Expert Report on Decoupled API Architecture for Zero-Cost Plant Identification and Care Platform

I. Strategic Overview: Architecting a Zero/Low-Cost Plant AI Platform

The development of a robust plant identification, care guidance, and health diagnosis platform under the strict constraint of utilizing free Application Programming Interfaces (APIs) necessitates a foundational shift from monolithic commercial solutions to a hybrid, decoupled service architecture. No single free API currently offers the combined vision intelligence, structured care data, and interactive generative capabilities required by the project mandate. This report outlines the technical blueprint for integrating multiple specialized free services to achieve full functionality while strategically minimizing operational expenditures.

A. The Economic Imperative and Hybrid Architecture Mandate

Commercial platforms, such as Plant.id, offer integrated identification and health assessment features by accepting payment upfront. For instance, accessing Plant.id's advanced features requires purchasing a minimum volume of credits, starting at 1,000 credits for €50 (at a rate of €0.05 per credit).[1] Furthermore, a single request combining species classification and health assessment consumes two credits.[2] This cost structure immediately places the full Plant.id suite outside the scope of a purely free-to-operate model for primary functionality.

This economic constraint mandates a **Decoupled Service Architecture** that links a dedicated Vision AI solely responsible for taxonomic classification (Tier 1) to an open knowledge base (Tier 2) and a free Large Language Model (LLM) for generative and conversational tasks (Tier 3). Since the selected free identification engine (Pl@ntNet) only provides core identification with strict limits (500 identifications per day) [3], it becomes structurally necessary to leverage secondary data and LLM layers to handle the

comprehensive requirements for care generation, diagnostic Q&A, and health assessment, which are typically bundled in paid solutions.

The architecture is therefore partitioned into three essential, interconnected layers:

1. **Tier 1: Vision Inference (Pl@ntNet):** Image processing to output a high-confidence Scientific Name (Taxonomy).
2. **Tier 2: Knowledge Grounding (Wikidata/Wikipedia):** Structured retrieval using the Scientific Name to pull factual identifiers and descriptive content.
3. **Tier 3: Generative Output (Together AI LLM):** Synthesis of factual data with conversational AI logic to create care guides and interactive diagnosis advice.

The selection process for the core components is critical, balancing availability against feature completeness, as demonstrated in the comparison below.

Table 1: Comparison of Core Plant Identification APIs

API Name	Cost Model	Daily/Request Limit	Health Assessment Feature	Care Details Coverage	Primary Data Output
Plant.id (v3)	Paid (Min. €50 purchase)	Highly Scalable	Yes (Integrated with Q&A logic) [2]	Comprehensive (Watering, Soil, Light, Toxicity) [2]	Scientific Name, GBIF ID, Detailed JSON
Pl@ntNet API	Free/Paid Subscription	500 ID/Day (Free) [3]	No (Identification only)	Minimal/None	Scientific Name, Probability
Self-Hosted (PlantVillage)	Free (Infrastructure cost only)	Unlimited (Hardware dependent)	Yes (Disease Classification) [4]	N/A (Only health status)	Disease Name, Binary Status (Healthy/Diseased)

B. Architectural Blueprint: The Interconnected System

The architectural strategy is to use Pl@ntNet as the entry point for identification while supplementing its lack of health and care data using structured open sources and generative models.

The identification outcome from Tier 1 (a scientific name like *Mesua ferrea*) is passed to Tier 2 to retrieve verifiable identifiers like the GBIF ID and taxonomic hierarchy.[6] This structured data then grounds the free LLM in Tier 3, ensuring that the generated care guide is factual and specific to the identified species, rather than being a generalized, potentially inaccurate, output. The optional health diagnosis component operates in parallel using a separate, self-hosted Vision AI model (Tier 5), and its results feed back into the Tier 3 LLM to initiate the required interactive troubleshooting sequence.

