# Design Model - PlantPal

| Team Name: SproutLab | |
|---|---|
| **Student Number** | **Name** |
| 2352280 | Hengyu Jin |
| 2352609 | Qi Lin |
| 2353409 | Baoyi Hu |
| 2352288 | Sirui Da |

## 1. Overview & Implementation Platforms

### 1.1 Design Progress Overview

Based on the "Ports and Adapters (Hexagonal)" architecture defined in the analysis model, this phase focused on completing the mapping from the logical view to the physical deployment view. The core design objectives are centered on addressing the Key Performance Indicators (KPIs) defined in the *SRS*, specifically "Local Recognition Response < 2s" and "First-Week User Retention Rate." The design strategy adopts frontend-backend separation and microservice componentization to ensure the stability of core business rules (such as care schedule calculations) while isolating volatile AI model dependencies.

### 1.2 Implementation Platforms & Frameworks

**Mobile Client (Client): Flutter**

- **Selection Rationale:** Meets the requirement for a unified codebase across iOS and Android. Leveraging the high-performance rendering capabilities of the Skia engine, it supports complex "Look-alike Contrast Views" and fluid animation interactions, ensuring a user experience where "Key Path Clicks ≤ 3."

**Backend Core: Java / Spring Boot**

- **Selection Rationale:** Hosts the Application Service Layer. It utilizes the mature Dependency Injection (DI) and Aspect-Oriented Programming (AOP) features of the Spring ecosystem to manage complex business rules and transactional consistency within the `CareService` and `DossierService`.

**AI Service Layer (AI Microservice): Python / FastAPI**

- **Selection Rationale:** Operates as an independent microservice to encapsulate calls to Cloud VLMs (e.g., GPT-4o / Gemini Vision). Python offers the richest ecosystem of vector calculation libraries (NumPy/Pandas), facilitating Embedding generation and similarity retrieval for the `PlantKnowledgeBase`.

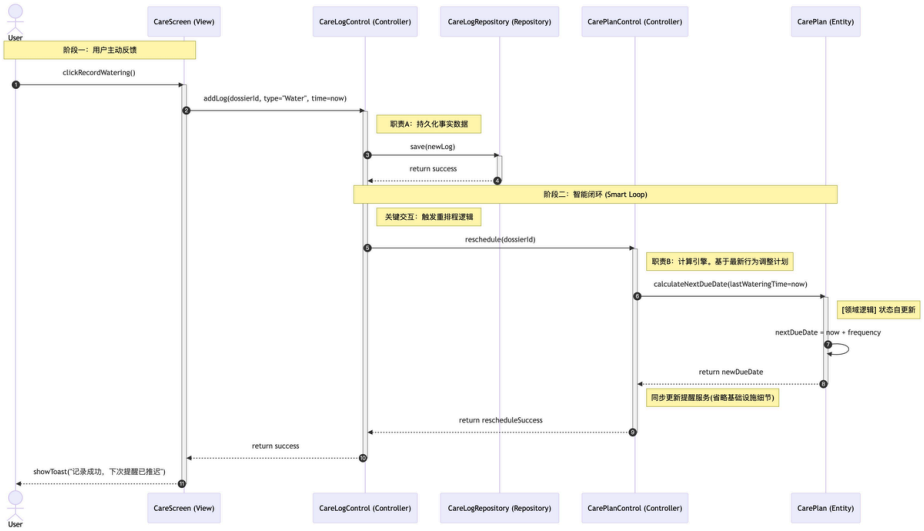
**Data Persistence:**

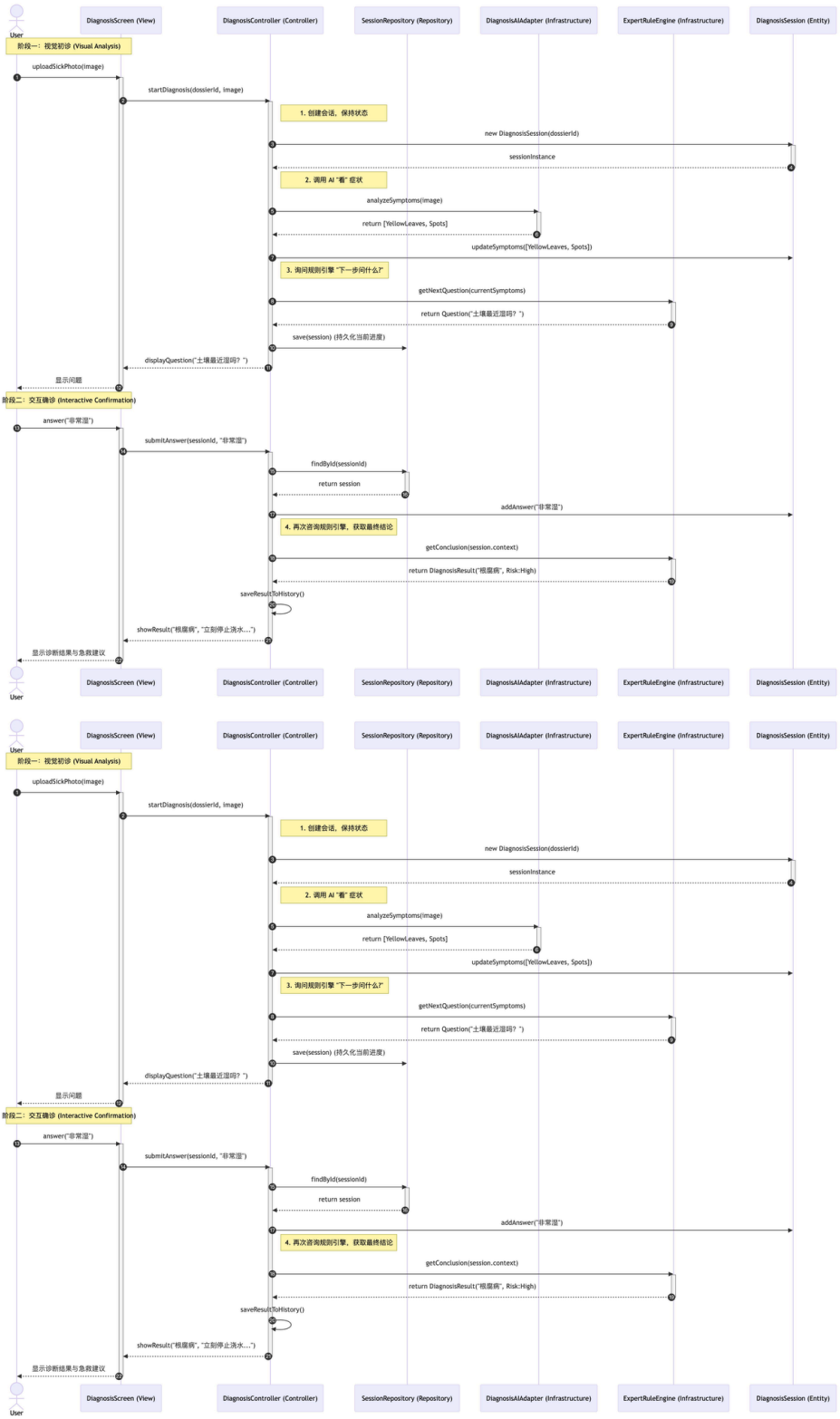
- **PostgreSQL:** A relational database used to store structured data for `PlantDossier`, `CarePlan`, and `CareLog`, ensuring strong transactional consistency.
- **Milvus / Pgvector:** A vector database used to store plant feature vectors, supporting Top-k nearest neighbor search for high-dimensional data.

