

WEED CLASSIFICATION ON MOBILE DEVICES VIA OFFLINE INFERENCE: A SOFTWARE ENGINEERING APPROACH

Matheus Bastos Chagas¹, Carlos Eduardo A. Dias¹, Alaine M. Guimarães¹, Rosane Falate¹

¹Graduate Program in Applied Computing (PPGCA) – State University of Ponta Grossa (UEPG) – Ponta Grossa – PR – Brazil

250401400010@uepg.br, 250401401002@uepg.br, alainemg@uepg.br, rfalate@uepg.br

Abstract: Computer vision techniques are a viable approach for the on-site evaluation of environmental conditions unfavorable to crop development, such as monitoring and identifying weeds. In this context, this work presents PlantScan, a system for Android platform designed for visual weed classification, adapted to the paradigm of edge computing to mitigate the effects of limited internet connectivity. The system integrates a deep learning model based on the ResNet-50 architecture — with weights originally trained and made available in a previous study in the DeepWeeds dataset — converted to the TensorFlow Lite format, making it suitable for on-device inference on Android devices. The tool also incorporates georeferencing features for mapping the distribution of detected species. Consistent with the baseline study, the classifier recognizes eight weed classes from the DeepWeeds repository. Furthermore, when evaluated on the official test subset of the same repository, the model achieved an accuracy of 96.07% on the official test subset. These results demonstrate the feasibility of deploying complex machine learning models on mobile devices and integrating offline agronomic diagnostics into field workflows.

Keywords: DeepWeeds, Mobile Computing, Edge Computing, ResNet-50, TensorFlow Lite

1. Introduction

1.1 Weeds and Their Impact on Agricultural Management

Weeds are a major cause of yield loss in global agriculture. These plants compete with crops for resources such as light, water, and nutrients, impairing plant development. Olsen *et al.* (2019) report that the annual cost of weed control and associated direct losses in pastoral systems amount to AUD 1.5 billion and AUD 2.5 billion, respectively.

Regarding yield loss in grain crops, analyses compiled by Negewo *et al.* (2024) indicate yield reductions of 35% to 87.5% due to weed competition, whereas field trials conducted by Rehman *et al.* (2020) recorded specific productivity decreases of 36% to 42%. In vegetable crops, Reginaldo *et al.* (2021) documented yield drops of up to 98% under continuous interference throughout the production cycle.

Recent studies demonstrate that weed presence affects crop development in early stages, inducing changes in physiological processes and gene expression patterns even before physical resource depletion occurs (HORVATH *et al.*, 2023). Given this urgency, rapid and accurate diagnosis is necessary to guide management interventions before the critical period of weed control ends.

Nevertheless, in-field weed identification and management rely on visual and manual inspections conducted by operators, a labor-intensive process prone to human errors (FERENTINOS, 2018; SOWMYA; BHAT, 2024). Therefore, automated, fast, and accurate diagnostic methods become important tools to enhance phytosanitary control and mitigate crop losses (PICON *et al.*, 2019; TOO *et al.*, 2019).

Conventional weed management is mostly based on the broadcast application of synthetic herbicides or mechanical soil tillage, practices that incur high operational costs and adverse ecological impacts (OLSEN *et al.*, 2019; NEGEWO *et al.*, 2024; SOLTANI *et al.*, 2022). A case study by Soltani *et al.* (2022) demonstrates that restructuring chemical programs to mitigate glyphosate-resistant biotypes imposed an additional annual cost of US\$ 43 million on producers. They also estimated that lack of updating in these control strategies would result in yield losses of up to US\$ 290 million annually.

From an environmental perspective, excessive and continuous dependence on chemical pesticides accelerates the evolution of resistance in weed populations, contaminates water resources, and degrades soil biodiversity, whereas intensive mechanical tillage increases erosion rates and fossil fuel consumption (MASSON *et al.*, 2024; SOLTANI *et al.*, 2022).

One strategy for more efficient herbicide application involves georeferenced mapping of weed infestations (OLSEN *et al.*, 2019; SOWMYA; BHAT, 2024). This approach is justified because the spatial distribution of invasive species in the field is not uniform, typically occurring in patches rather than being uniformly distributed across the field (OLSEN *et al.*, 2019). In this way, the accurate recording of the geographic coordinates of weed occurrences enables site-specific and targeted pesticide application (SOWMYA; BHAT, 2024). This site-specific management significantly reduces unnecessary chemical input consumption and operational costs (OLSEN *et al.*, 2019; NEGEWO *et al.*, 2024), while mitigating the ecotoxicological impact on the agricultural ecosystem (SOWMYA; BHAT, 2024).