II. Tier 1: Core Plant Identification (The Free Backbone)

The primary component for image-based identification is the Pl@ntNet API, which offers the most accessible free tier suitable for a preliminary website launch or proof-of-concept development.

A. Pl@ntNet API Integration Strategy

Pl@ntNet is built on a collaborative model leveraging deep learning for identification across over 50,000 species.[3] For developers, the service requires an API key for access.[8]

Free Tier Commitment and Operational Constraints

The free tier of the Pl@ntNet API imposes critical constraints that dictate the scaling strategy for the project. The service is limited to **500 identifications per day**.[3] Additionally, security quotas restrict usage to

20 simultaneous requests per client.[9] These limits define the initial capacity of the proposed website. The high restrictiveness of the 500 requests per day threshold implies that

while the API is an excellent starting point for development and showcasing the product, it will be rapidly exhausted if the website achieves initial popularity. Therefore, the implementation should be treated as a proof-of-concept pipeline. Future scaling will necessitate either migrating to the Pl@ntNet Pro customer tier (which offers a security quota of 1 million requests per day) [9], or investing resources into deploying an independent, self-hosted Vision AI model specifically for plant taxonomy.

Request Methods and Image Handling

The Pl@ntNet API supports two primary methods for image submission [8]:

1. **HTTP GET:** Used for remote images via URLs.
2. **HTTP POST:** Used for local images (files), typically leveraging a multipart/form-data payload.

For a robust web application handling user uploads, the multipart/form-data POST request is the standard practice. It is essential to configure this request correctly, including the necessary parameters such as the plant organ (leaf, flower, fruit), which aids the identification engine.[10] Improper configuration of the multipart payload is a documented source of error (e.g., error code 400 "Invalid multipart payload format"), demanding careful construction of the HTTP request body.[10] The API can handle up to five images per identification request, provided they represent a single plant.[8]

B. Python Implementation: Pl@ntNet Image POST Structure (Conceptual Code)

The following structure illustrates the necessary components for a Python implementation using the requests library to execute a POST request for image identification, fulfilling the need for core code examples [11]:

Python

```
import requests
```

```
# --- Configuration ---
API_KEY = "YOUR_PLANTNET_API_KEY"
PLANTNET_API_URL = "https://my-api.plantnet.org/v2/identify/all"

# Ensure the image path is correct on the server side
IMAGE_PATHS = [
    "path/to/local/leaf_image.jpeg",
    "path/to/local/flower_image.jpeg"
]

# Required parameters for identification
# The organ parameter is critical for accuracy
PLANT_ORGANS = [
    "leaf",
    "flower"
]

def identify_plant_with_plantnet(image_paths, organs):
    """
    Handles the multipart/form-data POST request to Pl@ntNet.
    """

    files =
    for i, path in enumerate(image_paths):
        # The file field names are not strictly important, but the structure is
        files.append(('images', (f'image_{i}.jpeg', open(path, 'rb'), 'image/jpeg')))

    # Payload for the request (includes organs and API key)
    params = {
        'api-key': API_KEY,
        'organs': organs
    }

    try:
        response = requests.post(PLANTNET_API_URL, params=params, files=files)
        response.raise_for_status() # Raise exception for bad status codes

        results = response.json()

        # Parsing the Response: Extract the top suggestion
        if results.get('results'):
            top_suggestion = results['results']
```

```
            scientific_name = top_suggestion['species']
            probability = top_suggestion['score']

            print(f"Identification successful: {scientific_name} (Confidence: {probability:.2f}%)")
            return scientific_name

        return None

    except requests.exceptions.RequestException as e:
        print(f"API Request Failed: {e}")
        return None

# Example usage (uncomment after replacing API key and paths)
# plant_name = identify_plant_with_plantnet(IMAGE_PATHS, PLANT_ORGANS)
```

III. Tier 2: Knowledge Grounding and Data Retrieval (Wikidata & Wikipedia)

The Scientific Name retrieved from Pl@ntNet (Tier 1) lacks associated care instructions, health status, or descriptive text necessary for the website. Tier 2 addresses this gap by retrieving structured, factual data from open databases, grounding the subsequent generative AI layer.

