ensures operational continuity through asynchronous messaging. The message broker acts as a buffer, preventing data loss even if downstream consumers are temporarily unavailable.
- **Interoperability** facilitates integration with diverse environments by using standard JSON payloads for internal communication and exposing REST APIs and WebSockets for external client interactions.
- **Usability** prioritizes user experience via a responsive **React** Dashboard that translates complex telemetry into intuitive KPI cards and real-time charts, making data accessible to non-technical operators.
- **Performance** minimizes latency for decision-making. The integration of Socket.IO for data streaming and lightweight AI models (*MobileNetV2*) ensures near real-time responsiveness.
- **Security** protects data and access through a **Microservice-based Auth System**. It enforces stateless authentication via **JWT**, input validation with **Zod**, and secure credential storage using **bcrypt**.
- **Maintainability** promotes code longevity and ease of updates through the strict application of Design Patterns (*Strategy*, *Observer*, *Chain of Responsibility*), allowing logic modularization without affecting the core system.
- **Explainability** ensures trust in automated decisions. Every alert generated by the system includes a specific, human-readable "Reason" field (e.g., specific threshold violation or probability score), distinguishing it from "black-box" solutions.

3.1.3 Chosen Architecture

The choice of the **Project Architecture** has been a critical factor in designing a system that is modular, scalable, and compatible. To achieve these objectives, a **microservice architecture** combined with an **event-driven** approach has been adopted.

The **microservice architecture** decomposes complex software systems into a set of autonomous components, each assigned a well-defined responsibility. This modularization fosters flexibility and maintainability, as services interact through standardized network protocols rather than direct integration. Unlike monolithic architectures—where components are tightly coupled and deployed as a single unit—microservices enable independent development and deployment, thereby enhancing scalability and adaptability in large-scale environments.

Although designed for autonomy, microservices must communicate to fulfill business functions, creating potential dependencies. To maintain loose coupling, **asynchronous event-driven communication** is often preferred over synchronous calls. This approach decouples producers from consumers, reducing temporal and structural dependencies by removing the need for immediate, direct invocation [11].

The **event-driven** paradigm further enhances responsiveness by enabling rapid reactions to data changes. It relies on the publish/subscribe (pub/sub) model, where sensors or other data sources publish events, and subscribed services automatically update themselves, ensuring real-time adaptability and efficient communication.

Event-driven microservice architectures offer substantial advantages compared to traditional monolithic systems. By employing asynchronous, event-based communication, these architectures achieve a high degree of loose coupling among services, thereby minimizing direct dependencies and reducing the systemic impact of localized modifications. This decoupling significantly enhances scalability and maintainability within distributed environments [12].

Empirical studies corroborate these benefits. For instance, several authors report that organizations adopting event-driven microservices observed a 42% reduction in cross-service dependencies relative to implementations based on conventional communication models [12]. These findings underscore the role of event-driven paradigms in mitigating integration complexity and promoting modularity.

Furthermore, multiple case studies document the practical advantages of this approach. Notably Ghosh [12] provides detailed analysis of implementations within a financial services firm, an e-commerce platform, and a healthcare provider, illustrating measurable improvements in operational efficiency and overall system effectiveness.

Event-driven architectures are particularly suited for Internet of Things (IoT) environments, where distributed sensors and devices generate large volumes of events [13].

The adoption of Event-Driven Architecture (EDA) requires a thorough cost-benefit analysis, considering implementation and maintenance expenses. Additionally, its distributed nature heightens security risks, necessitating robust frameworks with authentication, authorization, and encryption to ensure data integrity and user privacy [14].

3.2 Main Architectural Components

The project's **back-end** employs an event-driven microservices architecture specifically designed for monitoring and automation within precision agriculture. The system integrates simulated IoT telemetry, Machine Learning (Logistic Regression), and Deep Learning (Computer Vision) to deliver real-time agronomic recommendations.

The architectural organization of these components is illustrated in the following High-Level Component Diagram (Fig. 1). The schematic depicts the logical separation between the **Client Layer** and the **Docker Host Environment**, highlighting how the synchronous API Gateway orchestrates user interactions while the central **Kafka Event Bus** manages the asynchronous backend workflows.

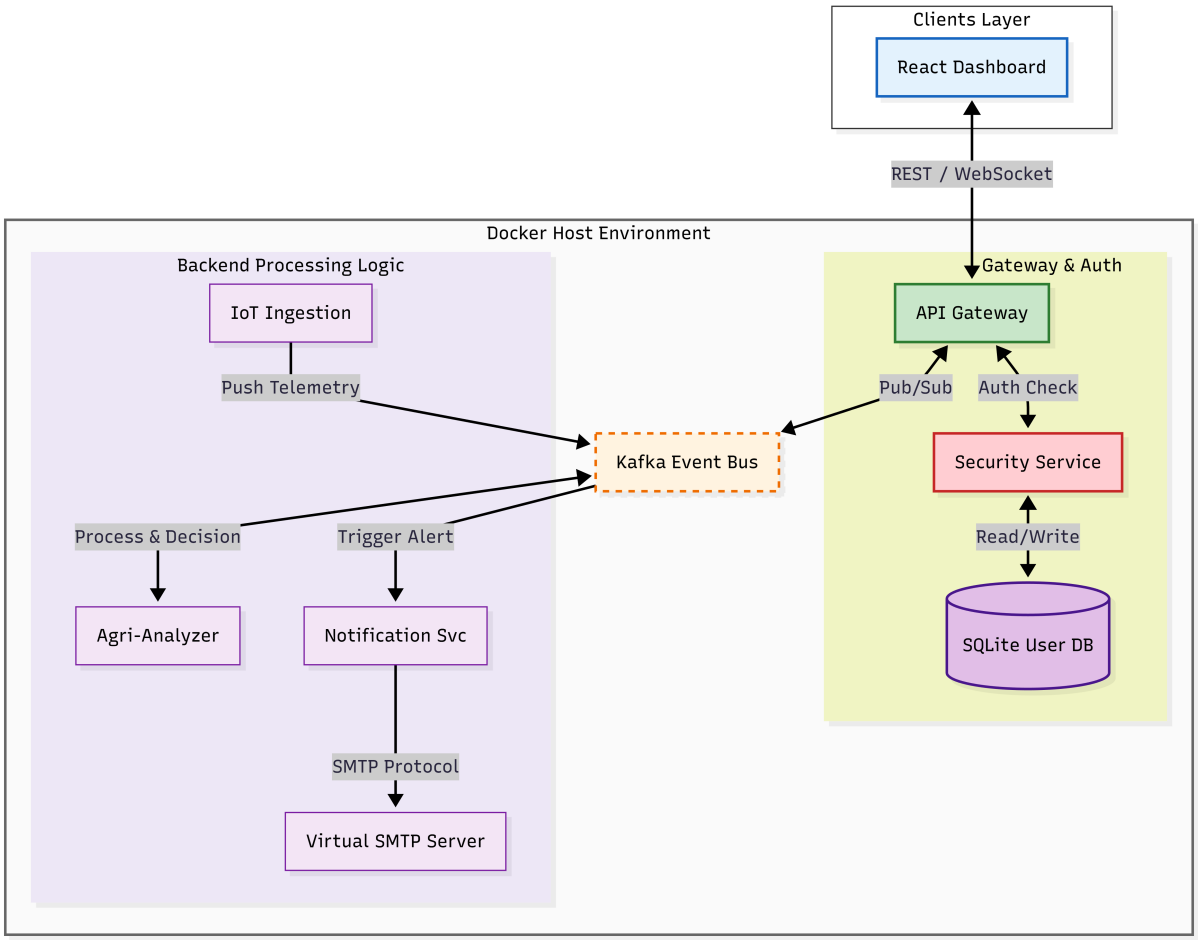

Figure 1: System Component Diagram showing the containerized microservices architecture and the central role of the Event Bus.