## 2. Architecture Refinement

### 2.1 Platform-dependent Architecture

We mapped the logical hexagonal architecture to a physical deployment architecture. The system adopts a typical Client-Server (C/S) pattern combined with microservice principles.






**Architecture Description:**

- **Mobile Node:** Contains the Flutter App and local SQLite (used for offline caching and local recognition in < 2s).
- **Cloud Gateway:** A unified API entry point responsible for authentication and rate limiting.
- **Core Business Service:** Hosts the Dossier and Care core domains, calling the AI Service via RPC/REST.
- **AI Service:** An independent node responsible for interfacing with external Large Model APIs to implement the "Hybrid Inference" strategy defined in the analysis model.

## 2.2 Subsystems and Interfaces

Based on the decomposition in the analysis model, the system comprises the following core subsystems and their external contracts:

| Subsystem Name | Responsibility Summary | Core Interface | Dependencies |
| --- | --- | --- | --- |
| **Identification Subsystem** | Handles image recognition, feature extraction, and confusion discrimination. | IIdentificationService | Depends on CloudVLM |

| | | | |
|---|---|---|---|
| **Care Subsystem** | Care rule calculation, schedule dispatching, and logging. | ICarePlanService, ICareLogService | Depends on WeatherService |
| **Dossier Subsystem** | Aggregate Root management (CRUD) and maintenance of dossier integrity. | IDossierRepository | Referenced by CareSubsystem |
| **Diagnosis Subsystem** | Multi-turn consultation and risk assessment. | IDiagnosisSession | Depends on IdentificationSubsystem |

### 2.3 Interface Specification Detail

**Scenario:** The system needs to call an external weather service to adjust the care plan (Corresponding to UC5 Intelligent Scheduling).

- **Interface Name:** `IWeatherProvider` (Port defined in the Application Layer)
- **Implementation Class:** `OpenMeteoAdapter` (Adapter defined in the Infrastructure Layer)
- **Interaction Protocol:** HTTPS / REST
- **External System:** Open-Meteo Public API

**Interface Definition (Java Pseudo-code):**

代码块

```
/**
 * External Weather Service Port
 * Follows DIP (Dependency Inversion Principle); the business layer depends only on this interface.
 */public interface IWeatherProvider {

    /**
     * Retrieves current and future weather briefs for specific coordinates.
     * @param latitude
     * @param longitude
     * @return WeatherDTO (Includes temperature, humidity, precipitation probability)
     * @throws ExternalServiceException Thrown when the external API is unavailable
     */WeatherDTO getLocalWeather(double latitude, double longitude) throws ExternalServiceException;
}

// DTO Definitionpublic class WeatherDTO {
    public float currentTemp;
    public float humidity;
    public boolean isRaining;
    public List<String> weeklyForecast; // Used to adjust the watering plan for the upcoming week
}
```