A. Leveraging Wikidata via SPARQL for Structured Botanical Data

Wikidata operates as a global, open database that stores facts as simple statements (triples), making it inherently machine-readable and highly valuable for connecting disparate botanical data.[6] It contains tens of thousands of entries relevant to botany, including taxonomic identifiers and scientific information.[6]

API Access and SPARQL Query Implementation

Access to Wikidata is achieved via the SPARQL Protocol and RDF Query Language, which

allows for complex searches against the structured knowledge graph.[13] This method is free to use.

The rationale for using Wikidata over simply scraping Wikipedia text is the ability to retrieve *structured properties* rather than narrative text. For example, once the scientific name is known, a SPARQL query can retrieve properties such as the taxonomic name authority (P225), subclass relationships, or even specialized identifiers like the GBIF ID.[6] Although explicit, detailed care requirements (like specific watering regimes) are often stored in narrative text fields and may be incomplete in Wikidata [2], the structural identifiers are robust.

Data Harmony via Identifiers

A crucial architectural principle for integrating multiple data sources is the use of a universal identifier. The **GBIF ID (Global Biodiversity Information Facility ID)** serves this function.[6] While Pl@ntNet provides the initial scientific name, Wikidata often indexes botanical entries by their GBIF ID.[6] Commercial APIs like Plant.id also include the GBIF ID (97.77% coverage) in their responses.[2] By querying Wikidata to resolve the scientific name into its corresponding GBIF ID, the platform establishes a canonical key that ensures semantic consistency across the Pl@ntNet identification, the structured Wikidata facts, and any future data enrichments, minimizing data harmonization complexity.

Conceptual Code Snippet: Wikidata SPARQL Query

The following Python structure uses the sparqlwrapper library (or similar methods) to query Wikidata for properties associated with a plant identified by name.

Python

```
# Conceptual SPARQL Query using a scientific name (e.g., Q39395060 for Oxalis insipida)
# This query searches for the taxonomic hierarchy and identifiers of a known plant item.

SPARQL_ENDPOINT = "https://query.wikidata.org/sparql"

def query_wikidata_for_plant_details(scientific_name_q_id):
```

```
    """
    Queries Wikidata for taxonomic details (simplified example).
    scientific_name_q_id: The Q-ID retrieved for the plant species.
    """
    query = f"""
    SELECT?propertyLabel?valueLabel WHERE {{
      wd:{scientific_name_q_id}?property?value.
      SERVICE wikibase:label {{ bd:serviceParam wikibase:language ",en". }}
    }}
    LIMIT 10
    """

    # In a real implementation, the code would execute this query
    # and parse the results to extract properties like P225 (Taxonomic Name) or P846 (GBIF ID).

    # The crucial first step is resolving the scientific name from Pl@ntNet into a Wikidata Q-ID.
    # This involves a search query preceding the detail query.
    return "Structured data retrieval ready for LLM grounding."
```

B. Wikipedia API for Narrative Content and Descriptions

While Wikidata provides the structured foundation, Wikipedia offers rich, accessible narrative text. The Wikipedia API can be used to search and fetch articles based on the scientific name.[15] This layer is essential for generating a credible output. A short summary or description of the plant can be extracted to preface the AI-generated care instructions, providing immediate context and increasing the perceived authority of the generated content. Wikipedia content focuses on taxonomy, history, and general characteristics.[16]

IV. Tier 3: Comprehensive Care Guide Generation and Interactive Q&A

This tier uses generative AI to synthesize the output from Tiers 1 and 2 into the required care guides and interactive troubleshooting dialogues, replacing the sophisticated, yet costly, features of commercial APIs like Plant.id.