Main architectural components are:

- **Event Bus (Kafka):** acts as the system's backbone, decoupling the microservices. Instead of a complex multi-hop pipeline, the system is optimized around two high-throughput streams:
  - **sensor-data** - The ingestion channel for raw telemetry streamed by the producers.

- **system-advice** - The decision channel where the Agri-Analyzer publishes operational commands, subscribed to by the Notification Service and Dashboard.

- **Microservices:** set of autonomous components which are part of the full architecture.

Microservice	Architectural Role	Logic (Code)
IoT Ingestion Service	Producer	Simulates field telemetry by ingesting CSV datasets. Adds traceability metadata and streams JSON events to the sensor-data topic.
Agri-Analyzer Engine	Processor & Consumer	Orchestrates the Data Pipeline (Cleaning/Feature Engineering), and applies Hybrid Strategy logic (Rules + Logistic Regression AI) to generate operational advice. Publishes to system-advice.
Notification Service	Observer & Aggregator	Subscribes to system advice. Implements a Stateful Sliding Window (buffer) to monitor anomalies over time. Aggregates multiple triggers (e.g., Irrigation + Energy) into a single SMTP Email Report sent via the local Debug Server or Gmail.
API Gateway & Vision AI Gateway	Processor	Acts as the bridge between Backend and Frontend (React). Manages Socket.IO for real-time data streaming and hosts the Deep Learning Model (CNN) for on-demand leaf pathology classification via REST API.
Security Service	Service Provider	Dedicated microservice for User Management. Validates credentials using Zod, stores data in SQLite, and issues JWT Tokens for stateless authentication across the platform.

Table 1: List of Microservices

- **Patterns:** general, reusable solutions to recurring problems in the design of complex software systems. They are not ready-to-use code but rather conceptual guidelines and structures that help organize system components in a coherent, scalable, and maintainable way.

  Have been used:

  - **Observer** for sensor subscriptions: implemented in the Notification Service (*notification_consumer.py*), which monitors the *system-advice* stream. It detects state transitions (e.g., OFF - ON) and triggers the reporting workflow.

– **Strategy** for interchangeable models: utilized in the Prediction Service (*strategies_model.py*) to define a common interface for decision logic. This allows the system to switch seamlessly between *RuleBasedStrategy* and *LogisticRegressionStrategy* based on configuration.
– **Chain of Responsibility** for the data pipeline: orchestrates the Data Processing Pipeline (*pipeline.py*). Data flows through a sequence of independent handlers (Cleaning - Feature Engineering - Inference), promoting component reusability.

The project's **front-end** was developed as an interactive, real-time dashboard utilizing **React**, a leading library for engineering modern and dynamic user interfaces. The primary objective is to provide users with an immediate and intuitive visualization of sensor data and AI-generated insights, while minimizing latency and computational overhead.

3.3 Implementation Details

3.3.1 Event Bus

The **Greenfield Advisor** event-driven backbone is architected around a containerized **Apache Kafka** cluster, managed via **Docker Compose** and **Zookeeper**. This containerization strategy was adopted to ensure environment immutability and cross-platform reproducibility, thereby mitigating configuration discrepancies between development and deployment phases.
To facilitate communication within a hybrid container-host environment, a dual-listener configuration was implemented:

- **Internal Listener (kafka:29092):** Dedicated to intra-container traffic and service orchestration within the isolated Docker network.
- **External Listener (localhost:9092):** Established via port mapping to allow host-resident microservices—including the Sensor Producer and Agri-Analyzer—to interface with the broker with minimal latency.

While the current implementation utilizes the **PLAINTEXT** protocol to streamline debugging and payload inspection, the architecture is designed for forward compatibility. The system is "production-ready," allowing for the seamless integration of **SASL/SSL** encryption and authentication protocols through modular configuration updates, ensuring scalability and security without requiring modifications to the underlying application logic.

3.3.2 Microservices and Pattern

IoT Ingestion Service, which is realized through the `producer_sensor.py` module (Listings 1÷2), serves as the primary ingestion point for the system's data stream. It loads a tabular dataset using the centralized `load_dataset_robust` function and streams each row as a JSON event to the Kafka topic `sensor-data`. Each event incorporates specific metadata fields—`_event_type`, `_row_id`, and `_ts`—to facilitate end-to-end traceability and stream categorization. The module implements row-by-row streaming with a defined throttling mechanism (0.5 s) to enable controlled testing and replay scenarios.

Architecturally, the producer implements the **Publisher** role within the **Observer pattern**, effectively decoupling telemetry generation from downstream domain consumers and modeling pipelines. This separation of concerns enhances both scalability and system resilience.

This module is responsible for the ingestion phase of the system. Within the overall architecture, it operates upstream of domain-specific consumers (energy, fertilization, irrigation) and the notification service, serving as the primary source that feeds the event-driven workflow.

- **Interface and I/O** The module takes as **input** a **CSV file** (`data_test.csv`), which is loaded through `load_dataset_robust(...)` from `data_loader.py`. This function ensures robust data handling, including type validation, missing value imputation, and proper formatting.

  Configuration parameters are hard-coded within the module: `TOPIC = "sensor-data"` and `BOOTSTRAP_SERVERS = "localhost:9092"`.

  For each row in the DataFrame, the module produces as **output** a **JSON event** and publishes it to the Kafka topic `sensor-data`.

  Examples of step by step functioning

```
def build_producer():
    return KafkaProducer(
        bootstrap_servers=BOOTSTRAP_SERVERS,
        # Spezziamo le righe lunghe per rientrare nei margini
        value_serializer=lambda v:
            json.dumps(v).encode("utf-8"),
        key_serializer=lambda k:
            k.encode("utf-8") if k else None
    )
```

Listing 1: Extract from code "producer_sensor.py" - Producer Initialization

Connecting to the Kafka broker and defining serializers: application establishes a link with the Kafka messaging system and specifies how data should be converted into a suitable format for transmission. In this case, the payload is serialized as JSON and encoded in UTF-8.

The message key is **important for partitioning** because Kafka uses it to decide which partition the message will go to. If all messages use the same fixed key (e.g., "sensor"), they will all be placed in the same partition. This guarantees **total ordering** of messages but **reduces parallelism**, since only one partition is being used.

```python
for idx, row in df.iterrows():
    event = row.to_dict()

    # Metadata
    event["_event_type"] = "sensor_reading"
    event["_row_id"] = int(idx)
    event["_ts"] = time.time()

    producer.send(TOPIC, key="sensor", value=event)
    print(f"[Producer] sent row={idx}")
    time.sleep(0.5)
```

Listing 2: Extract from code "producer_sensor.py" - Sending Loop

The **code loops** through each row of a DataFrame, converts it into a dictionary, adds metadata (event type, row ID, timestamp), and sends it as a JSON message to a Kafka topic using a fixed key "sensor".

