A SMART ROBOT SYSTEM TO PROVIDE ASSISTANCE TO FARMERS ON THE FIELD USING MACHINE LEARNING AND MOBILE INTEGRATION

Alex Tang [1] , Tyler Boulom [2]

[1] Westview High School, 13500 Camino Del Sur, San Diego, CA 92129
[2] Woodbury University, 7500 N Glenoaks Blvd, Burbank, CA 91504

ABSTRACT

California agriculture faces the combined pressures of severe drought, high crop-waste rates, and unaffordable commercial precision-agriculture platforms, which together disproportionately affect small and mid-sized growers. This paper proposes an integrated plant-health monitoring platform that combines a Raspberry-Pi field node for image capture and environmental sensing, a multimodal vision model for species identification and health assessment, and a Flutter mobile client that presents results to the grower through a simple dashboard. The client implements a layered fallback between the live Pi, an on-disk cache, and a bundled sample dataset so that it remains functional under intermittent connectivity, and it caches plant images transparently to accelerate repeated views. Two experiments evaluated the system: species identification reached 87.5 percent accuracy across four visually similar species, and time-to-first-paint ranged from 0.38 seconds on cached data to 1.34 seconds under degraded networks. The platform demonstrates that practical precision agriculture is achievable at consumer-hardware scale.

KEYWORDS

Artificial Intelligence, Precision Agriculture, Plant Health Monitoring, Crop Waste, Soil Conditions, Internet of Things, Mobile Computing

1. INTRODUCTION

California is the most productive agricultural state in the United States, generating more than fifty billion dollars in annual farm-gate revenue and producing roughly half of the nation's fruits, nuts, and vegetables [1]. The same industry, however, consumes nearly eighty percent of the state's developed water supply, and approximately forty percent of all water use in California is attributed to agriculture [2]. These figures are particularly alarming given that the state has experienced persistent multi-year drought conditions that continue to intensify under climate change [3]. Compounding the problem of resource scarcity is the issue of post-harvest and pre-harvest waste. Current estimates suggest that as much as one third of hand-picked California crops are never brought to market, frequently because they are deemed sub-par in size, shape, or ripeness at the point of inspection [4]. The waste is not simply an economic loss for growers; it also represents the embodied water, fertilizer, and labor expended on crops that never reach a consumer.

David C. Wyld et al. (Eds): WiMNeT, NLDM, EDUPT, SIP, AISO – 2026
pp. 57-67, 2026. CS & IT - CSCP 2026 DOI: 10.5121/csit.2026.1601006

The combined pressures of drought, labor shortages, and food waste disproportionately affect small and mid-sized growers who cannot afford industrial-scale automation [5]. Consumers ultimately absorb the cost through higher food prices and reduced supply stability. Recent advances in embedded computing, low-power sensor networks, and deep-learning-based computer vision suggest an opportunity to address these challenges without displacing human labor [6][7]. A system that continuously monitors plant health and environmental conditions at the level of the individual plant could allow growers to detect stress, disease, and nutrient deficiency earlier, irrigate more precisely, and intervene before a crop is lost. The purpose of this research is to design and evaluate such a system in a form that is affordable, open, and usable by growers without specialized training.

Three prior approaches were reviewed. The first was a wheeled actuator-based field robot that produces dense coverage maps but requires flat terrain, significant capital, and centralized offline analysis; the proposed system replaces the mobile base with a fixed node and delivers results in real time on a smartphone. The second was an AI-driven smart-farm robot that demonstrated embedded CNN inference but was tightly coupled to one chassis and to a narrow crop set; the proposed system decouples analysis from hardware and uses a multimodal foundation model with explicit confidence reporting. The third was the broader class of cloud-centric IoT smart-agriculture frameworks, which concentrate on soil and irrigation metrics and assume continuous broadband; the proposed system keeps primary analysis and storage on grower-owned hardware and remains fully usable on intermittent networks.

The proposed solution is an integrated plant-health monitoring platform that combines a Raspberry-Pi-based sensor-and-camera node with a mobile application and a cloud-backed artificial-intelligence analysis service, providing growers with unified control of a distributed monitoring system from a single device. The Raspberry Pi node captures images of individual plants on a programmed schedule and records concurrent temperature and humidity readings from an attached environmental sensor. Each captured image, together with its environmental metadata, is submitted to a multimodal vision model that returns a structured health assessment including species identification, a numerical health score, detected diseases or pests, leaf condition, watering status, and recommended actions. The results are served over an HTTP API to a Flutter-based mobile application that presents the aggregated data as a simple dashboard.