1.2 Computer Vision Technologies and Edge Computing

In the field of computer vision applied to agriculture, image classification consists of algorithmically assigning a discrete categorical label to an input file. This enables, among other possibilities, mapping the input to a predefined weed species or to a negative class representing background vegetation (OLSEN *et al.*, 2019; FERENTINOS, 2018).

To this end, deep learning methods based on Convolutional Neural Networks (CNNs) constitute one of the main approaches for weed classification (SOWMYA; BHAT, 2024). On image datasets aimed at weed identification, CNNs have achieved accuracies exceeding 99% (MOHANTY; HUGHES; SALATHÉ, 2016; FERENTINOS, 2018; TOO *et al.*, 2019). Furthermore, models based on the ResNet-50 architecture can perform in-field detection under uncontrolled lighting and background conditions (PICON *et al.*, 2019).

However, a challenge in deploying these technologies in the agricultural environments is related to limitations arising from restricted or unstable network connectivity in rural areas (KALYANI; COLLIER, 2021). A possible solution lies in edge computing, where data processing occurs locally on the device. Yang *et al.* (2022) emphasize that one of the benefits of edge computing is the ability to perform tasks even without stable network coverage to connect to local or remote servers.

Mobile devices, for instance, provide an enabling platform for implementing edge computing solutions. They are low-cost, portable, and accessible platforms for farmers (MOHANTY; HUGHES; SALATHÉ, 2016; FERENTINOS, 2018). Nevertheless, many of these devices have processing, memory, and battery constraints compared to more computationally robust solutions (YANG *et al.*, 2022). Consequently, image classification models, such as CNNs, must be optimized or converted into efficient formats to enable local and native execution on the device (YANG *et al.*, 2022).

1.3 Scope, Contributions, and Objectives

The main objective of this study is to present the design, implementation, and applied evaluation of *PlantScan*, a mobile architecture for Android devices that employs an edge computing model for weed classification.

To compose the visual classification engine, the solution integrates the local execution of a pre-trained ResNet-50 model obtained from the *DeepWeeds* project (OLSEN *et al.*, 2019). *DeepWeeds* is

a public dataset comprising 17,509 images of eight weed species and associated native flora (negative class), developed for training and evaluating models under uncontrolled field conditions (OLSEN *et al.*, 2019).

Importantly, this work does not propose a new neural network architecture, nor does it involve retraining, transfer learning, or fine-tuning of the deep learning model. Its contribution consists of presenting a mobile application developed for Android devices that unifies, on a single device, local inference via TensorFlow Lite, image preprocessing, offline storage in a local database (Hive), and the georeferencing of occurrences and their results.

2. Comparative Analysis of Available Applications

This section presents a documentary comparison based on public information available in application stores and official platforms. It does not constitute a direct experimental or empirical evaluation of the applications. The assessment considered the features described by their developers and available in the published versions as of November 2025.

The applications were identified through searches using the terms “weed identification”, “plant identification”, “weed mapping”, “invasive species”, and “plant diseases”. Publicly accessible solutions were selected when they provided at least one of the following features: image-based plant identification, species information, agricultural diagnosis, georeferenced mapping, or weed-related content.

Table 1 presents a documentary and feature-based comparison between the analyzed applications and the proposed solution. The comparison highlights differences in computer vision capabilities, weed-specific functionality, georeferencing, mobile integration, and offline operation.

Table 1. Documentary comparison of available solutions based on predefined features

Solution / Application	Computer Vision	Weed Focus	Georeferencing	Integrated Mobile App	Offline Operation
Agrobase (FARMIS, 2025)	No	Yes	No	Yes	No
PlantNet (PlantNet, 2025)	Yes	No	No	Yes	No
Plantix (PEAT GmbH, 2024)	Yes	No	No	Yes	No
LeafSnap (LeafSnap, 2025)	Yes	No	No	Yes	No
WeedScan (Invasive Animals Ltd, 2025)	Yes	Yes	Yes	Yes	No
WeedWise (WS Apps, 2024)	No	Yes	No	Yes	No
Weed-AI (WeedAI, 2025)	Yes	Yes	No	No	No
<i>PlantScan</i> (proposed)	Yes	Yes	Yes	Yes	Yes

Source: The authors.