This ensures all messages are in order but limits parallelism. A short delay (0.5s) is added between sends to control the rate.

- **Design Patterns and Architectural Choices** The system employs an event-driven, publish–subscribe model, decoupling data production from processing and allowing independent consumers.

  Each event includes a minimal metadata schema (`_event_type`, `_row_id`, `_ts`) to ensure traceability. Partitioning uses a fixed key (`"sensor"`), which guarantees message ordering but limits scalability.

Agri-Analyzer Engine, which is realized through the `analyzer.py` module (Listings 3÷6), represents the computational core of the Greenfield Advisor architecture. Unlike a simple passive consumer, this microservice acts as a real-time decision engine that orchestrates the entire processing flow, from raw data ingestion to the formulation of agronomic advice. Its internal architecture is designed to reconcile two apparently conflicting requirements: the deterministic reliability of rule-based systems and the predictive capacity of Machine Learning.

- **Initialization & Cold-Start Training**

  Upon service startup, before entering the event listening loop, the system executes a pre-training phase. This architectural choice ensures that the **Machine Learning** (ML) models are immediately operational without relying on external training services. The code loads a historical dataset (using `load_dataset_robust`), instantiates the processing pipelines, and trains three distinct **Logistic Regression** models for Irrigation, Fertilization, and Energy.

```python
try:
    csv_path = "dataset/enriched_tomato_irrigation_dataset.csv"
    if not os.path.exists(csv_path):
        csv_path = "dataset/data_test.csv"

    df_raw = load_dataset_robust(csv_path)
    cleaner = DataCleaner(FeatureEngineer())
    df_train = cleaner.handle(df_raw).fillna(0)

    # Training AI
    strat_irr = LogisticRegressionStrategy(max_iter=500)
    strat_fert = LogisticRegressionStrategy(max_iter=500)
    strat_en = LogisticRegressionStrategy(max_iter=500)

    print(" ... Addestramento Logistic Regression...")
    strat_irr.train(df_train[FEATURES],
                    rule_irr.predict(df_train))
    strat_fert.train(df_train[FEATURES],
                     rule_fert.predict(df_train))
    strat_en.train(df_train[FEATURES],
                   rule_en.predict(df_train))

    # Creazione Pipeline
    pipe_irr = DataCleaner(FeatureEngineer(ModelEstimator(
        strat_irr, FEATURES, "Irrigation")))

    pipe_fert = DataCleaner(FeatureEngineer(ModelEstimator(
        strat_fert, FEATURES, "Fertilization")))

    pipe_en = DataCleaner(FeatureEngineer(ModelEstimator(
        strat_en, FEATURES, "Energy")))

    print("ANALYZER: Modelli pronti e operativi.")

except Exception as e:
    print(f"Errore critico nel Training: {e}")
```

Listing 3: Extract from code “analyzer.py” - Training Phase

The analyzer implements the **Chain of Responsibility** pattern: it facilitates the addition, removal, or reordering of pipeline stages without necessitating modifications to other system components (such as models, consumers, or producers). Consequently, this approach fulfills the core requirements of **modularity** and **interchangeability** specified in the project brief.

The **Chain of Responsibility** orchestrates data processing through four specialized, decoupled stages:

- **Handler (Base Class):** Establishes the recursive delegation logic (`self.next`). It provides the structural backbone for sequential execution.
- **DataCleaner (Standardization):** Enforces data integrity by normalizing categorical targets into binary format (0/1), encoding phenological crop stages, and applying plausibility filters.
- **FeatureEngineer (Agronomic Enrichment):** Synthesizes high-level indicators from raw telemetry. It derives stress-related booleans and calculates the **Evapotranspiration ratio (ET_ratio)**.

– **ModelEstimator (Inference Layer):** Integrates the **Strategy Pattern** to execute predictive logic. It decouples the mathematical model (e.g., Logistic Regression vs. Heuristic Rules) from the pipeline flow.

- **Asynchronous Event Loop & Dual Subscription**

  A distinctive feature of the Analyzer is its Dual Subscription: the consumer listens not only to sensor data but also to a configuration channel. This configuration enables two parallel flows:

  – **Data Flow** (`sensor-data`): Receives field telemetry (Soil Moisture, NPK, Air Temperature).
  – **Control Flow** (`system-settings`): Receives threshold updates from the user.

  When a message arrives on the `system-settings` topic, the system updates the global `SYSTEM_CONFIG` variable in real-time (**Hot-Swapping**), allowing system behavior modification without requiring a Docker microservice restart.

- **The Hybrid Inference Model** The module's true innovation lies in its hybrid inference logic. For each received data packet, the Analyzer calculates two parallel opinions:

  – **Deterministic Layer (Rule-Based Strategy).** This layer guarantees safety and adherence to strict agronomic guidelines. It employs adjustable thresholds to trigger alerts for irrigation, fertilization, and energy consumption based on real-time agronomic telemetry.

```
res_rules = {
    'irrigation': {
        'status': 'ON' if rule_irr.predict(df_single)[0]
                  == 1 else 'OFF',
        'reason': f"Soglia attiva: < {rule_irr.m_thr}%"
    },
    'energy': {
        'status': 'ACTIVE' if rule_en.predict(df_single)[0]
                  == 1 else 'OFF',
        'reason': f"Range: {rule_en.tmin_thr}-
                  {rule_en.tmax_thr}C"
    },
    'fertilization': {
        'N': 'LOW' if data.get('Nitrogen_mg_kg',0) <
             rule_fert.n_thr else 'OK',

        'P': 'LOW' if data.get('Phosphorus_mg_kg',0) <
             rule_fert.p_thr else 'OK',

        'K': 'LOW' if data.get('Potassium_mg_kg',0) <
             rule_fert.k_thr else 'OK',

        'reason': f"Soglie NPK: {rule_fert.n_thr}/"
                  f"{rule_fert.p_thr}/{rule_fert.k_thr}"
    }
}
```

Listing 4: Extract from code "analyzer.py" - Rule Based Logic

The use of rules ensures immediate Explainability: if irrigation starts, the system can state exactly why (e.g., "Active Threshold: $< 40\%$").

– **Probabilistic Layer (AI/ML Inference – Logistic Regression).** It provides a supervised learning implementation for binary classification. By fitting a `scikit-learn` model to historical data, it captures statistical relationships between features and targets.

```
# Calcolo predizioni (spezzato su piu righe)
p_irr = pipe_irr.handle(df_ai)\
        ["Irrigation_Predicted"].iloc[0]

p_fert = pipe_fert.handle(df_ai)\
         ["Fertilization_Predicted"].iloc[0]

p_en = pipe_en.handle(df_ai)\
       ["Energy_Predicted"].iloc[0]

res_ai['irrigation'] = {
    'status': 'ON' if p_irr==1 else 'OFF',
    'reason': 'AI (LogReg)'
}
res_ai['energy'] = {
    'status': 'ACTIVE' if p_en==1 else 'OFF',
    'reason': 'AI (LogReg)'
}
```

Listing 5: Extract from code "analyzer.py" - AI Inference

This layer introduces an adaptability capability based on historical data, allowing the system to suggest interventions even in borderline situations that might escape static rules.