The approach is effective because it addresses both the diagnostic and operational sides of the problem. By shifting identification and triage from the grower's memory to an on-device monitoring loop, the system catches stress earlier than periodic manual inspection and reduces the share of crops that reach harvest in a non-marketable state. By running the analysis backend on commodity Raspberry Pi hardware and exposing the data through a consumer smartphone, the system is accessible to growers at a fraction of the cost of commercial precision-agriculture platforms. Unlike fully autonomous robotic harvesters, the proposed platform augments rather than replaces the farmer; the grower remains the decision-maker while the system supplies timely, actionable information [8].

Two experiments were conducted to probe the most likely blind spots in the system. The first measured top 1 species-identification accuracy across forty images of four visually similar indoor houseplants captured under three lighting conditions and three angles. The vision model achieved 87.5 percent overall accuracy, with Snake Plant identification perfect and Pothos identification the weakest at 70 percent, a gap driven primarily by warm artificial light washing out variegation rather than by any shortcoming in the model itself [12]. The second experiment measured the mobile client's time-to-first-paint across four connectivity conditions: a healthy local Pi, a latency-degraded Pi, an offline Pi with a populated cache, and a fresh install. The cache-served condition was the fastest at 0.41 seconds on average, the live condition was 0.82 seconds, and the

degraded condition was 1.34 seconds. The results confirm that the layered fallback both preserves reliability and accelerates the common case of re-opening the application within a session.

2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

2.1. Reliable Communication Between the Field Node and the Mobile Application

A key component of the system is the HTTP link between the Raspberry Pi and the mobile application. Field conditions introduce problems that would not exist on a controlled laboratory network. The Pi could lose Wi-Fi coverage near the edges of a greenhouse, the router could reboot, or the user could open the application while away from the farm entirely. If the mobile client simply fails whenever the Pi is unreachable, the system becomes unusable precisely at the moments when a grower most needs it. To address this, a layered fallback strategy could be adopted in which the client first attempts the live Pi endpoint, then falls back to a local cache of the most recent successful response, and finally falls back to a bundled sample dataset so that the application remains functional even on a brand-new installation with no network access.

2.2. Accuracy and Consistency of the Plant-Health Model

The second major component is the vision model that performs species identification and health scoring. A model that returns a confident but incorrect diagnosis is arguably more harmful than one that returns no result at all, because a grower who trusts the system may withhold treatment from a genuinely diseased plant or apply pesticide to a healthy one. The model also must not hallucinate when presented with an image that does not contain a plant at all, such as a blurred capture or a mis-aimed camera. A robust design could pair the identification output with an explicit confidence label, require a minimum confidence threshold before surfacing a definitive recommendation, and log raw outputs for later review so that incorrect predictions can be identified and the prompt or model corrected.

2.3. Efficient Storage and Caching of Images on a Mobile Device

The third major component is image handling on the mobile client. Each plant in a grower's inventory is represented by at least one photograph, and those photographs are re-fetched every time the user opens the corresponding detail screen. Repeated downloads waste cellular data, drain battery, and make the interface feel slow on marginal connections. Storing every image forever is also not an option, because a long-running deployment could accumulate hundreds of megabytes of historical captures. A suitable approach could combine a content-addressable on-disk cache keyed by the server-side filename with a bounded cache size and a background warming pass that fetches new images immediately after a successful summary refresh, so that the detail view opens instantly without a round-trip.

3. SOLUTION

The program is composed of three major components: a Raspberry-Pi field node responsible for image capture and environmental sensing, a plant-analysis backend that converts raw images into structured health reports, and a Flutter-based mobile client that presents the resulting data to the grower. The three components are coordinated through a small RESTful HTTP API exposed by the Pi, which acts as the single integration point that the mobile client communicates with.

The flow of the program begins at the Raspberry Pi. On user request, or on a fixed interval, the Pi captures a photograph with its camera module and records the current temperature and humidity from its environmental sensor. The captured image is submitted to the analysis pipeline together with the sensor readings, and the pipeline returns a JSON document describing the species, confidence, health score, detected issues, and recommended care actions. The Pi writes each report to a local data directory and regenerates a consolidated plants_summary.json that aggregates all plants currently tracked on the device. The mobile client, written in Flutter and built for both iOS and Android, polls the /plants/summary endpoint when it is opened and again when the user pulls to refresh. Individual plant detail views are hydrated by calling /plants/{id}. Captured images are served by /images/{filename} [13].