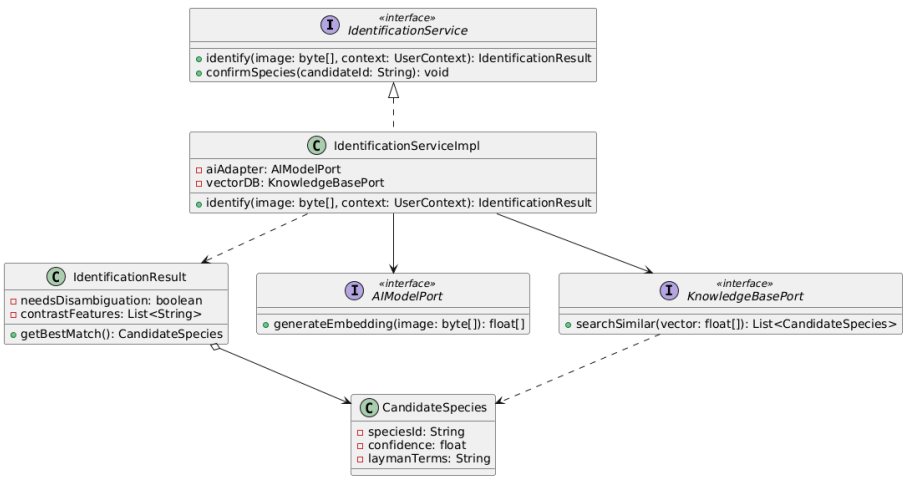
## 3. Detailed Subsystem Design

**Selected Subsystem: Identification Subsystem**

- **Selection Rationale:** This subsystem acts as the traffic entry point and possesses the most complex logic, involving cloud/local hybrid inference and specific business logic for "Look-alike Disambiguation."

### 3.1 Interface Design

The core service interface exposed by this subsystem for use by the `DossierController`.



### 3.2 Detailed Specifications

- **Method:** `identify(byte[] image, UserContext context)`
- **Input:**
  - `image`: Raw image data captured by the user.
  - `context`: Includes `geoLocation` (to assist in determining local species) and `userPreferences`.
- **Output:** `IdentificationResult` object.
- **Key Logic:** If the confidence difference between the Top-1 and Top-2 candidates is < 15% (referencing the Analysis Model), the `needsDisambiguation` field will be flagged as `true` and populated with `contrastFeatures` (e.g., "Serrated Leaf Margin" vs. "Entire Margin"), which the frontend uses to render the contrast view.

## 4. Design Mechanisms

We selected key mechanisms identified in the analysis model for detailed implementation design: persistence, AI service degradation, and asynchronous processing.

### 4.1 Mechanism 1: Persistence & ORM

- **Analysis Origin:** The Analysis Model defines `PlantDossier` as an Aggregate Root.
- **Design Challenge:** Must ensure lifecycle consistency among Plant `Dossier`, `CarePlan`, and `CareLog`.
- **Solution:**
  - **ORM Mapping:** Use Spring Data JPA (Hibernate implementation).
  - **Composition Implementation:** In the `PlantDossier` class, associate `CarePlan` using `@OneToOne(cascade = CascadeType.ALL, orphanRemoval = true)`.
  - *Significance:* This means that when a `PlantDossier` is deleted, the database automatically cascade-deletes the corresponding `CarePlan` and `CareLog` records. This strictly conforms to the UML solid diamond composition relationship, preventing orphan data.
  - **Lazy Loading Strategy:** Use `FetchType.LAZY` for the `CareLog` list. Since log volume grows indefinitely over time, data is paginated and loaded only when the user clicks the "Timeline" view, meeting the KPI of < 60s for initial screen loading.

### 4.2 Mechanism 2: AI Adaptation & Circuit Breaker

- **Analysis Origin:** "Weak Network Degradation" requirements in the Analysis Model and reliability indicators in the SRS.
- **Design Challenge:** Cloud VLM (Vision-Language Model) calls may time out or become unavailable; this must not block the user's main flow.
- **Solution:**
  - **Adapter Pattern:** Define a unified `AIModelPort`. Implement `CloudVLMAdapter` (calling GPT-4) and `LocalTFLiteAdapter` (calling local MobileNet).
  - **Circuit Breaker Pattern:** Introduce the `Resilience4j` library.
  - **Configuration:**
    - `failureRateThreshold`: 50% (Circuit breaker opens if failure rate exceeds 50%).
    - `waitDurationInOpenState`: 10s (Wait 10 seconds before retrying after circuit opens).
    - `timeoutDuration`: 5s (Meets KPI and acts as a hard timeout setting).
  - **Fallback Flow:** When `CloudVLMAdapter` throws a `TimeoutException` or the circuit breaker is open, the system automatically catches the exception and switches to call `LocalTFLiteAdapter`.
  - *Significance:* Although the local model cannot provide "detailed distinguishing features," it can return the Top-1 Genus-level classification, ensuring users can complete the basic dossier creation process, satisfying the principle of "Minimum Viability."

### 4.3 Mechanism 3: Asynchronous Identification & State Tracking

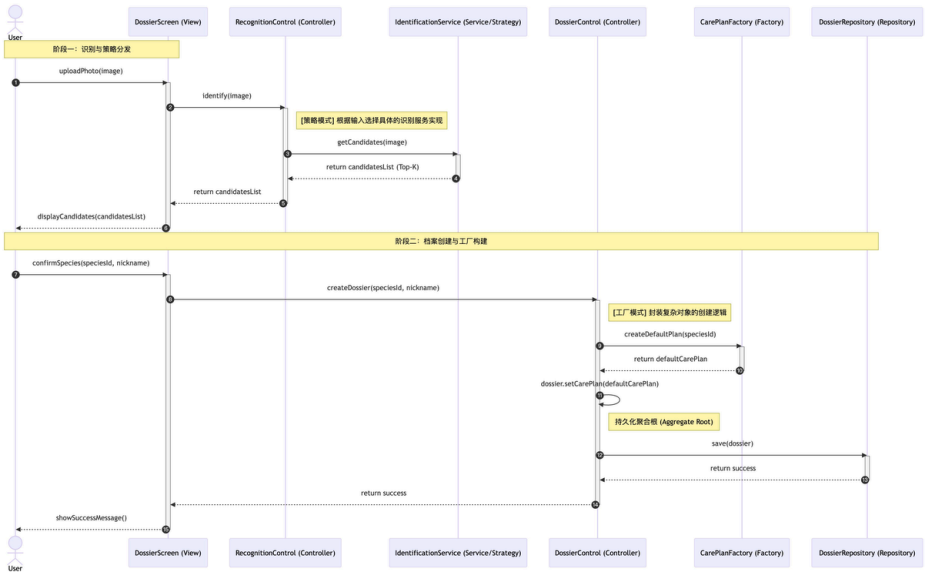
- **Analysis Origin:** Resolving the conflict between the long latency (> 2s) of cloud VLM identification logic and frontend interaction fluidity.
- **Design Challenge:** Image analysis involves high-dimensional vector retrieval and large model inference, averaging 3-5s. Synchronous waiting would block the main thread and cause user anxiety.
- **Solution:**
  - **Non-blocking Call:** The identification interface of `IdentificationService` adopts an asynchronous mode. The backend immediately returns a `JobTicketID`, and the frontend enters an "Analyzing" UI state.
  - **Future/Promise Implementation:** Use `CompletableFuture<IdentificationResult>` in the Java backend to handle Cloud APIs. Even if the logic appears sequential in the Sequence Diagram for UC3 (Create Dossier), the underlying implementation uses callbacks or WebSocket to push results, avoiding I/O thread blocking.
  - **Client-side Optimization:** When the identification process exceeds 2s, the frontend automatically triggers a "Knowledge Tips" carousel to compensate for perceived latency.