- **Advice Aggregation & Dispatching** At the end of processing, the Analyzer does not directly actuate hardware changes but publishes a "System Advice" (Advice Packet) to the `system-advice` topic.

```
# 5. Pubblicazione Risultato (System Advice)
advice_packet = {
    'ts': data.get('ts', time.time()),
    'rules': res_rules,
    'ai': res_ai,
    'config': SYSTEM_CONFIG,
    'settings_updated': SETTINGS_UPDATED
}
```

Listing 6: Extract from code "analyzer.py" - Advice Packet

This decoupled design allows downstream consumers (such as the Notification Service or the Dashboard) to receive a complete view of the system state, independently deciding how to present the information to the user (e.g., showing an alert if AI and Rules disagree).

Notification Service, which is realized through the `notification_consumer.py` module, implemented as the terminal observer of the Event-Driven architecture, represents a significant evolution compared to traditional "stateless" alerting systems.

Instead of merely forwarding instant notifications for every single threshold breach—an approach that often generates information redundancy—this microservice adopts an advanced **Stateful** logic. The primary objective is to transform high-frequency event streams into consolidated operational reports, offering the agronomist a clear picture free of background noise.

- **Stateful Logic & Contextualization**: Contextualization capabilities are ensured by the implementation of a **Sliding Window (Circular Buffer)**. Since an isolated alarm possesses little diagnostic value without the data preceding it, the system constantly maintains the last 15 operational "snapshots" in memory. Consequently, when a report is generated, it does not simply record the instant of the anomaly but includes the entire temporal evolution recorded in the buffer, enabling immediate forensic analysis of the triggering causes and system dynamics.

- **Edge Detection & Finite State Machine**: Control of the monitoring flow is entrusted to a **Finite State Machine** operating according to an **Edge Detection** logic. The system is designed to react exclusively to "Rising Edges" (transitions from the OFF state to the ON state), deliberately ignoring stationary states. This architectural choice prevents notification duplication for persistent anomalies, ensuring that the alerting process is triggered only at the exact moment an actuator is activated or a critical threshold is violated.

- **Mitigation of Alert Fatigue**: To mitigate the phenomenon of **"Alert Fatigue"**, typical of complex IoT environments where a single environmental factor can trigger chain reactions across multiple subsystems (e.g., simultaneous irrigation and climate control), a **Temporal Aggregation** logic has been introduced. Upon detection of the first trigger, the service enters an "Active Monitoring" state, initiating a predefined countdown. During this time window, any emerging new alarms do not generate separate transmissions nor reset the cycle, but are accumulated into the current report. Only at the end of the countdown does the system transmit a single cumulative notification, drastically optimizing communication towards the end user.

- **Output Management & Virtual Debugging**: Finally, the architecture provides for flexible output management via the **SMTP protocol**. Reports are structured as multipart MIME payloads to ensure the readability of tabular data on any client. To support a secure and efficient development cycle, the system integrates a **"Virtual Postman" (Debug Server)**: an asynchronous microservice that intercepts local SMTP traffic and renders the content directly to the console. This approach decouples the test environment from the production infrastructure, allowing validation of formatting and sending logic without risking credential locking or the accidental dispatch of real communications.

API Gateway & Vision AI Gateway, which is realized through the `server.py` module, is the third pillar of the backend architecture which serves as the primary synchronous interface between the React frontend and the asynchronous microservices ecosystem.

The **API Gateway** acts as the essential link between the backend and the user's dashboard. One of its main jobs is to solve a communication mismatch: while the internal system uses **Apache Kafka** to move data, web browsers cannot easily "talk" to Kafka. To fix this, the Gateway acts as a **bridge**, converting Kafka messages into a **WebSocket** stream. This allows the system to "push" sensor data and AI results directly to the screen the moment they happen, avoiding the delays found in older methods like constant refreshing (HTTP polling).

A key design choice was how the **Computer Vision** (image analysis) is handled. While most data is processed in the background, we integrated the photo analysis directly into the Gateway. We did this because users expect an immediate result when they upload a photo of a plant. By using a direct "Request-Response" approach instead of sending the photo through a long queue, the system provides instant diagnostic feedback.

The following Sequence Diagram details the synchronous interaction flow.

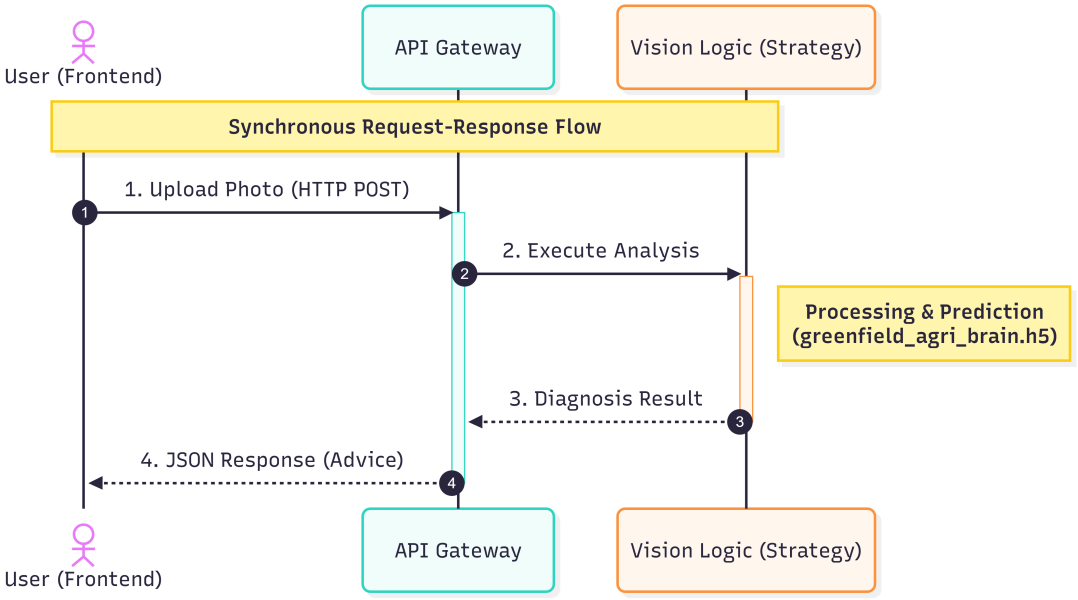

Figure 2: Vision AI Sequence Diagram.

"brain" of this system is a **MobileNetV2** neural network. This specific model was chosen because it is "lightweight"—it is designed to run fast on smaller servers or devices without sacrificing accuracy. It uses a clever mathematical shortcut called "depthwise separable convolutions" to stay efficient and responsive.