A. Free LLM Alternative to GPT-4 (The Together AI Solution)

The user requires a free alternative to high-performance models like GPT-4. The analysis identifies several open-source models (GPT-NeoX-20B, LLaMA 2, BLOOM).[17] However, managing and hosting these models is computationally intensive and costly. A more strategically sound approach for a budget-constrained developer is to use a free API access layer provided by an inference provider.

Together AI provides serverless API access to leading open-source models, including **Llama 3.1 8B Instruct-Turbo**, offering a high-quality alternative to proprietary APIs.[18] The generous free limits, which allow for up to 60 requests per minute [20], make it operationally viable for a rapidly growing free service, offering a strong balance of accessibility and performance.

The following table summarizes the operational viability of free LLM alternatives:

Table 2: Viable Free/Low-Cost LLM Alternatives

Provider/Model	Free Limits	Primary Use Case	Integration Ease
Together AI (Llama 3.1 8B)	Up to 60 requests/minute [18]	Conversational Q&A, Structured Output Generation	High (Standard Python SDK) [19]
HuggingFace Serverless	Limited free credits, complex limits [20]	Specialized NLP tasks, Model exploration	Medium (Custom Inference API calls)
Self-Hosted LLaMA 2/BLOOM	Unlimited (After high setup cost)	High volume, low latency specific tasks	Low (Requires dedicated GPU infrastructure)

B. Prompt Engineering for Care Guide Synthesis

The greatest technical challenge in using a free LLM for this task is ensuring **factual grounding**. The LLM must be tightly constrained by the verifiable data retrieved in Tier 2 to

prevent "hallucination," where it generates generic or incorrect advice.

Prompt Structure 1: Comprehensive Care Guide Generation

The prompt must instruct the model to adopt a persona (e.g., Master Botanist) and use the retrieved inputs as mandatory factual components.

Input Data:

- Scientific Name (from Pl@ntNet).
- Taxonomy (Kingdom, Family, Genus, extracted from Wikidata).
- Wikipedia Summary (description text).

LLM System Instruction Template (Llama 3.1 format):
"You are a Master Horticultural Consultant. Your goal is to generate a concise, accurate, and professional care guide for the identified plant. The guide must strictly adhere to the provided factual context. If specific care details (watering, light) are missing from the input, state that the information is generally unknown and offer a sensible default for similar plants in its genus. Structure your response into four clearly labelled sections: 1. Overview and Taxonomy, 2. Watering Requirements, 3. Light and Soil Needs, 4. Propagation Methods."

Prompt Structure 2: Interactive Troubleshooting and Q&A Loop

The user's requirement for a follow-up diagnosis loop, similar to the logic provided by the paid Plant.id health assessment feature which includes a follow-up question field [2], must be replicated using the LLM's conversational capabilities.

If the self-hosted health API (Tier 5) returns a Diseased status and identifies a potential issue (e.g., "Maize Rust" [4]), the LLM initiates a focused dialogue.

LLM Q&A Instruction Template:
"The user’s has been diagnosed with. Initiate an interactive troubleshooting dialogue. You must ask the user three specific, differential questions to refine the diagnosis and understand the plant's environment. Questions should cover recent watering schedule, location of symptoms (e.g., leaf underside, trunk), and presence of visible pests. After receiving the user's answers, immediately synthesize the necessary treatment (prevention and biological/chemical steps). Use the detailed treatment guidelines (if provided via Tier 5 data) to ground your advice."
This strategy leverages the advanced conversational skill of the LLM to provide the complex,

interactive diagnostic feature without requiring manual branching logic in the website's backend code.

V. Tier 5: Plant Health Diagnosis and Interactive Disease Detection

The health assessment component is optional but highly desirable. Achieving this freely requires contrasting the comprehensive commercial solution with a practical open-source deployment.