As shown in Table 1, the reviewed solutions exhibit fragmented functionality. Applications that employ computer vision, such as Plantix (PEAT GmbH, 2024) and LeafSnap (LeafSnap, 2025), are not designed for weed classification, focusing on crop diseases or tree species. Agrobase (FARMIS, 2025) and WeedWise (WS Apps, 2024) provide weed-related information but do not perform image-based classification. WeedScan (Invasive Animals Ltd, 2025) combines computer vision, weed-specific functionality, georeferencing, and mobile deployment, but its documented features do not include offline operation. Academic initiatives such as Weed-AI (WeedAI, 2025) combine computer vision with a weed-specific scope but operate as repositories or experimental web platforms without an integrated mobile application.

The documentary comparison did not identify any existing solution that explicitly combines image-based weed classification, georeferencing, mobile deployment, and offline operation. The proposed *PlantScan* architecture addresses this functional gap by integrating these capabilities with local record storage on a single mobile device.

3. Materials and Methods

This section outlines, sequentially, all the steps undertaken to develop the *PlantScan* application: (1) preparation of the development environment, (2) configuration of project dependencies, (3) sourcing and (4) conversion of the classification model, (5-6) loading the model and labels into the application, (7) image preprocessing, (8) inference execution and predicted class selection, (9) results storage, (10) auxiliary features incorporated into the application, and finally, (11-12) the procedures used to evaluate the model and functionally test the application.

3.1 Development Environment

Development began with the definition of the workspace. The *PlantScan* application was built using the Flutter 3.48.0-0.2.pre framework on the *master* channel, alongside Dart 3.12.0. The source code was written in Visual Studio Code, while Android Studio and the Android SDK were employed for subsequent stages of compilation, emulation, APK package generation, and deployment to Android devices.

To validate the classification pipeline within a mobile environment, the implementation was tested on an Android tablet equipped with a Qualcomm Snapdragon 732G processor, 8 GB of RAM, and Android 15. This same device was also used to execute the TensorFlow Lite model throughout the entire development process.

3.2 Dependencies and Project Configuration

Once the development environment was established, the project structure was set up following the standard Flutter application pattern, with the Dart environment configured for versions 3.9.0 or higher, but lower than 4.0.0. An equivalent project can be created using the commands shown in Code 1, which initialize the standard Flutter structure, define the working directory, and get the declared dependencies. From this initial structure, *PlantScan*'s dependencies were declared and managed within the `pubspec.yaml` file, reproduced in Code 2.

```
flutter create plantscan
cd plantscan
flutter pub get
```

Code 1: Flutter project creation.

To implement the machine learning pipeline, the `tflite_flutter` package (version 0.12.1) was integrated for executing the TensorFlow Lite model, alongside the `image` package (version 4.5.4) for image decoding and processing.

Also in this file, each package was mapped to a specific application responsibility: `image_picker` handled image acquisition via camera or gallery, `geolocator` retrieved geographic coordinates, and local storage was implemented using `hive` and `hive_flutter`. Subsequently, the TensorFlow Lite model and the labels file were incorporated into the project as local resources, declared in the `assets` section of the `pubspec.yaml`, as shown in Code 3.

Whenever `pubspec.yaml` is modified, the declared dependencies and resources must be updated using the command shown in Code 4, concluding the project configuration phase.

3.3 Model Origin and Architecture

With the project configured, the classification model to be deployed in the application was selected. The model from the *DeepWeeds* project (OLSEN *et al.*, 2019) was chosen, meaning no training, retraining, or fine-tuning of the neural network was performed. The *DeepWeeds* dataset compiles images of eight weed species, plus a negative class intended to represent background vegetation and non-target

images.

```
environment:
  sdk: ">=3.9.0 <4.0.0"

dependencies:
  flutter:
    sdk: flutter

  # AI / TFLite
  tflite_flutter: ^0.12.1
  image: ^4.5.4

  # Camera / Files
  image_picker: ^1.0.7
  path_provider: ^2.1.5

  # Location
  geolocator: ^14.0.2

  # Storage
  shared_preferences: ^2.2.3
  hive: ^2.2.3
  hive_flutter: ^1.1.0

  # Maps / Geo
  latlong2: ^0.9.1
  flutter_map: ^8.2.2

  # Internet / APIs
  http: ^1.1.0
  connectivity_plus: ^7.0.0

  # Charts / UI
  fl_chart: ^0.64.0
  webview_flutter: ^4.13.0
```

Code 2: Main dependencies used in PlantScan.

```
flutter:
  uses-material-design: true

  assets:
    - assets/icons/logo.png
    - assets/icons/models/weed_detector_compat.tflite
    - assets/icons/models/labels.txt
```

Code 3: Registering the model and labels as Flutter assets.

```
flutter pub get
```

Code 4: Update command.