To keep the code clean and organized, we used a specific programming layout (the **Strategy Pattern**) that keeps the image-processing math separate from the web-server logic. This makes the system easier to update and maintain. Additionally, the system

doesn't just give a technical diagnosis; it adds a "human" layer by providing actual farming advice based on what the AI sees. Finally, we built the system to be resilient: if the AI model fails to load, the rest of the dashboard (like the sensor monitoring) will continue to work perfectly, ensuring the most critical parts of the app are always available.

Security Service is realized through the user profilation system which is designed to ensure security, modularity, and scalability toward advanced functionalities. The workflow originates at the registration phase, where the system collects essential user data, including email, name, and credentials.

To ensure data integrity, validation is performed via **Zod**, a TypeScript library that enables the definition of rigorous schemas and the enforcement of validation rules, such as email formatting and password complexity (minimum length and alphanumeric requirements).

Once validated, the data is persisted in an **SQLite** database, selected for its lightweight footprint and seamless integration. Database connectivity is managed through **better-sqlite3**, a high-performance driver providing synchronous APIs and support for prepared statements, thereby mitigating SQL injection risks and optimizing performance. The database is configured in **Write-Ahead Logging (WAL)** mode to enhance concurrency between read and write operations. The data schema is defined through a migration script (`migrate.ts`), which establishes the users table with standard integrity constraints, including primary keys, unique email identifiers, and mandatory fields.

For credential management, the system employs **bcrypt**, a library that applies a secure hashing algorithm to passwords to prevent plaintext storage. Following successful registration or authentication, the system issues a **JSON Web Token (JWT)** via the jsonwebtoken library. This token, signed with an application secret (`JWT_SECRET`) to ensure authenticity and integrity, contains the user identifier and facilitates **stateless authentication**, thereby eliminating the need for server-side session persistence.

Every protected request is intercepted by the **requireAuth middleware**, which verifies the token's validity and extracts the subject (`sub`) claim. This identifier is injected into the request context, allowing subsequent handlers to retrieve the corresponding profile. The middleware is implemented within **Express**, the Node.js framework selected for REST API management, and leverages **TypeScript's** static typing to ensure code robustness and maintainability.

The architecture follows a modular design: routes are managed by **Express Router**, business logic is encapsulated within **services** (`authService.ts`), and data access is delegated to **repositories** (`userRepo.ts`). Configuration is centralized via **dotenv**, which manages environment variables for sensitive parameters such as the JWT secret and database paths. To safeguard endpoints against abuse, **express-rate-limit** is integrated to restrict the frequency of requests, thereby reducing the risk of brute-force attacks.

In summary, this profiling mechanism synthesizes modern, secure technologies: TypeScript for type safety, Express for API orchestration, better-sqlite3 for persistence, bcrypt for credential protection, and JWT for stateless authentication, with a clear path toward intelligent profiling via machine learning. This architecture guarantees security, scalability, and extensibility, rendering the system suitable for production environments and future integrations.

3.4 Dashboard

3.4.1 User Interface

- **Authentication Gateway** : the entry point to the **Greenfield Advisor** platform 3 is defined by an authentication module designed to serve as a robust security barrier for protected resources. The interface features a minimalist and streamlined design, centering user interaction on a primary authentication card for credential entry. This component manages the login procedure for authorized personnel, such as agronomists, while providing navigation paths for new user registration.

  Architecturally, the interface visually previews the sidebar navigation—highlighting core features such as Vision AI, Models, and Settings—while strictly subordinating their access to the successful completion of the security handshake managed by the back-end service.

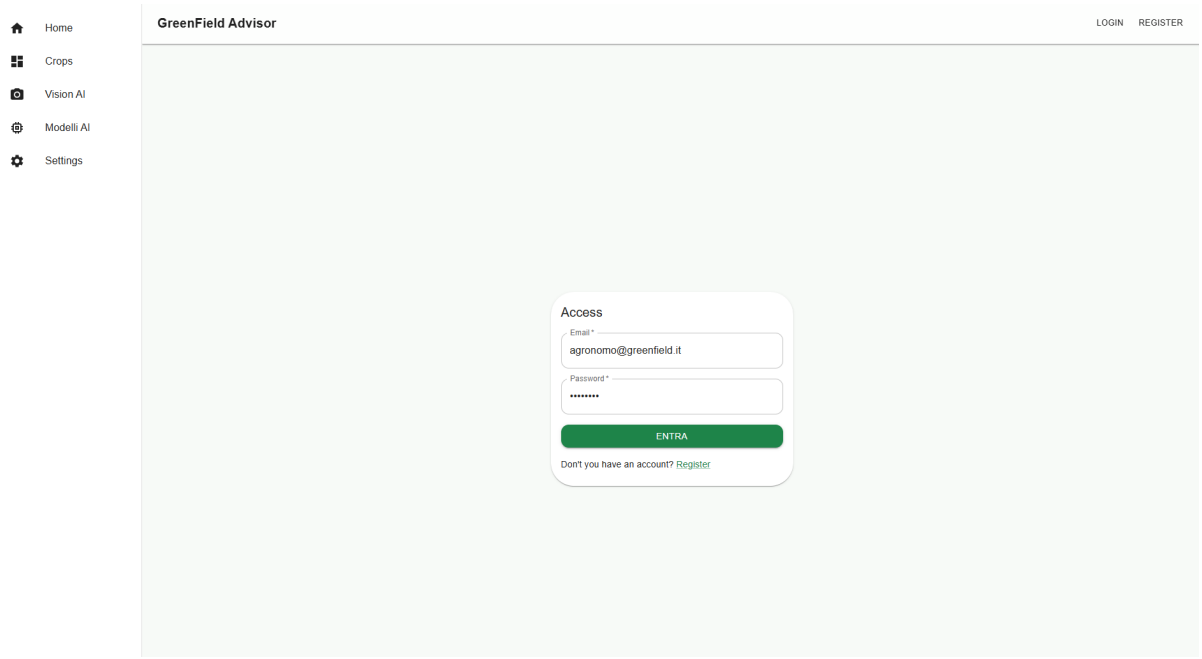

Figure 3: Image of Front-End (Log-In)

- **System Home & Navigation Hub** : Upon successful authentication, the user is directed to the Home Page 4, which serves as the platform's central navigation hub. The layout is organized around a persistent sidebar, ensuring rapid and intuitive access to various functional modules while maintaining a clear separation between control elements and operational content.

  Beyond its role in routing, the interface acts as an "architectural showcase." Through three prominent information cards, the system informs the user of the underlying background technologies: the Event-Driven IoT infrastructure powered by Kafka, the Dual Strategy decision logic, and the Deep Vision engine. Centrally, the "Hero" section provides two immediate calls-to-action (CTAs), streamlining the

workflow toward the core activities: real-time sensor monitoring (Dashboard) and disease diagnosis (Vision AI).