## 5. Use Case Realizations

### 5.1 Use Case 1: Create Plant Dossier (UC3)

This use case demonstrates the entire process from the user uploading a photo to creating a complete plant dossier. At the design level, we adopted the following two key design patterns to enhance system flexibility and maintainability:

1. **Strategy Pattern:** During the identification phase, `RecognitionControl` dynamically selects a specific identification strategy (defined by the `IdentificationService` interface) based on the input type (image or text), decoupling the identification logic from business control.
2. **Factory Pattern:** During the dossier creation phase, `CarePlanFactory` is introduced. Since different plants (e.g., Succulents vs. Ferns) vary significantly in their initial care plans (involving complex object construction for watering frequency, light requirements, etc.), the factory encapsulates this creation logic to prevent `DossierControl` from becoming bloated.



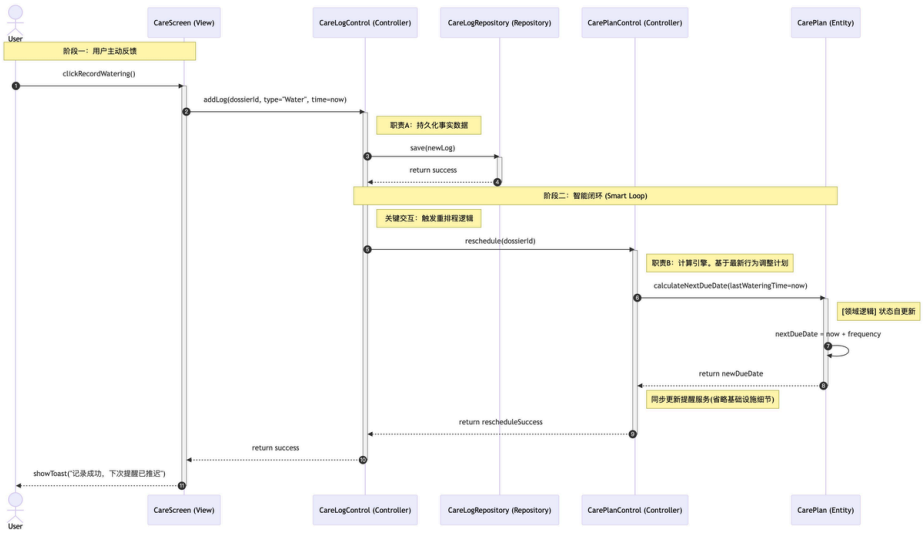

**Design Rationale:**

- **RecognitionControl (Controller):** As the coordinator of the identification flow, it does not contain specific image processing algorithms but calls `IdentificationService`. This adheres to the Single Responsibility Principle (SRP).
- **CarePlanFactory (Factory):** This is a critical design decision. Creating a `CarePlan` object requires not only the `speciesId` but also queries regarding the species' attributes (e.g., drought tolerance, growing season). If this logic were entirely within `DossierControl`, it would lead to excessive coupling. The factory class is specifically responsible for "manufacturing" default plans that conform to the species' characteristics.
- **DossierRepository (Repository):** Responsible for data persistence. Note that we save the `Dossier` (Aggregate Root); according to Domain-Driven Design (DDD) principles, it cascades the saving of the internal `CarePlan`, ensuring data consistency.

## 5.2 Use Case 2: Proactive Care Recording (UC5-2)

This use case demonstrates the design concept of an "Intelligent Feedback Loop." Unlike traditional fixed-cycle reminders, when a user proactively completes care (e.g., watering early), the system does not simply log the action but triggers the core **Rescheduling Mechanism**, automatically postponing the next reminder to avoid ineffective disturbances.