The architecture of this model employs a ResNet-50 initialized with ImageNet weights. After the convolutional extractor, there is a GlobalAveragePooling2D layer named `avg_pool`, and finally, a fully connected layer named `fc9` containing nine output units with sigmoid activation. These nine outputs correspond to the classes listed in Table 2, in the exact order defined by the original model (OLSEN *et al.*, 2019). This order was preserved throughout the integration of the model into the mobile application.

Table 2. Classification model output classes

Output	Class
1	Chinee apple
2	Lantana
3	Parkinsonia
4	Parthenium
5	Prickly acacia
6	Rubber vine
7	Siam weed
8	Snake weed
9	Negatives

Source: The authors.

3.4 Conversion to TensorFlow Lite

The selected model was then converted to TensorFlow Lite for mobile deployment. To achieve this, the Keras-compatible ResNet-50 model provided by the *DeepWeeds* project was converted into the TensorFlow Lite format. The conversion was performed using TensorFlow 2.19.0, generating the file `weed_detector_compat.tflite`, with a size of approximately 23.93 MB (23,932,304 bytes).

To characterize the resulting artifact, the tensors of the converted model were inspected using the `get_tensor_details()` method of the TensorFlow Interpreter class. This inspection revealed that the model uses dynamic range quantization: out of its 187 tensors, 130 are in `float32` format, 54 are in `int8` format—all accompanied by quantization parameters (scale and zero point)—and 3 are in `int32` format. The input tensor, with shape `[1, 224, 224, 3]`, and the output tensor, with shape `[1, 9]`, remain in `float32`.

In other words, the internal weights of the model are stored in `int8` and dequantized at runtime by the interpreter, while the input, output, and intermediate activations remain in `float32` precision. This characterization was confirmed by direct inspection of the binary artifact distributed in the project repository, independent of the original conversion script.

This conversion procedure can be replicated from an equivalent Keras model using the TFLite-Converter, following the structure shown in Code 5, which applies the dynamic range quantization step of weights and saves the resulting `.tflite` file. Once generated, the `.tflite` file was added to the resource directory of the Flutter application along with the labels file, following the structure presented in Code 6.

```
import tensorflow as tf

model = tf.keras.models.load_model("resnet.hdf5")

converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]

tflite_model = converter.convert()

with open("weed_detector_compat.tflite", "wb") as file:
    file.write(tflite_model)
```

Code 5: Structure for converting a Keras model to TensorFlow Lite using dynamic range quantization.

3.5 Labels Configuration

Together with the converted model, the `labels.txt` file was utilized to map the output vector indices of the neural network to the classes recognized by *PlantScan*. This file contains one label per line, respecting the order of the nine output units of the network, as shown in Code 7.

```
assets/  
|-- icons/  
    |-- models/  
        |-- weed_detector_compat.tflite  
        |-- labels.txt
```

Code 6: Directory structure for model and label files in the Flutter project.

```
Chinee apple  
Lantana  
Parkinsonia  
Parthenium  
Prickly acacia  
Rubber vine  
Siam weed  
Snake weed  
Negatives
```

Code 7: Contents of the labels.txt file.

Within the application, this file is loaded via `rootBundle`: the lines are split, extraneous spaces are trimmed, and empty entries are discarded, as detailed in Code 8. At the end of this loading, the index of each element in the resulting list corresponds to the position of its respective class in the vector generated by the model.

```
Future<List<String>> _loadLabels() async {  
  final raw = await rootBundle.loadString(  
    'assets/icons/models/labels.txt',  
  );  
  
  return raw  
    .split('\n')  
    .map((e) => e.trim())  
    .where((e) => e.isNotEmpty)  
    .toList();  
}
```

Code 8: Loading the labels.

3.6 Loading the TensorFlow Lite Model in Flutter

With the model and labels already prepared, the model execution was encapsulated within the `WeedDetector` class, located in the machine learning processing module of the application. In this class, the TensorFlow Lite interpreter is initialized using the `Interpreter.fromAsset()` method, which loads the `.tflite` file embedded in the assets of the application; subsequently, the labels are loaded using the previously described function. Code 9 presents the structure used by *PlantScan* for this step, which is executed prior to any classification.

Notably, this implementation does not configure GPU, NNAPI, or any other specific delegate, nor does it set custom thread parameters in the application code. Therefore, all inference used for image classification occurs natively on the mobile device, without transmitting the image to an external server.

```
late Interpreter _interpreter;
late List<String> _labels;

Future<void> loadModel() async {
  _interpreter = await Interpreter.fromAsset(
    'assets/icons/models/weed_detector_compat.tflite',
  );

  _labels = await _loadLabels();
}
```

Code 9: Loading the TensorFlow Lite model.