A. Plant.id Health Assessment (The Structural Benchmark)

Commercial APIs like Plant.id integrate health assessment via a health parameter (only, auto, or all).[2] When a disease is detected, Plant.id returns detailed information, including a

treatment dictionary compiled by domain experts (not AI-generated), covering prevention and chemical/biological steps.[2] Most uniquely, the response can include a crucial follow-up

question designed to differentiate between similar diseases or suggestions, complete with optional text.[2]

While this integrated system provides high confidence and ready-made interactive logic, it is entirely credit-based.[1] The developer must use this structure as a template for designing the necessary data pipeline and conversational flow for the free LLM solution.

B. Open-Source Alternative: Self-Hosting a Disease Classification Model

To achieve free health detection, the visual classification task must be removed from paid APIs and managed internally.

Model Selection and Data Specialization

High-accuracy deep learning models, typically Convolutional Neural Networks (CNNs), trained on standardized datasets like the **PlantVillage-augmented dataset**, are readily available on platforms like GitHub.[4] These models, often built using frameworks like Keras (Tensorflow), classify images (usually leaves) into specific categories (e.g., healthy, various fungal diseases, pests).[4]

A critical architectural constraint arises from the specialization of these free models. Open-source solutions often exhibit high *depth* (accuracy within their training set, e.g., 95%+ accuracy on specific leaf diseases like Maize Rust) [4] but poor

breadth. They are trained on limited, controlled image sets, focusing heavily on leaf diseases.[23] They lack the generalization capabilities of massive commercial models that can diagnose broader issues like structural problems, nutrient deficiencies, or general plant stress based on environmental context. The final website must manage user expectations, acknowledging that the self-hosted API is specialized.

Deployment Strategy: Wrapping the Model in an Internal API

The chosen Keras/Tensorflow model must be wrapped in a microservice API, typically using **Flask or FastAPI**, to provide a single internal endpoint for the website backend. This service acts as the dedicated Health API (/api/v1/diagnose).

The workflow is:

1. User submits image.
2. Backend sends image to the self-hosted Health API (Tier 5).
3. The API uses the pre-trained model to run model.predict(image_data) and returns the disease classification and confidence score.
4. If a disease is identified, the disease name and confidence are sent to the Together AI LLM (Tier 3) to trigger the conversational troubleshooting template (Prompt Structure 2).

VI. Technical Implementation Details and Harmonization Code Structures

The key to a successful hybrid architecture is building a robust middleware layer that harmonizes the diverse outputs and inputs required by each specialized service.

A. Data Schema Mapping: Unifying Disparate API Outputs

The results from Pl@ntNet (scientific name), Wikidata (structured facts), and the internal Health API (disease name) must be mapped to a standardized internal schema before being passed to the LLM for synthesis. The middleware layer is responsible for creating a **fact bundle** that grounds the LLM.

Table 3 details the required information and the optimal free source for its retrieval, guiding the developer in pipeline construction.

Table 3: Required Data Attributes and Sources for Comprehensive Care

Required Information	**Optimal Primary Source (Free)**	**API Access Method**	**Purpose in LLM Prompt**
Scientific Name	Pl@ntNet API (Tier 1) [3]	HTTP POST	Core Identification and search key.
GBIF ID	Wikidata (Tier 2) [6]	SPARQL Query	Canonical reference and cross-database linking.
Taxonomy (Family, Order)	Wikidata (Tier 2) [6]	SPARQL Query	Factual anchors for grounding.
General Description	Wikipedia API (Tier 2) [15]	Search/Extraction	Narrative context generation.
Health Status (Binary)	Self-Hosted Keras Model (Tier 5) [4]	Internal API Call	Trigger for Q&A troubleshooting loop.

Treatment/Prevention Steps	LLM Synthesis (Tier 3)	Generative Prompt 2	Actionable advice generation (grounded by external expert data where possible).

The LLM output for the care guide should ideally mimic the desirable, though paid, Plant.id care details (e.g., best_watering, best_light_condition, best_soil_type, and toxicity).[2] While Plant.id provides these as pre-compiled string paragraphs (English only) [2], the free architecture relies on the LLM to generate them accurately based on the factual data points retrieved from Wikidata's related properties.