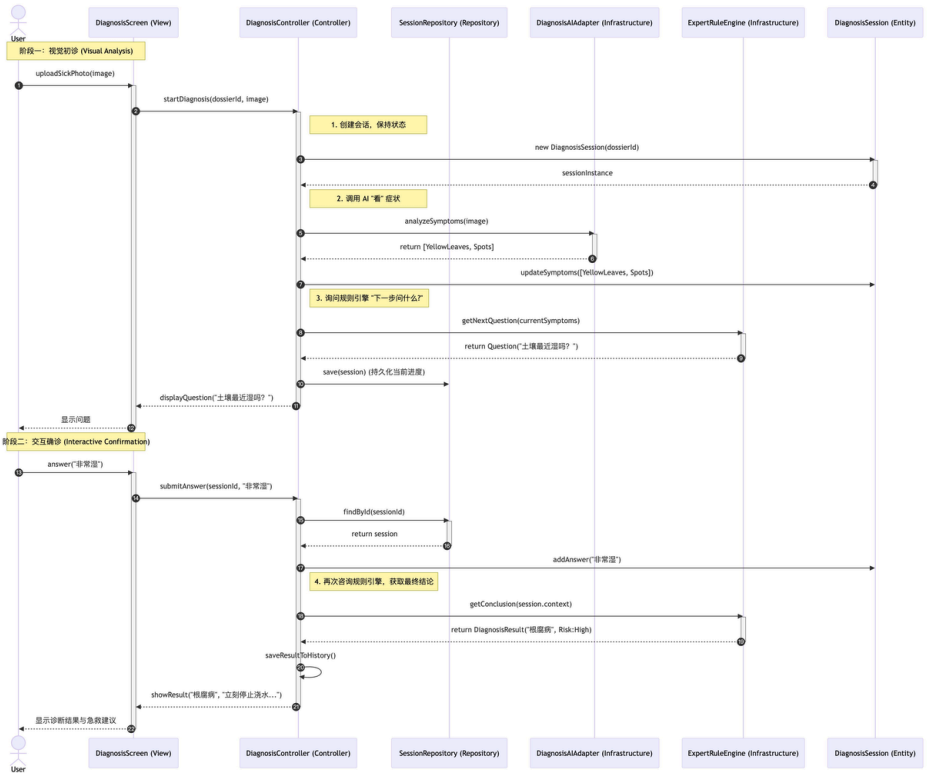
In terms of design, we ensured high system cohesion and low coupling by clearly separating the responsibilities of the "Logger (Log Control)" and the "Planner (Plan Control)."




## 5.3 Use Case 3: Plant Health Diagnosis (UC4)

This use case demonstrates how the system solves complex plant pathology diagnosis problems via **"Human-in-the-loop"** collaboration.

Unlike simple "Snapshot Identification," diagnosis is a multi-step process (Upload -> Initial Analysis -> Follow-up Confirmation -> Proposed Solution). To manage the context during this process (e.g., what the user answered in the previous question), we introduced the **Session Pattern** in the design.




**Design Rationale:**

- **Session Pattern:**
  - *Why is it needed?* Diagnosis is not a one-off HTTP request; it may involve 3-5 rounds of Q&A. If the user switches out of the App to reply to a message, they must be able to resume progress upon return.
  - *Implementation:* We designed the `DiagnosisSession` entity to act as the "memory." It temporarily stores photo analysis results and the user's previous answers. `DiagnosisController` does not store state directly but restores the session from the `SessionRepository` for each request, adhering to the stateless server principle.
- **Decoupling Vision & Logic:**
  - We split the diagnostic capability into two adapters:
  - **DiagnosisAI Adapter (Vision Layer):** Responsible for "Seeing." It handles converting images into structured data (e.g., "Leaves are yellowing") but does not understand pathology.
  - **ExpertRuleEngine (Logic Layer):** Responsible for "Thinking." It does not need to see the image; it infers conclusions (e.g., "That is root rot") based on structured data ("Yellow leaves" + "Wet soil").
  - *Benefit:* This design is highly flexible. If pathology rules need upgrading (e.g., correcting the features of a specific disease) in the future, we only need to update the JSON configuration of the rule engine without retraining expensive vision AI models.
- **Defensive Design:**
  - The sequence diagram illustrates `saveResultToHistory()` . Even if the App crashes, the diagnostic result is saved as part of the `CareLog` , ensuring data reliability.

## 6. Architectural Styles & Critical Decisions

This chapter articulates the core architectural principles and critical technical decisions for the PlantPal system. These decisions aim to resolve the core conflicts presented in the SRS (Software Requirements Specification): the balance between the need for high-precision AI identification and the requirement for low-latency/offline availability on mobile devices, and the conflict between the volatility of business rules and the stability of software release cycles.

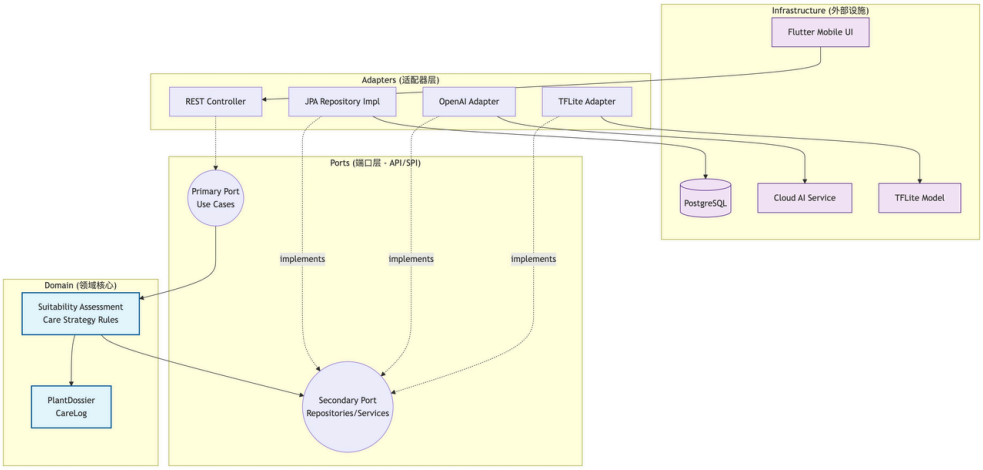
### 6.1 Architectural Style: Ports and Adapters (Hexagonal Architecture)

We have adopted the Hexagonal Architecture (also known as the Ports and Adapters pattern).