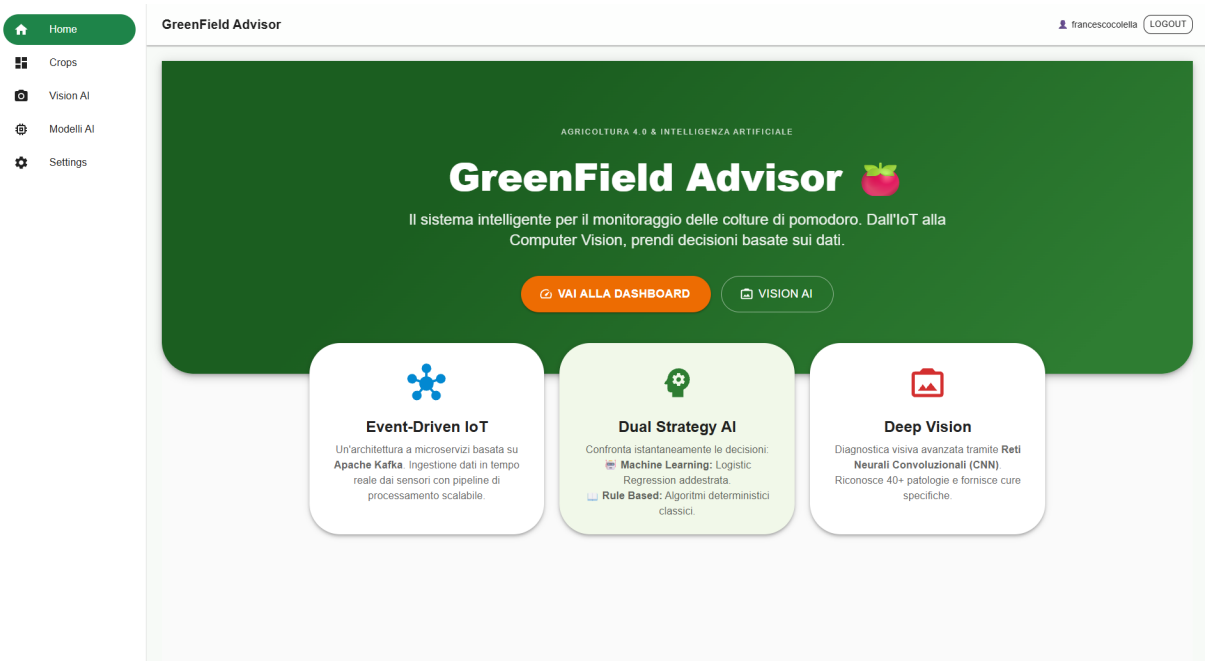

Figure 4: Image of Front-End (Home Page)

- **Real-Time Dashboard & Decision Support** : The platform's operational core is the Dashboard 5, a sophisticated interface engineered for the synchronous visualization of field telemetry streams. The layout follows a hierarchical organization: the upper section provides an immediate overview of Key Performance Indicators (KPIs) and historical trends, rendered through dynamic charts that are updated in real time via the **WebSocket** protocol.

  However, the most significant architectural feature is located in the lower control panel, which monitors the status of system actuators (Irrigation, Fertilization, and Energy). In this section, the interface visually represents the **Strategy Pattern** implemented in the backend. Through a functional toggle mechanism, the operator can switch the displayed decision logic between "AI Model" (probabilistic predictions based on **Logistic Regression**) and "Rule-Based" (deterministic algorithms based on static thresholds). This functionality facilitates an immediate comparison between the two strategies, providing the agronomist with a dual perspective for validating operational decisions and ensuring the transparency of the automated processes.

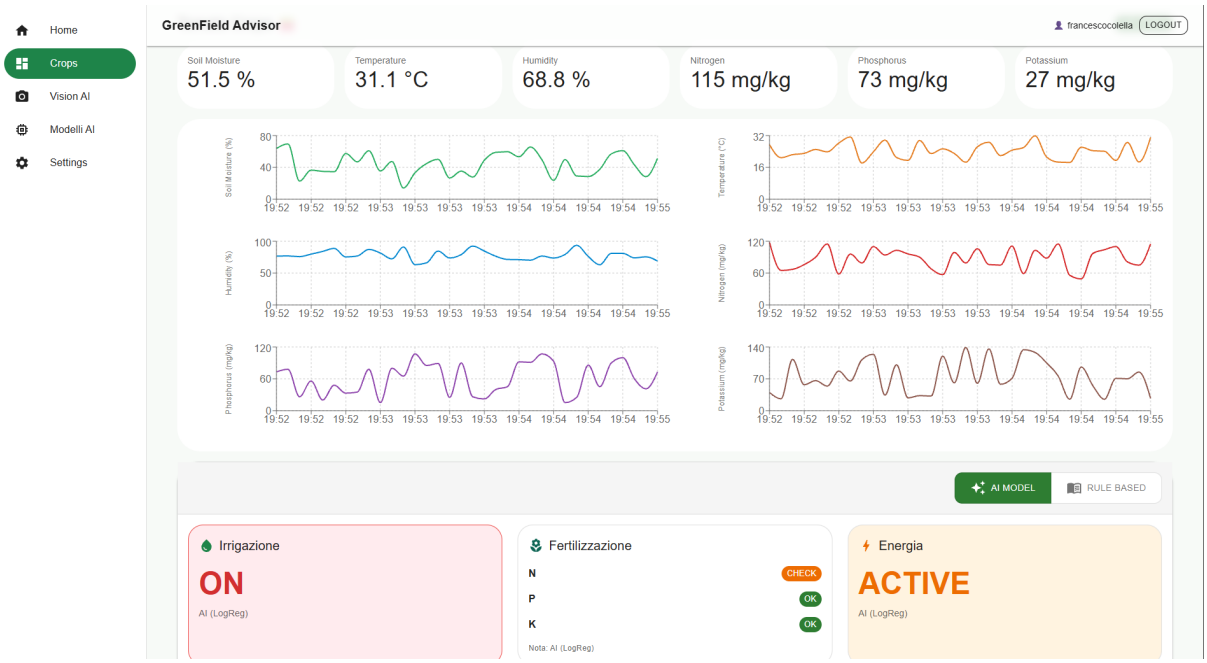

Figure 5: Image of Front-End (Cockpit)

- **Vision AI Specialist** : This module facilitates on-demand interaction with the Computer Vision subsystem, providing an instantaneous diagnostic tool 6 for plant pathology. The interface employs a clean, two-column layout that visually mirrors the underlying **Request-Response** pattern. The left panel is dedicated to input management, allowing the operator to upload and preview photographic assets, while the right panel displays the inference results generated by the backend.

  A fundamental aspect of the system's usability is the **semantic translation** of technical data. Rather than merely presenting a raw classification and its associated confidence index (e.g., 99.9%), the interface enriches the output with structured operational protocols. By incorporating "Immediate Actions" and "Future Prevention" sections, the system converts the neural network's probabilistic predictions into actionable agronomic instructions. This approach effectively bridges the gap between raw data and decision support, providing a comprehensive solution for crop management.