B. Robust Error Handling and Authentication

Maintaining service reliability requires rigorous attention to API key security and rate limit management.

Rate Limit Management

The Pl@ntNet constraint of 500 identifications per day [3] requires proactive management. The backend system must implement client-side monitoring and, ideally, basic caching for recently identified species to conserve API calls. Should the service approach the daily limit, a graceful fallback must be implemented, such as queuing requests or temporarily notifying the user of capacity limitations. When an identification returns a low confidence score, the system should avoid querying subsequent Tiers and instead prompt the user for better images.

API Key Management

Both the Pl@ntNet and Together AI services require API keys for authentication.[8] These keys must never be exposed client-side. The backend server (e.g., Flask, Django, or Node.js) must securely store these credentials, preferably using environment variables (

TOGETHER_API_KEY=xxxxx) [19], and manage all API transactions server-side.

VII. Appendix: Essential Python Code Snippets (Full Code Reference)

This section provides actionable code templates for the core components of the decoupled architecture.

A. Code 1: Pl@ntNet API POST Request (Image Identification Template)

The structure below is a refined example of the essential HTTP POST request using the requests library for sending local images, which is necessary for production web applications.[8]

Python

```
import requests
import json

def plantnet_identification_request(api_key, image_paths, organs):
    """
    Submits plant images to the Pl@ntNet API using multipart/form-data.

    :param api_key: Your private Pl@ntNet API key.
    :param image_paths: List of local paths to image files.
    :param organs: List of organs visible in the corresponding images (e.g., ['leaf', 'flower']).
    :return: JSON response object or None on failure.
    """
    url = "https://my-api.plantnet.org/v2/identify/all"

    files =
    for i, path in enumerate(image_paths):
```

```
        files.append(('images', (f'image_{i}.jpg', open(path, 'rb'), 'image/jpeg')))

    params = {
        'api-key': api_key,
        'organs': organs,
        'lang': 'en'
    }

    try:
        response = requests.post(url, params=params, files=files)
        response.raise_for_status()
        return response.json()
    except requests.exceptions.RequestException as e:
        print(f"Pl@ntNet Request Error: {e}")
        return None

# Example Usage (requires local files and API key)
# # results = plantnet_identification_request("YOUR_KEY", ["img/leaf.jpg"], ["leaf"])
# # print(json.dumps(results, indent=2))
```

B. Code 2: Together AI LLM Chat Completion (Care Guide Generation Prompt Template)

This template demonstrates how to use the Together AI Python SDK to query the Llama 3.1 8B Instruct-Turbo model, using the retrieved facts (represented here as placeholders) to generate the care guide.[18]

Python

```
from together import Together
import os

# Ensure TOGETHER_API_KEY is set in your environment variables
# os.environ = 'YOUR_TOGETHER_API_KEY'
client = Together()
```

```
def generate_care_guide_with_llm(scientific_name, description_summary, disease_status=None):
    """
    Generates comprehensive care instructions using a grounded LLM prompt.
    """

    # Factual inputs gathered from Tier 1, 2, and 5
    FACTUAL_CONTEXT = f"""
    Scientific Name: {scientific_name}
    General Description: {description_summary}
    Health Status: {disease_status if disease_status else 'Unknown'}
    """

    CARE_PROMPT = f"""
    You are a professional botanist specializing in home care.
    Using only the following factual context, generate a detailed 3-paragraph care guide.

    --- FACTUAL CONTEXT ---
    {FACTUAL_CONTEXT}
    --- END CONTEXT ---

    Paragraph 1: Overview, Common Names, and Taxonomy.
    Paragraph 2: Detailed Care Instructions (Watering frequency, Light requirements).
    Paragraph 3: Soil, Fertilization, and Troubleshooting based on health status.

    If the health status is 'Diseased', end the guide with a prompt for the user to provide more specific symptoms.
    """

    try:
        completion = client.chat.completions.create(
            model="meta-llama/Meta-Llama-3.1-8B-Instruct-Turbo", # High-performance open model
            messages=
        )
        return completion.choices.message.content

    except Exception as e:
        print(f"Together AI Request Failed: {e}")
        return "Error generating care guide. Please try again later."

# Example usage
# guide = generate_care_guide_with_llm("Oxalis insipida A.St.-Hil.", "A type of flowering plant collected in Brazil between 1816 and 1821.", disease_status="Healthy")
# print(guide)
```