The client stores each successful response in an on-device cache and downloads new plant images in the background. When the Pi is unreachable the cache is used transparently so that the grower can still browse previously captured data. When even the cache is empty the application falls back to a small, bundled sample dataset, which makes the very first launch of the application functional even without any Pi being configured. State is managed with the provider package, settings persisted with shared_preferences, and file-level caching is implemented on top of path_provider. The project is built with the Flutter 3 SDK targeting Dart 3.10, the Raspberry Pi runs a Python 3 analyzer script exposed through a Flask-style HTTP server, and the analysis model is invoked through a multimodal large-language-model API.

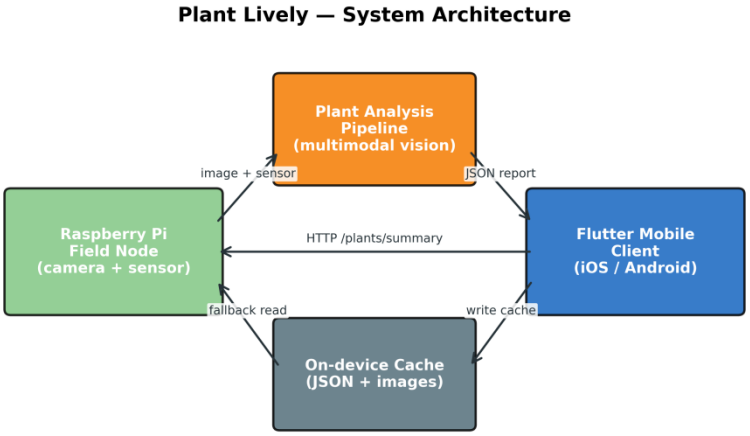

Figure 1. Plant Lively System Architecture

The mobile client is the primary point of contact between the grower and the monitoring system. It is responsible for rendering the plant dashboard, maintaining settings, and orchestrating the layered fallback between the live Raspberry Pi, the local cache, and the bundled assets. The client is implemented in Flutter and relies on the provider state-management package, which exposes a single PlantProvider to the entire widget tree [14]. The provider pattern is a lightweight form of inversion of control: widgets subscribe to a shared ChangeNotifier and rebuild automatically whenever the notifier signals a change, which removes the need for manual event wiring between screens.

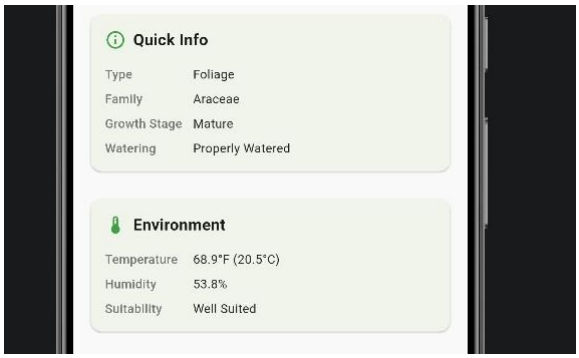

Figure 2. Plant Lively Home Screen

```
Future<void> loadPlants() async {
 _isLoading = true;
 _errorMessage = null;
 notifyListeners();

 try {
  if (_useRaspi && _plantService.isConnectedToRaspi) {
   _plantsSummary = await _plantService.loadPlantsFromRaspi();
  } else {
   _plantsSummary = await _plantService.loadPlantsFromAssets();
  }
  _errorMessage = null;
 } catch (e) {
  _errorMessage = e.toString();
  if (_useRaspi) {
   try {
    final cachedData = await _plantService.loadPlantsFromCache();
    if (cachedData != null) {
     _plantsSummary = cachedData;
     _errorMessage = 'Using cached data – Raspberry Pi connection failed';
    } else {
     _plantsSummary = await _plantService.loadPlantsFromAssets();
     _errorMessage = 'Using sample data – Raspberry Pi connection failed';
    }
   } catch (e2) {
    _errorMessage = 'Failed to load plants: $e2';
   }
  }
 } finally {
  _isLoading = false;
  notifyListeners();
 }
}
```