3.7 Image Acquisition and Pre-processing

After loading the model and labels, the application flow proceeds to image acquisition. *PlantScan* allows the user to capture an image using the device camera or select an existing one from the gallery, implemented via the `image_picker` package. During this step, the image is requested with a quality setting of 65 and a maximum width of 1200 pixels, as shown in Code 10.

```
final picker = ImagePicker();

final foto = await picker.pickImage(
  source: origem,
  imageQuality: 65,
  maxWidth: 1200,
);
```

Code 10: Image acquisition via camera or gallery.

Once acquired, the image is read into memory and decoded using the `image` package, and then resized to a width and height of 224 by 224 pixels, the input dimensions required by the model. Code 11 details this initial pre-processing step, which omits a specific interpolation parameter in the `copyResize` method, thereby defaulting to the behavior of the `image` package version used in the project.

```
final imageBytes = imageFile.readAsBytesSync();

img.Image? image = img.decodeImage(imageBytes);

if (image == null) {
  throw Exception("Error decoding image");
}

img.Image resized = img.copyResize(
  image,
  width: 224,
  height: 224,
);
```

Code 11: Image decoding and resizing.

Then, the resized image is converted into an input tensor. To achieve this, a `Float32List` vector is created with $N = 150,528$ elements, determined by Equation (1), where *width*, *height*, and *channelColorNumber* represent the width, height, and number of color channels (red, green, and blue) of the image, respectively. The pixels are traversed in row-major order, and the red, green, and blue channels of each are sequentially inserted into this vector.

$$N = width \times height \times channelColorNumber \quad (1)$$

During insertion, each pixel intensity x is divided by 255, according to Equation (2), where x represents the original channel value within the range $[0, 255]$. This operation scales the values to the range $[0, 1]$.

$$x_{\text{norm}} = \frac{x}{255}, \quad (2)$$

Code 12 demonstrates the construction and normalization of this input tensor, concluding the pre-processing stage.

```
Float32List input = Float32List(224 * 224 * 3);

int pixelIndex = 0;

for (int y = 0; y < 224; y++) {
    for (int x = 0; x < 224; x++) {

        final pixel = resized.getPixel(x, y);

        input[pixelIndex++] = pixel.r / 255.0;
        input[pixelIndex++] = pixel.g / 255.0;
        input[pixelIndex++] = pixel.b / 255.0;
    }
}
```

Code 12: Construction and normalization of the input tensor.

It is worth noting that, during this stage, ImageNet mean subtraction, standard deviation normalization, or transforming pixels to the $[-1, 1]$ range are not applied. The resulting tensor uses `float32` values and corresponds to the logical input shape $[1, 224, 224, 3]$ of the model.

3.8 Inference Execution and Class Selection

With the input tensor prepared, it is fed to the TensorFlow Lite interpreter. Since the model contains nine classes, a nine-element output structure is pre-allocated, and inference is then executed via the interpreter's `run()` method, as detailed in Code 13.

```
var output = List.filled(
    _labels.length,
    0.0,
).reshape([1, _labels.length]);

_interpreter.run(input, output);
```

Code 13: TensorFlow Lite model execution.

The model returns a vector shaped $[1, 9]$, containing one output for each class. From this vector, the implementation identifies the highest value, such that the index i of the class with the highest output is defined by Equation (3), where s_i represents the output of class i .

$$i^* = \arg \max_i (s_i), \quad (3)$$

Code 14 presents the procedure used to locate this maximum output, iterating through the results vector starting from the second element.

```

int maxIndex = 0;
double maxValue = output[0][0];

for (int i = 1; i < _labels.length; i++) {
    if (output[0][i] > maxValue) {
        maxValue = output[0][i];
        maxIndex = i;
    }
}

```

Code 14: Selecting the model's maximum output.

Once the maximum output is identified, *PlantScan* applies a threshold τ of $1/9 \approx 0.111$ before accepting the prediction. This τ value matches the one adopted by Olsen *et al.* (2019), derived, according to the authors, as a random guess to account for the high variability of the *DeepWeeds* negative class—meaning the probability of this class occurring depends less on image features compared to the eight positive classes.

Thus, the decision rule utilized by the application is defined as:

$$\hat{y} = \begin{cases} \text{Negatives}, & \text{if } \max_i(s_i) < \tau, \\ \arg \max_i(s_i), & \text{otherwise.} \end{cases} \quad (4)$$

When the maximum output exceeds the threshold τ , the identified index is used to retrieve the corresponding class from the `labels.txt` file, as shown in Code 15.

```
String original = _labels[maxIndex];
```

Code 15: Retrieving the label corresponding to the selected class.