C. Code 3: Basic Flask Wrapper for a Self-Hosted Keras Model (Conceptual Structure for Health API)

To achieve free health assessment, the self-hosted model must be exposed via an internal API endpoint. This Flask structure provides the framework for loading and running a Keras/Tensorflow model trained on data like PlantVillage.[4]

Python

```
from flask import Flask, request, jsonify
from tensorflow.keras.models import load_model # Assuming Keras model is used
import numpy as np
import cv2 # Library for image processing

app = Flask(__name__)

# --- Conceptual Model Loading (Insert actual model path) ---
# MODEL_PATH = 'models/plant_disease_cnn.h5'
# try:
#     DISEASE_MODEL = load_model(MODEL_PATH)
#     CLASS_NAMES = # Define your model's output classes
# except Exception as e:
#     print(f"Error loading model: {e}. Health API disabled.")
#     DISEASE_MODEL = None


@app.route('/api/v1/diagnose', methods=)
def diagnose_plant_health():
    """
    Internal endpoint to classify an uploaded image as healthy or diseased.
    """
    if 'image' not in request.files:
        return jsonify({"error": "No image file provided"}), 400

    image_file = request.files['image']

    # 1. Image Preprocessing (critical for self-hosted models)
```

```
    image = cv2.imdecode(np.fromstring(image_file.read(), np.uint8), cv2.IMREAD_COLOR)
    # Resize and normalize image according to the model's training input shape
    # processed_image = preprocess_function(image)

    # 2. Prediction
    # if DISEASE_MODEL:
        # prediction = DISEASE_MODEL.predict(processed_image)
        # predicted_class_index = np.argmax(prediction)
        # confidence = float(np.max(prediction))
        # disease_name = CLASS_NAMES[predicted_class_index]

        # return jsonify({
        #     "status": "success",
        #     "is_healthy": (disease_name == 'Healthy'),
        #     "diagnosis": disease_name,
        #     "confidence": confidence
        # })

    return jsonify({"status": "error", "message": "Health detection model unavailable or failed."}), 503

if __name__ == '__main__':
    # app.run(debug=False, host='0.0.0.0', port=5001) # Run on internal port
    pass
```

Conclusions and Technical Recommendations

The successful implementation of a zero-cost plant information and identification website requires a pragmatic, multi-vendor approach. The reliance on the decoupled architecture—Pl@ntNet for identification, Wikidata for grounding, and Together AI for generation—is not merely an option but a technical necessity driven by the cost structures of modern Vision AI services.

The primary architectural risk is the severe limitation of the Pl@ntNet free tier (500 identifications per day).[3] This limitation dictates that the system must initially be viewed as a

Minimum Viable Product (MVP) for functional testing only. A successful product will quickly require resource allocation toward either scaling the Pl@ntNet subscription or investing in a specialized, self-hosted deployment of a comprehensive Vision AI solution.

For the optional but desired health diagnosis feature, the project must accept the trade-off

between the depth and breadth of detection. While self-hosting a model trained on the PlantVillage dataset provides high accuracy for common leaf diseases, it will not offer the generalized diagnostic capability of commercial platforms.[23] The LLM layer (Together AI) is essential not only for generating care guides but also for providing the crucial

interactive scaffolding required to refine disease diagnosis, replacing the pre-programmed Q&A logic found in paid alternatives like Plant.id.