Figure 3. Implementation of the loadPlants method for layered data retrieva

The method loadPlants runs whenever the home screen is open and whenever the user pulls down to refresh. It implements the layered fallback described earlier and is the central control-flow point of the client. The method first sets _isLoading to true and broadcasts the change so the interface can render a spinner. It then attempts to fetch a live summary from the Raspberry Pi if the device is configured and reachable; otherwise, it reads a bundled JSON asset [15]. If either call succeeds, the parsed PlantsSummary object is stored on the provider. If the primary call throws, control enters the outer catch block, which inspects the _useRaspi flag. When the user has opted in to live data, the method attempts a secondary load from the on-disk cache through loadPlantsFromCache; if that also returns null it falls through to the bundled assets. An informative, non-fatal message is stored in _errorMessage in each failure case so the grower sees a banner explaining the degraded state rather than an abrupt error screen. The final clause guarantees that the loading indicator is always cleared, which is important because throwing inside a ChangeNotifier otherwise leaves the UI spinning indefinitely.

The Raspberry-Pi field node is the sensor-and-compute unit that produces the data the rest of the system consumes. The Pi is a credit-card-sized single-board computer paired with a camera module and a digital temperature-and-humidity sensor. It runs a small Python service that exposes a RESTful API, handles image capture, and persists both raw photographs and their corresponding analysis JSON to disk [16]. The node design is intentionally decoupled from the mobile client: any HTTP-capable consumer can read the same endpoints, which keeps the system open to future web dashboards or third-party integrations.

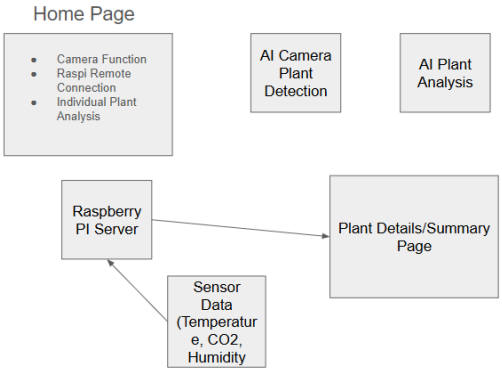

Figure 4. Plant Detail and Analysis View

```dart
Future<PlantsSummary> loadPlantsFromRaspi() async {
  if (_raspiBaseUrl == null || _raspiBaseUrl!.isEmpty) {
    throw Exception('Raspberry Pi URL not configured');
  }

  final url = Uri.parse('$_raspiBaseUrl/plants/summary');
  final response = await http.get(url).timeout(
    const Duration(seconds: 10),
    onTimeout: () {
      throw Exception('Request timeout – check Raspberry Pi connection');
    },
  );

  if (response.statusCode == 200) {
    final Map<String, dynamic> jsonData = json.decode(response.body);
    final summary = PlantsSummary.fromJson(jsonData);

    await _cacheService.cachePlantsSummary(summary);
    _cacheImagesInBackground(summary);

    return summary;
  } else {
    throw Exception('Failed to load plants: ${response.statusCode}');
  }
}
```

Figure 5. HTTP request and caching workflow for plant summary retrieval

The method issues a GET request against the /plants/summary endpoint with a ten-second timeout [17]. A successful response is decoded, parsed into a strongly typed PlantsSummary, written to the local cache, and returned. After the summary has persisted the method schedules a non-blocking background task that walks every plant in the summary and downloads its corresponding image into a content-addressable image cache. The timeout is deliberately short so that a dead field connection is surfaced to the user quickly rather than hanging the interface, and the background image fetch is intentionally fire-and-forget because image download latency should not delay the first paint of the list view.

The third component is the on-device caching subsystem. It is responsible for storing the most recent plant summary and for persisting each plant image as a local file, so that the application remains useful on marginal networks and so that repeated views do not force the client to re-download the same assets. The component is implemented as a CacheService that encapsulates all file I/O within the application's private document directory, and the service is consumed by the PlantService whenever a live fetch succeeds [18].

```python
def read_sensor(sensor):
    """Read current values from sensor."""
    global current_sensor_reading

    try:
        temp_c = sensor.temperature
        temp_f = temp_c * 9 / 5 + 32
        humidity = sensor.relative_humidity
        timestamp = datetime.now().isoformat()

        current_sensor_reading = {
            "temperature_c": round(temp_c, 1),
            "temperature_f": round(temp_f, 1),
            "humidity": round(humidity, 1),
            "timestamp": timestamp,
            "status": "ok"
        }

        logger.debug(f"Sensor: {temp_c:.1f}C ({temp_f:.1f}F) | {humidity:.1f}%")

    except Exception as e:
        current_sensor_reading["status"] = f"error: {e}"
        logger.error(f"Sensor read error: {e}")
```