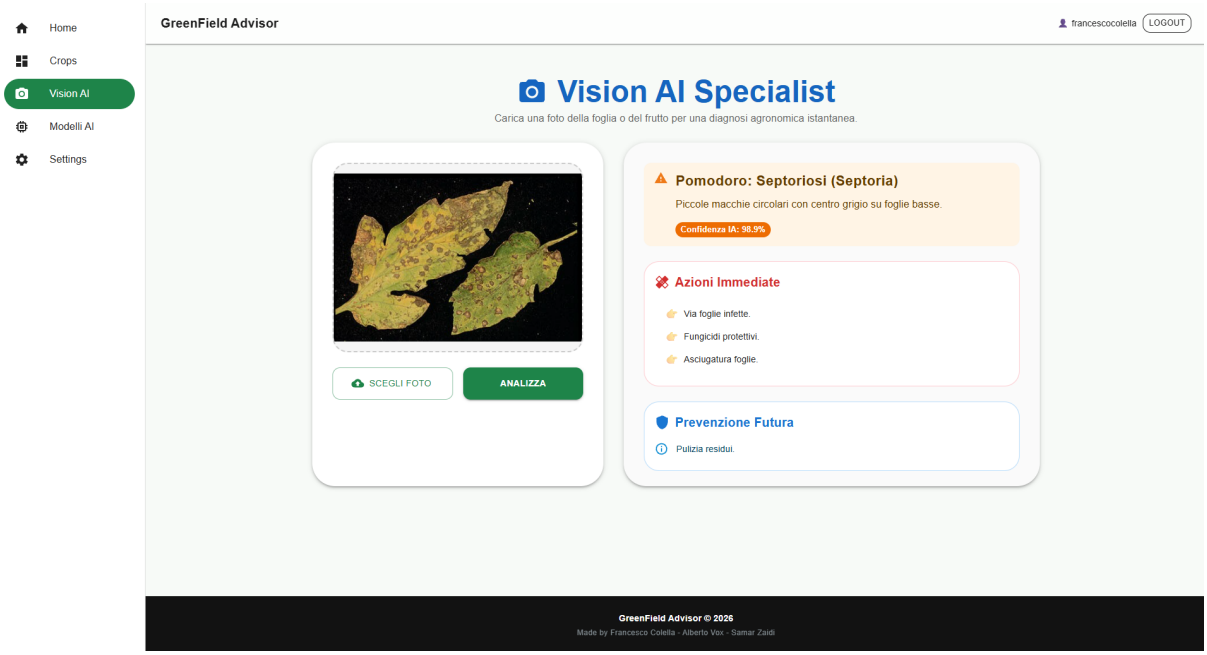

Figure 6: Extract from "Front End (Vision AI Specialist)"

- **System Configuration & Calibration** : The Settings & Configuration module serves as the control interface for the dynamic management 7 of the platform's operational parameters. Utilizing a **tabbed interface** layout, the module enables agronomists to directly modify the backend business logic without requiring source code interventions or system restarts.

  Within the "Agronomic Thresholds" section, the user is provided with granular controls—including sliders for temperature and humidity ranges and specific input fields for **NPK nutrient** levels—designed for the fine-tuning of the IoT engine. Architecturally, this interface functions as more than a local preference menu; it acts as a trigger for **real-time system reconfiguration (hot-swapping)**. Once saved, these modifications instantaneously redefine the threshold values that govern decision-making logic and the triggering of automated alerts, ensuring the system remains responsive to evolving environmental conditions.

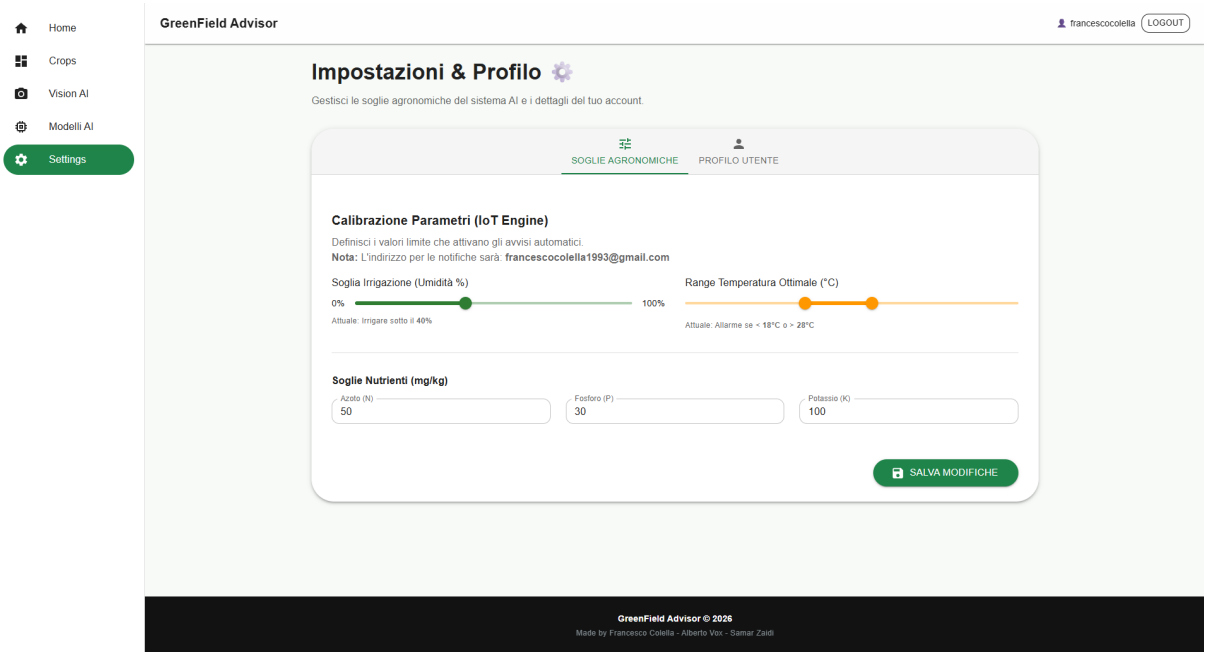

Figure 7: Image of Front-End (Settings)

4 Results

In this section Experimental Results are described and discussed. The study started with the acquisition of an external dataset. For methodological simplicity, the current research focuses exclusively on tomato cultivation; however, the framework is designed to be scalable to other crop varieties in the future. The investigation centered on fundamental parameters influencing tomato growth, with the primary objective of optimizing resource allocation and minimizing waste. Those parameters have been utilized for the treshold setting of irrigation, heating and fertilizing inside the rule-based strategy[15][16][17].

The baseline dataset [18] was enriched through the addition of new features to identify potential inter-variable relationships. This approach stems from the recognition that external datasets often lack the necessary depth to capture the full complexity of the phenomenon under observation. In this context, while the initial dataset provided fundamental data, achieving more accurate profiling necessitated the introduction of derived or calculated variables. This process, known as **feature engineering**, enables the transformation of raw data into more significant informational inputs for the model. These augmented features allow the system to discern patterns that would remain latent if only the original variables were utilized. Specifically, metrics such as ratios, averages, and normalized scores were integrated to ensure data homogeneity and comparability.

The enrichment of the dataset provides two fundamental advantages: first, it enhances the **discriminative capacity** of the model, thereby reducing the risk of misclassification; second, it facilitates the **construction of more holistic and realistic profiles** that reflect not only static attributes but also dynamic behaviors. This is particularly critical in recommendation or segmentation systems, where the quality of the features directly dictates the precision of the predictions.

In summary, the integration of new features was a strategic decision to bolster model robustness and the quality of profiling. This process effectively transformed a basic dataset into a more sophisticated knowledge base, optimized for personalized decision-making.