**Decision Description:** We plan to partition the system into a "Core" and a "Shell."

- **Core (Domain):** Contains only pure care logic (e.g., "Determining if this plant is suitable for the current environment"). It contains absolutely no technical code—no SQL, no HTTP requests, and no AI SDKs.
- **Shell (Infrastructure):** All external tools (databases, mobile UI, OpenAI interfaces, Weather APIs) are treated merely as plugins.
- **Connection Mechanism:** The Core defines interfaces (**Ports**), and the Shell plugs in via **Adapters** to implement functionality.

**Decision Rationale (Why?):** This is primarily to cope with the rapid iteration of AI technology. While the current version may use OpenAI for identification, if a cheaper visual model fine-tuned for plants appears next month, or if we need to swap models, the Hexagonal Architecture allows us to simply write a new **Adapter** to replace the old one without modifying any core care business code. This ensures that our core business logic remains stable while the technical implementation remains flexible.




## 6.2 Hybrid AI Inference

**Decision Description:** We are not relying solely on the cloud. We are adopting a combined strategy of **"Cloud Large Models + Local Small Models."**

- **Online (Cloud):** Invokes the Cloud VLM (Vision-Language Model). It not only identifies the plant as "Monstera" but also provides detailed insights, such as "leaves are yellowing due to nitrogen deficiency."
- **Offline/Weak Network (Local):** Automatically switches to the local TFLite model on the device. It ensures a quick response identifying "This is a Monstera" and allows the user to save it to the dossier immediately.

**Decision Rationale:** To address the **"signal black hole"** issue in flower markets. The most frequent usage scenarios for identification are flower markets or balcony corners, where network signals are often unstable. If relying entirely on the cloud, a 10-second loading spinner would ruin the user experience. Hybrid inference guarantees that users can **"start using it immediately"** in any network environment, syncing detailed data silently in the background once connectivity improves.

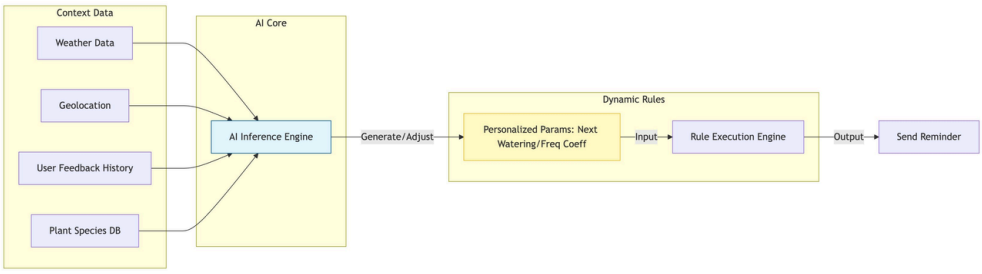
## 6.3 Critical Decision: AI-Driven Adaptive Policy

**Decision Description:** We are discarding "one-size-fits-all" static rule tables. Care Rules within the system are no longer manually maintained JSON files managed by operations staff; instead, they are generated and dynamically maintained in real-time by an **AI Agent** based on multi-dimensional data.

- **Input Layer:** Plant species characteristics + User micro-environment (Geolocation, current weather, indoor/outdoor) + User behavioral feedback (e.g., "Just watered" or "Leaves are wilting" recorded via `JournalEntry`).
- **Processing Layer:** `CarePlanControl` calls the LLM or inference model to generate personalized `CarePlan` parameters specific to this individual plant (e.g., correcting a default `interval: 7` to `interval: 5`).
- **Execution Layer:** The Rule Engine executes these parameters dynamically set by the AI.

**Decision Rationale (Why?):** To solve the challenge of care failure caused by **"environmental differentiation."** There is no standard answer for plant care; the environment is the single largest variable.

- **Scenario Pain Point:** Taking a Pothos as an example: the watering frequency differs drastically between a dry, heated room in Beijing and a humid "Back-to-South" (Huinan) weather environment in Guangzhou. Manually maintained rule tables cannot exhaust all combinations of "Region × Season × Housing Type."
- **AI Value:** Through dynamic AI calculation, the system achieves **"Hyper-personalization for every plant."** When the API reports "continuous rain for the next week," the AI automatically modifies the user's care parameters to postpone watering reminders, truly achieving an understanding of the environment that surpasses even the user's.




## 6.4 Critical Decision: Human-in-the-Loop Architecture

**Decision Description:** Addressing the inherent flaw of **"AI Hallucination,"** we reject a strategy of singular AI augmentation. Instead, we solidify "Human Confirmation" as a high-priority component within the architectural layer.

**Engineering Implementation:**

- **Uncertainty Interceptor:** Introduce a confidence threshold within the AI Service Layer. When the Top-1 confidence is < 85% or the difference from the Top-2 is < 15%, the architecture **forcibly routes** the process to "Human Intervention Logic."
- **Contrast View Injection:** The system does not merely return a single result; it extracts conflicting features of two suspected species (e.g., "Does the petiole have a groove?") and pushes them to the frontend, guiding the user to close the final 1% of the loop.

**Decision Rationale:** To acknowledge the unreliability of AI. By using engineering means to downgrade "Machine Recognition" to **"Machine Suggestion + Human Decision,"** we eliminate erroneous care plans caused by hallucinations.