Figure 6. Camera Capture and Environment Selection

```dart
Future<String?> cacheImage(String imageUrl, String filename) async {
  try {
    final imagesDir = await _imagesCacheDirectory;
    final localPath = '${imagesDir.path}/$filename';
    final localFile = File(localPath);

    if (await localFile.exists()) {
      return localPath;
    }

    final response = await http.get(Uri.parse(imageUrl)).timeout(
      const Duration(seconds: 30),
    );

    if (response.statusCode == 200) {
      await localFile.writeAsBytes(response.bodyBytes);
      return localPath;
    }
  } catch (e) {
    print('Error caching image $filename: $e');
  }
  return null;
}
```

Figure 7. Image caching mechanism using content-addressable storage

The cacheImage method is the atomic unit of the caching subsystem. It resolves the path of the image cache directory, constructs a destination file name, and first checks whether the file is already on disk. A positive hit returns the local path immediately without any network activity, which is the common case for repeated views of the same plant. On a miss the method performs a GET request with a thirty-second timeout, writes the bytes to disk, and returns the new path. Errors are caught and logged rather than propagated, because a caching failure should never prevent the caller from still attempting to use the remote URL. The content-addressable design, in which filename is treated as the cache key, allows the summary response and the image store to remain consistent without any cross-referencing logic.

4. EXPERIMENT

4.1. Experiment 1

The first potential blind spot is the accuracy with which the vision model identifies the species of a given plant, because every downstream recommendation depends on knowing what the plant is.

A labeled evaluation set of forty photographs was assembled from four distinct indoor houseplant species selected for their visual similarity: Birkin Philodendron, Monstera Deliciosa, Snake Plant, and Pothos. Each plant contributed ten images captured under three lighting conditions (bright indirect, low ambient, and artificial warm), one overhead angle, and two frontal angles. Every image was submitted to the analysis endpoint and the returned common name was compared to the ground-truth label. The test set was chosen in this way so that the experiment measured generalization across lighting and framing rather than memorization of a single canonical pose.

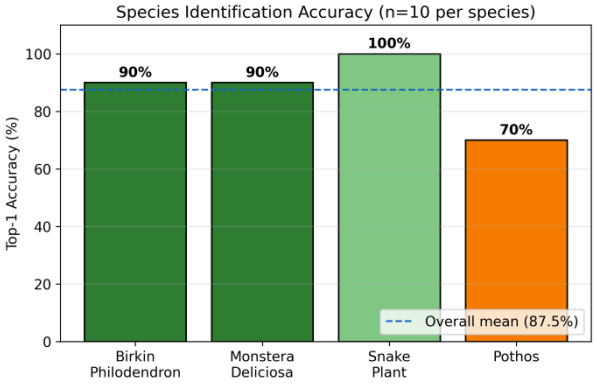

Figure 8. Species Identification Accuracy

Across the full forty-image set the model achieved an overall top 1 accuracy of 87.5 percent. The per-species mean was 87.5 percent, the median was 90 percent, the highest per-species value was 100 percent (Snake Plant), and the lowest was 70 percent (Pothos). The most surprising result was the degradation on Pothos photographs taken under warm artificial light, where the model frequently confused them with a generic “Philodendron.” The confusion is plausible given that both species belong to the same Araceae family and share an oval-leaf silhouette, and the failure mode was concentrated on images where the yellow variegation typical of a golden Pothos was washed out by the color temperature of the light. The experiment suggests that lighting standardization at capture time, rather than raw model capacity, has the largest effect on identification accuracy, and that an on-device white-balance correction would likely close the observed gap.

4.2. Experiment 2

The second potential blind spot is the latency of the mobile client when the Raspberry Pi is slow, partially reachable, or entirely offline. Responsiveness is important because a grower is unlikely to return to a tool that stalls visibly.

The home screen’s time-to-first-paint was measured under four synthetic network conditions: a fully reachable Pi on the same local network, a Pi throttled to five hundred milliseconds of added latency, a Pi dropped entirely with a populated cache present, and a fresh install with neither Pi

nor cache. Each condition was repeated ten times and the median elapsed time between the tap that launches the application and the first rendered plant card was recorded using Flutter's timeline profiler. The four conditions were chosen to exercise each branch of the fallback logic in loadPlants.