4.1 Experimental Protocol and Environment

The validation phase was designed to evaluate the efficency of predictive strategies within environmental and agronomic monitoring scenarios. In the absence of a physical sensor network, the data flow was orchestrated via an **Apache Kafka** event-streaming infrastructure utilizing a **Digital-Twin** approach.

A Producer module replayed the time-series datasets, while Consumers—implementing the Strategy pattern—processed observations and published decisions to designated output topics. The evaluation utilized two primary data sources:

- **External Dataset (Agronomic Domain)[18]:** Employed as the primary benchmark for in-sample training and validation.
- **Synthetic Dataset (OOD Stress Test):** A stochastically generated control set used to assess Out-of-Distribution (OOD) robustness and the model's generalization capabilities.

While **accuracy** was adopted as the primary metric for evaluating decision-making tasks (intervention required vs. not required), the experiment also implicitly validated the system's **responsiveness and stability** under continuous data ingestion loads.

4.2 Performance Analysis – Logistic Regression

The results indicate high model efficiency within agronomic domains, contrasted by increased complexity in the energy sector. The following table summarizes the experimental outcomes:

Domain	Benchmark Accuracy	OOD Accuracy	Delta (Δ)
Irrigation	99.7%	99.21%	-0.49%
Fertilization	96.0%	96.13%	+0.13%
Energy (HVAC)	77.0%	78.89%	+1.89%

Table 2: Logistic Regression Performance Table

Domain-Specific Findings:

- **Irrigation and Fertilization :** Both tasks demonstrated excellent performance, approaching near-perfect linear separability. This high accuracy suggests that agronomic variables (soil moisture, pH, and N-P-K nutrients) possess distinct decision boundaries and strong semantic alignment with the prescribed intervention thresholds.
- **Energy Management :** The lower performance (77.0%) indicates that Logistic Regression is insufficient to capture the dynamic nature of energy consumption. Factors such as thermal inertia and occupancy variables introduce non-linearities that necessitate more complex models or extensive feature engineering.

4.3 Critical Discussion and Implications

The marginal performance decay observed in the OOD set confirms the stability of the learned decision boundaries. However, the analysis highlights several critical points for system evolution:

- **Overfitting and Bias Risk :** The near-total accuracy in irrigation may stem from minimal label noise in the training set. Future validation should involve "noisy" real-world data to ensure robustness.
- **Explainability :** The choice of logistic regression facilitates the extraction of odds ratios, allowing domain experts (agronomists and energy managers) to validate the statistical weight assigned to each variable.
- **Simulation Constraints :** While the event-driven architecture is structurally sound, the simulation does not account for physical hardware variables such as network jitter, latency, or sensor drift.

4.4 Visual System

Experimental validation of the API Gateway & Vision AI module focused on integrating synchronous user interaction with heavy Deep Learning workloads. The selection of **MobileNetV2** was pivotal, providing an optimal balance between accuracy and computational efficiency. This choice ensured inference speeds remained within the limits of the Request-Response paradigm, preventing the performance bottlenecks typical of more complex architectures.

Architecturally, the **Strategy Pattern** successfully decoupled HTTP lifecycles from tensor processing, allowing the Gateway to handle image analysis without compromising concurrent WebSocket telemetry. Furthermore, functional testing confirmed the effectiveness of the semantic layer, which successfully translated raw neural network probabilities into structured, actionable agronomic protocols.

5 Conclusions

This research culminated in the design and implementation of Greenfield Advisor, a modular precision agriculture framework. The project demonstrates how the adoption of modern architectural patterns can effectively address the inherent complexities of IoT-based systems.

From a Software Engineering perspective, the primary contribution lies in the validation of the Event-Driven paradigm. The orchestration via Apache Kafka ensured effective decoupling between data ingestion layers (Producers) and decision engines (Consumers). This enabled the system to manage high-frequency telemetry streams without bottlenecks while providing native horizontal scalability. Furthermore, the use of a containerized microservices architecture (Docker) confirmed the robustness of the solution, ensuring process isolation and deployment environment reproducibility.

A distinctive feature of the project is the implementation of a Dual Strategy decision logic. By combining the Strategy and Chain of Responsibility patterns, the system harmonizes the rigor of rule-based algorithms with the predictive power of Machine Learning. This provides agronomists with a flexible and transparent decision-support tool.

Finally, the integration of the Computer Vision subsystem directly within the API Gateway illustrates a successful balance between intensive computational loads (Deep Learning) and the low-latency requirements essential for a responsive user experience.

References

[1] R. Finger, "Digital innovations for sustainable and resilient agricultural systems," *European Review of Agricultural Economics*, vol. 50, no. 4, 2023.

[2] M. N. Reza *et al.*, "Trends of soil and solution nutrient sensing for open field and hydroponic cultivation in facilitated smart agriculture," 2025.

[3] S. Mansoor *et al.*, "Integration of smart sensors and iot in precision agriculture: trends, challenges and future prospectives," 2025.

[4] T. O'Donoghue, B. Minasny, and A. McBratney, "Digital regenerative agriculture," *Sustainable Agriculture*, 2024.

[5] M. Awais *et al.*, "Advancing precision agriculture through digital twins and smart farming technologies: A review," *Agriengineering*, 2025.

[6] A. C. Tagarakis *et al.*, "Digital twins in agriculture and forestry: A review," *Sensors*, 2024.

[7] A. Chole, A. Jadhav, and V. Shinde, "Vertical farming: Controlled environment agriculture," *Just Agriculture*, 2021.

[8] F. Tian, "An agri-food supply chain traceability system for china based on rfid & blockchain technology," 2016.

[9] Y. Rouphael and G. Colla, "Biostimulants in agriculture," *Frontiers in Plant Science*, 2019.

[10] A. Boyd *et al.*, "Cross-cutting concepts transform agricultural research," *Frontiers in Sustainable Food Systems*, 2013.

[11] R. Laigner *et al.*, "An empirical study on challenges of event management in microservice architectures," 2025.

[12] A. Ghosh, "Event-driven architectures for microservices: A framework for scalable and resilient rearchitecting of monolithic systems," *International Journal on Science and Technology (IJSAT)*, 2025.

[13] S. Kumar, "Event-driven microservices architectures: Principles, patterns and best practices," *World Journal of Advanced Engineering Technology and Sciences*, 2025.

[14] M. S. Silva *et al.*, "Guidelines for the application of event driven architecture in micro services with high volume of data," in *Proceedings of the 27th International Conference on Enterprise Information Systems (ICEIS 2025)*, vol. 2, 2025.

[15] R. R. Shamshiri *et al.*, "Review of optimum temperature, humidity, and vapour pressure deficit for microclimate evaluation and control in greenhouse cultivation of tomato: a review," *International Agrophysics*, 2018.

[16] University of California, *Tomato Pest Management Guidelines*. UC-IPM. `https://ipm.ucanr.edu/agriculture/tomato/fertilization/#gsc.tab=0`.

[17] Demetra, *Interpretazione delle analisi dei substrati*. https://www.demetralab.it/substrati-di-coltivazione-interpretare-i-risultati/.

[18] R. Kasera and T. Acharjee, "A comprehensive iot edge based smart irrigation system for tomato cultivation," 2024.