## 7. Addressing Non-Functional Requirements (NFR)

### 7.1 Performance: Millisecond-Level Local Recognition

**Requirement:** In local/weak network environments, identification response time must be < 2s.

**Design Strategy: Local Vector Matching with LRU Cache.**

- **Core Concept:** We must not force users to wait for network loading. We assume that the plants users encounter most frequently are often the top dozens of popular varieties (e.g., Pothos, Money Tree).
- **Implementation:**
  - **Hotspot Pre-loading:** Upon the App's initial launch, the Embedding feature vectors (the "digital fingerprints") of the Top-50 common indoor plants are pre-loaded into an in-memory **LRU Cache** (Least Recently Used Cache).
  - **Local Fast Comparison:** When a user takes a photo, the on-device TFLite lightweight model immediately calculates the feature vector of the image.
  - **Instant Hit:** The system prioritizes vector similarity calculation within the LRU Cache (taking only milliseconds). If the similarity is high (> 0.9), the result is returned immediately without networking; if there is no hit, a network request is then initiated.
- **Result:** For common plants, users experience a **zero-latency** "shutter-to-result" experience.

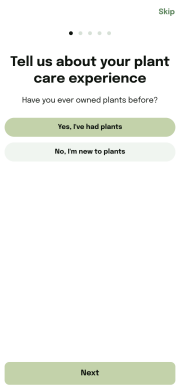
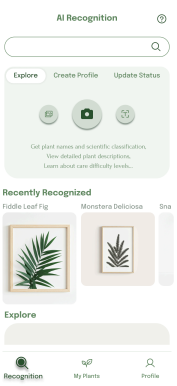
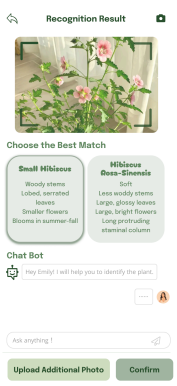
### 7.2 Reliability: Offline-First Data Synchronization

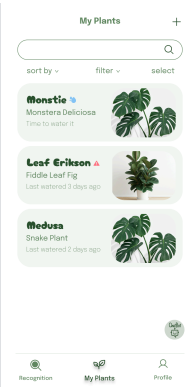
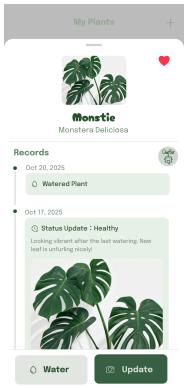
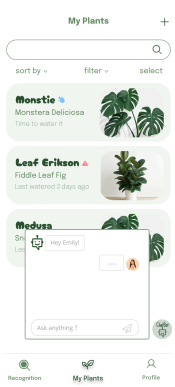
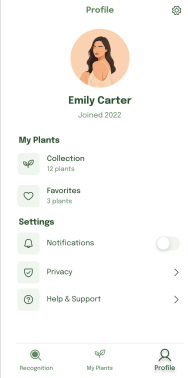
**Requirement:** Support offline data saving and automatic retry uploads within 10s of network recovery.

**Design Strategy: "Local-First" Architecture & SyncManager Component.**

- **Core Concept:** Never block the user from recording due to a lack of internet. We treat the local mobile database (SQLite) as the temporary **"Single Source of Truth,"** with the network serving merely as a backup channel.
- **Implementation:**
  - **Optimistic UI:** After the user clicks "Save," the system writes directly to the local SQLite database, and the UI immediately shows "Saved Successfully" (without waiting for a server response), ensuring operational fluidity.
  - **Pending Sync Queue:** Simultaneously, this record is marked as `dirty` (pending sync) and placed into a persistent task queue.
  - **Smart Sync (SyncManager):** We designed a `SyncManager` component that registers listeners for the system's network state. Once network recovery (Wi-Fi or 4G) is detected, a background **Worker Thread** immediately launches to batch the data from the queue and upload it to the cloud PostgreSQL, achieving final consistency.

## 8. Progress on Prototyping

| page | UI snapshots |
| --- | --- |
| **Registration / Onboarding Page** |  UI 1.1 |
| **AI Recognition Page (Home / Explore)** |  UI 2.1  UI 2.2  UI 2.3  UI 2.4  UI 2.5 |
| **My Plants Page** | |

| | | | |
|---|---|---|---|
| |  UI 3.1 |  UI 3.2 |  UI 3.3 |
| **Personal Center Page** |  UI 4.1 | | |

### Updated Prototype UI 2.5:

When the difference in AI confidence scores for plant identification results is **less than 15%**, the system determines that the result is uncertain. Instead of providing a single conclusion, it enters a **"Human Confirmation Guidance Mode."** In this interface:

- The system displays the **two most likely plants side-by-side**, clearly listing their key morphological features and care characteristics to assist the user in making a visual comparison.
- Users can manually select the target plant that best matches the physical features based on this description.

**Integrated ChatBot:** A ChatBot is built into the interface. Users can converse with the ChatBot to further inquire about the differences between the two plants and ask for identification points to aid their judgment.

**Two Confirmation Methods:**

1. **"Upload Additional Photo":** Upload more photos for the system to perform further analysis and confirmation.
2. **"Confirm":** Select the more similar option among the two candidates and confirm the target plant directly.

*After the user confirms the final plant species, the system redirects to **UI 2.4** to display the detailed introduction of that plant.*

### Updated Prototype UI 2.4, UI 3.1, UI 3.2, UI 3.3:

A **ChatBot Floating Orb** has been added to the side of the Identification Result Page and My Plants Page (as shown in UI 2.4, UI 3.1, UI 3.2).

- Users can drag the floating orb to any position along the side of the screen.
- When needed, users can **click the floating orb** to pull up a dialog box and ask the agent any questions regarding plant care or plant introduction (as shown in UI 3.3).
- Clicking the orb again, or clicking anywhere outside the dialog box, closes the dialog.