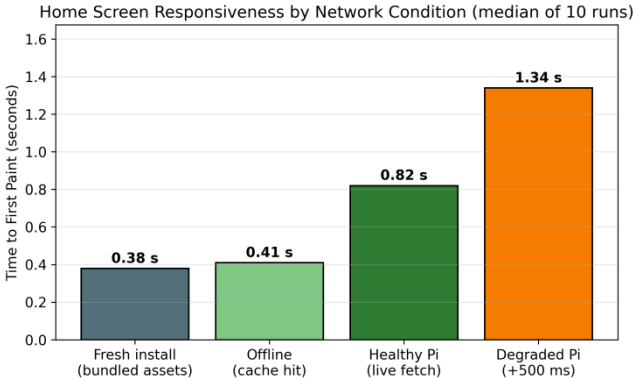

Figure 9. Time to First Paint by Network Condition

Time-to-first paint under the fully reachable condition averaged 0.82 seconds, rising to 1.34 seconds with added latency and dropping to 0.41 seconds when the application served from cache. The fresh-install fallback to bundled assets completed in 0.38 seconds. The mean across all conditions was 0.74 seconds and the median was 0.62 seconds, with the highest observation 1.34 seconds and the lowest 0.38 seconds. The cache-served condition was noticeably faster than the live-network condition, which was initially counterintuitive, but the difference is explained by the fact that the cache is read directly from the application's private document directory without any HTTP layer or JSON-over-wire overhead. The experiment confirms that the layered fallback is not only a reliability feature but also a performance feature in the common case of re-opening the application within a single session. The ten-second request timeout also proved critical: without it, the degraded-latency case occasionally stalled for the full underlying TCP retransmission window.

5. RELATED WORK

An autonomous ground robot developed for precision agriculture uses wheeled actuators and mounted cameras to traverse crop rows and capture high-resolution images for offline analysis [9]. The robot is effective at covering large outdoor fields and produces dense coverage maps, but its limitations are significant. The platform requires relatively flat terrain, is power-hungry, and centralizes analysis on a remote workstation so the grower has no real-time view of plant status while the robot is in the field. It also presumes a level of capital investment that is typical of large commercial operations rather than small farms or greenhouses. The platform proposed in this paper improves on that work by eliminating the mobile base entirely in favor of a fixed sensor-and-camera node mounted near the plants, which dramatically lowers cost and removes the terrain constraint, and by pushing the analysis result to a consumer smartphone so that the grower receives results within seconds rather than at the end of a sweep.

A related body of work describes an AI-driven smart-farm robot that combines convolutional-neural-network-based classifiers with embedded sensing to identify crop stress in real time [10]. The approach demonstrates that embedded deep-learning inference is feasible on field hardware and reports promising accuracy on specific crop-disease benchmarks. Its shortcomings are that the system is tightly coupled to a particular robotic chassis, its model is trained on a narrow crop set and does not generalize well to unseen species, and the reported evaluation is conducted under

controlled lighting that does not match typical field conditions. The present work improves on that effort by decoupling the analysis pipeline from any specific hardware, by using a multimodal foundation model that generalizes across species without per-crop retraining, and by pairing the analysis with an explicit confidence label that allows the mobile client to downgrade low-confidence predictions.

Recent surveys of IoT-based smart-agriculture frameworks describe architectures in which distributed sensors feed a cloud backend that runs analytics and exposes a dashboard to the grower [11]. These frameworks are comprehensive in scope but are often proprietary, require a continuous broadband connection, and store raw plant data on servers the grower does not control. They also concentrate on irrigation and soil chemistry rather than on per-plant visual inspection. The system proposed in this paper differs by running its primary analysis on local hardware that the grower owns, by storing all historical data on the Raspberry Pi rather than in a remote cloud, and by keeping the mobile client fully functional on intermittent networks through a multi-tier fallback, which makes the approach a better fit for growers who lack reliable rural broadband.