For display purposes, the class name is then mapped to the labels used in the *PlantScan* interface, following the translation detailed in Code 16.

```

final Map<String, String> translation = {
    "Chinee apple": "Chinee apple",
    "Lantana": "Lantana",
    "Parkinsonia": "Parkinsonia",
    "Parthenium": "Parthenium",
    "Prickly acacia": "Prickly acacia",
    "Rubber vine": "Rubber vine",
    "Siam weed": "Siam weed",
    "Snake weed": "Snake weed",
    "Negatives": "Common vegetation",
};

```

Code 16: Mapping labels for UI display.

Finally, the output of the selected class is multiplied by 100 and displayed in the interface alongside the predicted label, indicating clearly that this value represents the raw model output rather than a calibrated probability.

3.9 Storage and Complete Application Flow

Once the classification is performed, *PlantScan* requests the geographic position of the device using the `geolocator` package. When location permission is granted, the latitude and longitude associated with the analysis are acquired, and the complete result is stored locally via Hive. For each classification, the app records the local image path, the identified class, the output value of the network,

the date and time, and the coordinates. This information is subsequently utilized by the *PlantScan* history, map, and dashboard interfaces.

The complete *PlantScan* classification pipeline can be summarized as follows: (1) initialize the application; (2) load the `weed_detector_compat.tflite` file; (3) load the `labels.txt` file; (4) capture an image or select one from the gallery; (5) decode the image; (6) resize it to 224×224 pixels; (7) iterate through RGB channels; (8) divide each pixel intensity by 255; (9) construct the `float32` tensor; (10) execute the TensorFlow Lite interpreter; (11) retrieve the nine-output vector; (12) locate the maximum output; (13) map the index to the `labels.txt` file; (14) display the identified class; (15) acquire latitude and longitude; and finally, (16) store the record locally. Thus, the entire primary classification process—from image acquisition to result logging—is executed natively on the mobile device; an internet connection is not required for TensorFlow Lite model inference.

3.10 Auxiliary Features: Weather Forecast and Conversational Assistant

Beyond the completely localized image classification module, two auxiliary features reliant on external services were integrated into the app: current weather conditions retrieval and a conversational assistant.

To retrieve meteorological data, the application prompts the user for permission to access the GPS location of the device, again utilizing the `geolocator` library. Once acquired, the latitude and longitude are transmitted to the *Open-Meteo* API, configured to return current weather conditions, from which data such as temperature, wind speed, and an atmospheric condition code are extracted. This code is then translated into a textual description, such as clear sky, partly cloudy, rain, or thunderstorm.

Next, to present the location in a user-friendly format, the same coordinates are submitted to the *Nominatim* reverse geocoding service, based on OpenStreetMap data, to identify the corresponding municipality and state. The information obtained is displayed on the main interface of the application, and the last valid response is stored locally using the `shared_preferences` library, making it possible to present the previously queried data in the event of temporary unavailability of network or location service.

The agricultural conversational assistant was integrated via the *Chatbase* platform and made available through a `WebView` interface within the application. It can be accessed from the history screen for queries related to the agricultural context. As the processing of messages is handled by a remote service, this feature depends on an active internet connection, operates independently of GPS data, and functions separately from the local image classification module.

Therefore, although the weed classification module operates independently of internet connectivity, as described in Section 3.9, the two auxiliary features outlined here require network access to function. Without connectivity, the weather module displays only the last valid response stored locally via `shared_preferences`, without updates until the connection is restored; the conversational assistant, being entirely reliant on remote processing via *Chatbase*, remains completely unavailable offline, possessing no contingency mechanism. The application monitors this network connectivity status using the `connectivity_plus` package.

3.11 Evaluation Metrics

The performance of the deployed model was evaluated using accuracy, precision, recall, and F1-score metrics (SOKOLOVA; LAPALME, 2009). Accuracy represents the proportion of correctly classified images relative to the total evaluated; for each class, precision represents the proportion of correct classifications among the images assigned to that class, while recall represents the proportion of actual class instances correctly classified. The F1-score is the harmonic mean of precision and recall.

Based on these per-class metrics, two types of aggregation were calculated: macro metrics were obtained by averaging the results of the nine classes, assigning equal weight to each; weighted metrics

factored the result of each class by the number of instances of that class present in the test subset. All of these metrics were computed against the 3,507 images in the official `test_subset0.csv` partition of the *DeepWeeds* dataset. Confidence intervals were not calculated, nor were bootstrap procedures applied.