## 9. Open Issues

### 9.1 The "Noisy Data" Challenge in Adaptive Scheduling

- **The Problem:** Our core selling point is "Intelligent Scheduling," which uses the user's actual watering records to fine-tune the next reminder time. However, we discovered a logical loophole during design: user behavioral data is full of **Ambiguity**.
- **Scenario:** The system reminds the user to water on Monday, but the user delays clicking "Done" until Thursday.
- **System Confusion:** Did the user delay because the soil was still wet on Monday (implying our prediction was too frequent and needs reduction)? Or was it simply because the user was busy or forgot (implying the prediction was accurate and needs no adjustment)?
- **Current Status & Deficit:** The current `CarePlanControl` can only mechanically postpone the next reminder based on `lastWateringTime`, lacking the ability to distinguish the user's "True Intent." Feeding this "procrastination data" directly to the algorithm would result in "Model Poisoning," erroneously stretching the watering interval indefinitely.
- **Proposed Solution:** We need to innovate in UI interaction design by creating a **"Low-friction Intent Capture"** mechanism. For example, when a user marks a task late, a lightweight prompt pops up: *"Was the soil still wet? (Yes/No)."* This requires rigorous testing to balance "data acquisition" against "user

disturbance."

### 9.2 Edge-side Rendering Performance

- **Problem Description:** Generating Segmentation Masks in the cloud and overlaying highlights (e.g., circling diseased areas) on the mobile end in real-time imposes significant bandwidth consumption and rendering jitter.
- **Architecture Evolution Direction: Edge Computing.** In future versions, we plan to utilize Flutter's Texture rendering capabilities and the mobile GPU (based on TensorFlow Lite) to migrate tasks like image segmentation from the cloud to the edge.
- **Goal:** The cloud will only return lightweight coordinate instructions, with pixel-level rendering completed locally on the phone. This not only solves image transmission latency but also enables basic visual guidance in offline states, further aligning with the "Offline-First" design philosophy.

## 10. Project Reflection & Contributions

Reflecting on the evolution from the *Part II Analysis Model* to the *Part III Design Model*, our team experienced a mindset shift from "stacking features" to "building systems." Here are our three most profound reflections:

1. Architecture is more than diagrams

In the Part II phase, although we selected the "Hexagonal Architecture," it was primarily to meet assignment requirements. It wasn't until we actually designed the IdentificationService in Part III that we realized its power: when discussing "What if the Cloud API goes down?", we discovered we only needed to add a LocalTFLiteAdapter to solve the problem without modifying a single line of core business code. This "security brought by design" is something we had never experienced in previous coding efforts.

2. AI is the engine, Process is the steering wheel

Early in the project, we fantasized that AI could automatically solve all plant identification and diagnosis problems. However, when designing the sequence diagrams for UC4 (Diagnosis) and UC2 (Identification), we realized that existing VLM models carry a risk of "Hallucination." Therefore, we forcibly introduced a "Human-in-the-loop" step in the design—where the system provides a candidate set, and the user makes the final confirmation. We learned to use interaction flows to compensate for algorithmic uncertainty, which represents a more mature engineering mindset than merely pursuing model accuracy.

3. Details determine feasibility

When designing "Proactive Recording (UC5-2)," we initially overlooked concurrency issues. During the drawing of the design-level sequence diagram, we realized: If a user records while offline, how is the Data ID generated? This forced us to introduce the SyncManager and a local UUID generation strategy. This assignment taught us that grand architecture must ultimately return to specific interface definitions and exception handling.

## 11. AI Tools Acknowledgment

In accordance with the academic integrity requirements of the course, we honestly declare: This project (Part III) utilized Generative AI tools (Gemini 3 Pro, Claude 4.5 Opus) during the completion process. The specific usage is as follows:

- **Diagram Generation:** We used AI to convert logic sketches (hand-drawn) determined by team discussions into Mermaid.js code for generating sequence and architecture diagrams. *Note: Core interaction logic (e.g., call order between objects) was designed by team members; AI was responsible only for syntax formatting.*
- **Interface Specification:** We used AI to reference the Google OpenAPI specification to generate JSON templates for API definitions. This helped unify the naming styles of `IdentificationService` and `CarePlanService`, saving repetitive typing work.
- **Proofreading:** We used AI to grammar-check and polish the English abstract and parts of the complex architectural descriptions to ensure the accurate use of professional terminology (e.g., "Dependency Injection," "Graceful Degradation").
- **Brainstorming:** For example, when designing the "Open Issues" section, we consulted AI regarding "Common data consistency challenges in hybrid architectures." It provided several lines of thought (such as conflict resolution strategies), which we filtered and refined based on the actual context of the project (the non-sensitive nature of plant dossiers).

**Declaration:** All architectural decisions, business logic designs, and core viewpoints in this report are original to this team.

## 12. Contributions of Team Members

| Student ID | Name | Percentage | Responsibilities |
|---|---|---|---|
| 2352280 | Hengyu Jin | 25% | Architecture Design & Prototype Update<br>Responsible for refining the Hexagonal Architecture and drawing deployment diagrams; updated the Figma UI prototype to adapt to the new "Proactive Recording" feature. |
| 2352609 | Qi Lin | 25% | Interface Definition & Subsystem Design<br>Responsible for the detailed API design of IdentificationService and CarePlanService; wrote subsystem interface documentation. |
| 2353409 | Baoyi Hu | 25% | Use Case Implementation (UC3, UC4)<br>Responsible for drawing detailed design sequence diagrams for "Create Dossier" and "Problem Diagnosis"; designed the implementation logic for the Session Pattern. |
| 2352288 | Sirui Da | 25% | Key Mechanisms & NFR Design<br>Responsible for designing the technical implementation of "Hybrid AI Inference" and "Offline Synchronization (SyncManager)"; wrote the Performance and Reliability chapters. |