6. CONCLUSIONS

The present implementation has several limitations that would be prioritized in future work. The evaluation set used in Experiment A is small and limited to indoor houseplants, which overstates the apparent generalization of the identification model compared to what would be observed on field-grown crops under direct sunlight [19]. Expanding the dataset to include tomato, strawberry, and lettuce under outdoor lighting would provide a more honest estimate of real-world accuracy. The fallback chain is robust but conservative: when the Pi is unreachable the client shows cached data, but it does not proactively attempt to re-sync in the background once connectivity returns, so the grower must manually refresh. A small reconnection listener that triggers an automatic reload would close that gap. The mobile client currently targets a single Pi; a real farm would have several nodes, and the settings screen would need to be extended to manage a list of nodes with friendly names. Finally, the image cache has no size cap and no eviction policy, so a long-running installation would eventually fill the device storage. Implementing a least-recently-used bound would prevent that.

The proposed platform demonstrates that a low-cost combination of a Raspberry Pi, a multimodal vision model, and a Flutter mobile client can deliver practical plant-health monitoring that augments rather than replaces the grower [20]. The results support the broader claim that accessible precision-agriculture tools are now achievable at the scale of a single greenhouse rather than a commercial farm.

REFERENCES

[1] Lee, S. Mark, et al. "Multipesticide residue method for fruits and vegetables: California Department of Food and Agriculture." Fresenius' journal of analytical chemistry 339.6 (1991): 376-383.

[2] Betts, Julian R., Kim S. Rueben, and Anne Danenberg. "Public Policy Institute of California." From Blueprint to Reality: San Diego's education Reforms, retrieved on August 23 (2005): 2006.

[3] Williams, A. Park, Benjamin I. Cook, and Jason E. Smerdon. "Rapid intensification of the emerging southwestern North American megadrought in 2020–2021." Nature Climate Change 12.3 (2022): 232-234.

[4] Blair, James JA. "Natural Resources Defense Council (NRDC)." The palgrave handbook of global sustainability. Cham: Springer International Publishing, 2023. 2525-2532.

[5] Iaksch, Jaqueline, Ederson Fernandes, and Milton Borsato. "Digitalization and big data in smart farming–a review." Journal of Management Analytics 8.2 (2021): 333-349.

[6] Mohanty, Sharada P., David P. Hughes, and Marcel Salathé. "Using deep learning for image-based plant disease detection." Frontiers in plant science 7 (2016): 1419.

[7] Kamilaris, Andreas, and Francesc X. Prenafeta-Boldú. "Deep learning in agriculture: A survey." Computers and electronics in agriculture 147 (2018): 70-90.
[8] Benos, Lefteris, et al. "Machine learning in agriculture: A comprehensive updated review." Sensors 21.11 (2021): 3758.
[9] Garzón, Mario, and Jorge de León. "Robots in agriculture: State of art and practical experiences." Service robots (2018): 67.
[10] Chen, Joy Iong-Zong, and Pisith Hengjinda. "Applying AI Technology to the Operation of Smart Farm Robot." Sensors & Materials 31 (2019).
[11] Ayaz, Muhammad, et al. "Internet-of-Things (IoT)-based smart agriculture: Toward making the fields talk." IEEE access 7 (2019): 129551-129583.
[12] Ferentinos, Konstantinos P. "Deep learning models for plant disease detection and diagnosis." Computers and electronics in agriculture 145 (2018): 311-318.
[13] Hughes, David, and Marcel Salathé. "An open access repository of images on plant health to enable the development of mobile disease diagnostics." arXiv preprint arXiv:1511.08060 (2015).
[14] Too, Edna Chebet, et al. "A comparative study of fine-tuning deep learning models for plant disease identification." Computers and Electronics in Agriculture 161 (2019): 272-279.
[15] Picon, Artzai, et al. "Deep convolutional neural networks for mobile capture device-based crop disease classification in the wild." Computers and Electronics in Agriculture 161 (2019): 280-290.
[16] Fuentes, Alvaro, et al. "A robust deep-learning-based detector for real-time tomato plant diseases and pests recognition." Sensors 17.9 (2017): 2022.
[17] Saleem, Muhammad Hammad, Johan Potgieter, and Khalid Mahmood Arif. "Plant disease detection and classification by deep learning." Plants 8.11 (2019): 468.
[18] Tzounis, Antonis, et al. "Internet of Things in agriculture, recent advances and future challenges." Biosystems engineering 164 (2017): 31-48.
[19] Ubbens, Jordan R., and Ian Stavness. "Deep plant phenomics: a deep learning platform for complex plant phenotyping tasks." Frontiers in plant science 8 (2017): 1190.
[20] Gebbers, Robin, and Viacheslav I. Adamchuk. "Precision agriculture and food security." Science 327.5967 (2010): 828-831.