3.12 Functional Testing Procedures

In addition to quantitative model evaluation, functional tests of *PlantScan* were conducted on the same Android device described in Section 3.1. The objective was to verify the primary functions of the application and the workflow bridging image acquisition, classification, and storage. The following functions were evaluated in this order: image capture via camera, gallery image selection, classification, result presentation, record storage, and history querying.

Table 3 outlines the criteria required for each function to be considered verified. Camera capture and gallery selection were deemed successful when the application loaded the image and routed it to the classification pipeline; classification was considered verified when the TensorFlow Lite model received the image and outputted one of the nine labels defined in the `labels.txt` file; presentation was verified when the interface displayed this label alongside the output value of the model; storage was considered successful when the classification record was saved; and history querying was verified when the saved record could be retrieved via the application interface.

Table 3. Criteria for verifying application functions

Function	Verification Criterion
Camera image capture	The application successfully loads the image
Gallery image selection	The application successfully loads the image
Classification	The model receives the image and returns one of the nine labels defined in the file
Result presentation	The interface displays the label and model output value
Record storage	The classification record is saved
History querying	The saved record is retrievable via the interface

Source: The authors.

Consequently, the entire workflow was considered verified when all these functions exhibited the expected behavior without application interruption.

4. Results

This section presents the results of integrating Flutter and executing the TensorFlow Lite model on mobile devices within *PlantScan*. The evaluation considered execution of the model on an Android device, classification performance on the official *DeepWeeds* test subset, and functional verification of the application's workflow. Classification metrics characterize the implemented inference pipeline on the *DeepWeeds* dataset.

4.1 Model Performance in the Mobile Environment

The *PlantScan* application uses the `weed_detector_compat.tflite` model, with a file size of 23.93 MB (Section 3.4). Model inspection revealed input and output tensors of type `float32`; however, 54 of the 187 internal tensors are quantized to `int8` with associated quantization parameters, resulting from the dynamic range quantization applied during conversion (Section 3.4). Therefore, the distributed file corresponds to a version with quantized weights rather than a pure `float32` version.

For functional integration verification, an exploratory average inference latency of 43 ms was characterized on the evaluated physical device. This latency value was recorded during *ad hoc* opera-

tional tests on the Android tablet using the default and unrestricted configuration of the `tflite_flutter` interpreter. Because a strict benchmark protocol was not employed, this latency value is presented strictly as a contextual baseline for user experience responsiveness. It should not be used to assert mathematical or computational performance advantages over other architectures or execution environments.

4.2 Evaluation on the DeepWeeds Test Subset

As presented in Section 3.3, the TensorFlow Lite model was evaluated using the official `test_subset0.csv` partition of the *DeepWeeds* dataset. The metrics and aggregation methods used in this evaluation are defined in Section 3.11. Table 4 presents the metrics obtained by the TensorFlow Lite model on the test subset.

Table 4. TensorFlow Lite model performance metrics on the *DeepWeeds* test subset

Metric	Result
Accuracy	96.07%
Weighted precision	96.14%
Weighted recall	96.07%
Weighted F1-score	96.02%
Macro precision	97.04%
Macro recall	93.58%
Macro F1-score	95.20%

Source: The authors.

The model achieved an accuracy of 96.07%, a weighted precision of 96.14%, a weighted recall of 96.07%, and a weighted F1-score of 96.02%. Considering the macro average across classes, the model attained 97.04% precision, 93.58% recall, and a 95.20% F1-score.

To analyze the distribution of errors across classes, Table 5 presents the confusion matrix obtained for the 3,507 images in the test subset. The rows represent the true classes, while the columns represent the predicted classes of the model.

Table 5. Confusion matrix of the TensorFlow Lite model on the *DeepWeeds* test subset

True / Predicted	CA	LA	PA	PH	PAC	RV	SW	SN	NEG
Chinee apple (CA)	190	1	1	1	0	0	0	8	25
Lantana (LA)	0	190	0	0	0	0	1	2	20
Parkinsonia (PA)	0	0	197	0	4	0	0	0	6
Parthenium (PH)	0	0	0	199	1	0	0	0	5
Prickly acacia (PAC)	0	0	0	2	202	0	0	0	9
Rubber vine (RV)	0	0	0	0	0	183	2	2	15
Siam weed (SW)	0	0	0	0	0	0	212	0	3
Snake weed (SN)	2	0	0	0	0	0	1	191	10
Negatives (NEG)	5	2	0	0	1	0	7	2	1805

Source: The authors.