The recommended path forward is to strictly implement the data harmonization layer, utilizing the **GBIF ID** as the persistent key, ensuring that the structured data extracted from Tier 2 reliably anchors the creative output of the LLM in Tier 3.

Works cited

1. Pricing - Plant.id, accessed October 4, 2025, https://web.plant.id/pricing/
2. plant.id AI Plant Identification API by kindwise, accessed October 4, 2025, https://www.kindwise.com/plant-id
3. Pricing and conditions - Pl@ntNet API for developers - PlantNet, accessed October 4, 2025, https://my.plantnet.org/pricing
4. uditmahato/api_maize_plant_disease: An API that utilizes a Deep Learning model built with Keras (Tensorflow) to detect if a plant is healthy or suffering from Rust and Powder formation. - GitHub, accessed October 4, 2025, https://github.com/uditmahato/api_maize_plant_disease
5. plantdiseaseclassification · GitHub Topics, accessed October 4, 2025, https://github.com/topics/plantdiseaseclassification
6. Wikidata for Botanists: Connecting People, Plants, and Data - Botany One, accessed October 4, 2025, https://botany.one/2025/07/wikidata-for-botanists-connecting-people-plants-and-data/
7. Home - Pl@ntNet - PlantNet, accessed October 4, 2025, https://plantnet.org/en/
8. Pl@ntNet-API system and user guide - Zenodo, accessed October 4, 2025, https://zenodo.org/records/7789861/files/Pl@ntNet-API-system-and-user-guide.pdf?download=1
9. General Terms, Conditions and Access policy of Pl@ntNet API - PlantNet, accessed October 4, 2025, https://my.plantnet.org/terms_of_use
10. Swift https POST request for the PlantNet API - Stack Overflow, accessed October 4, 2025, https://stackoverflow.com/questions/65761183/swift-https-post-request-for-the-plantnet-api
11. I have created a program with an interface that identifies plants : r/Python - Reddit, accessed October 4, 2025, https://www.reddit.com/r/Python/comments/xmqgx0/i_have_created_a_program_with_an_interface_that/
12. How to use my.plantnet API - GitHub, accessed October 4, 2025, https://github.com/plantnet/my.plantnet

13. Wikidata:SPARQL tutorial, accessed October 4, 2025, https://www.wikidata.org/wiki/Wikidata:SPARQL_tutorial
14. A collection of examples of SPARQL queries to Wikidata - David S. Batista, accessed October 4, 2025, https://davidsbatista.net/blog/2023/01/19/SPARQL_WikiData/
15. How to Use Wikipedia API with Python | Scrape Wikipedia Pages Easily! - YouTube, accessed October 4, 2025, https://www.youtube.com/watch?v=pcFk4pbjrgo
16. Plant taxonomy - Wikipedia, accessed October 4, 2025, https://en.wikipedia.org/wiki/Plant_taxonomy
17. Top Free LLM tools, APIs, and Open Source models - Eden AI, accessed October 4, 2025, https://www.edenai.co/post/top-free-llm-tools-apis-and-open-source-models
18. Together.ai Docs: Introduction, accessed October 4, 2025, https://docs.together.ai/
19. Quickstart - Together.ai Docs, accessed October 4, 2025, https://docs.together.ai/docs/quickstart
20. cheahjs/free-llm-api-resources - GitHub, accessed October 4, 2025, https://github.com/cheahjs/free-llm-api-resources
21. Pricing - Hugging Face, accessed October 4, 2025, https://huggingface.co/pricing
22. A CNN-based image detector for plant leaf diseases classification - PMC, accessed October 4, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC9547307/
23. waqas-ali/tf_plant_disease_classification: A machine learning model to classify disease and healthy plants images. - GitHub, accessed October 4, 2025, https://github.com/waqas-ali/tf_plant_disease_classification