The confusion matrix shows a concentration of results on the main diagonal, indicating that most images were classified into their respective classes. The *Negatives* class had 1,805 correct classifications. Among the weed species, correct predictions on the diagonal ranged from 183 to 212 images. The largest

errors occurred, in part, when samples of target species were classified as *Negatives*.

4.3 PlantScan Application Results

Fifty images from the official *DeepWeeds* test partition, defined by the `test_subset0.csv` file, were used to verify the classification flow of the application. *PlantScan* processed the images from input to result storage. The tests met the criteria defined in Section 3.12. Figure 1 presents the distribution of classifications produced by *PlantScan* for these 50 images.

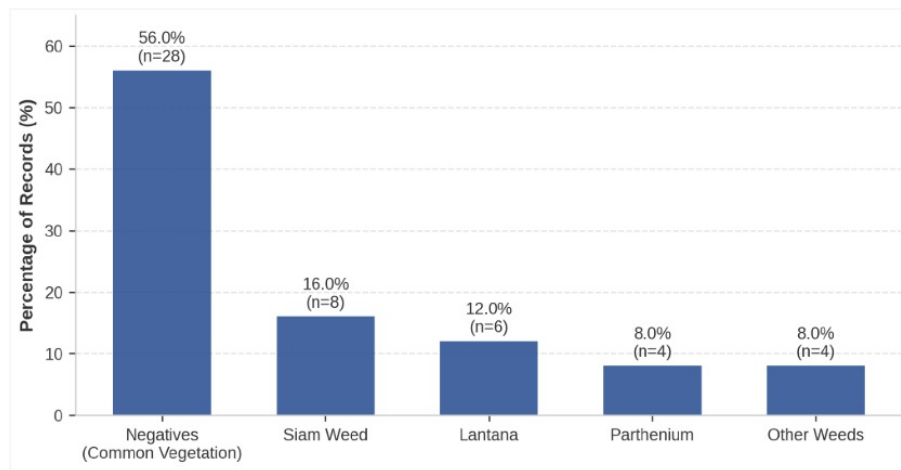


Figure 1. Distribution of classifications produced by *PlantScan* for 50 images from the official *DeepWeeds* test subset. Source: The Authors

Functional tests confirmed the operation of the main *PlantScan* workflow on the evaluated Android device. The application performed camera image capture, gallery file selection, classification with the TensorFlow Lite model, result presentation, record storage, and history querying.

5. Discussion

The technical contribution of *PlantScan* centers on integrating the *DeepWeeds* ResNet-50 model into a Flutter application via TensorFlow Lite, maintaining the underlying mobile workflow. The implementation connects image acquisition, preprocessing, on-device inference, local persistence, geolocation, dashboard access, and map-based visualization within a single application. The implemented TensorFlow Lite model utilizes `int8` dynamic range quantization for weights, keeping input, output, and activations in `float32`, as detailed in Section 3.4. It demonstrated an exploratory average inference time of 43 ms on the evaluated Android tablet.

One of the key features of the application is local model execution, which eliminates the need to transmit images to a remote server during classification and allows the core classification functionality to operate without cellular or Wi-Fi connectivity. Furthermore, detection records are also stored on the device.

The ResNet-50 architecture was adopted because it corresponds to the one distributed by the *DeepWeeds* project, and not because it is optimized for mobile constraints. Notably, even after dynamic weight quantization, the implemented model (23.93 MB) remains larger than lightweight mobile architectures. This indicates that the storage cost stems primarily from the complete ResNet-50 architecture rather than solely the weight representation format.

The confusion matrix presented in Section 4.2 corresponds to an accuracy of 96.07% and a weighted F1-score of 96.02% on the official *DeepWeeds* test subset. When interpreting this result, one must consider that the evaluation process relied on the test data of this subset, and not on real agricul-

tural field conditions. Most observed errors involved misclassifications between weed species and the *Negatives* class. Among these, 20 *Lantana*, 5 *Parthenium*, and 3 *Siam weed* images were classified as *Negatives*. Across the eight weed classes, 93 images were misclassified as *Negatives*, whereas 17 images belonging to the *Negatives* class were assigned to one of the weed classes.

This behavior was also observed in the original *DeepWeeds* study, which reported a mean top-1 accuracy of 95.7% for ResNet-50 and a 97.6% correct classification rate for the *Negatives* class (OLSEN *et al.*, 2019). The authors also identified errors between weeds and the negative class, attributing some of these misclassifications to the high visual variability of the non-target vegetation present in the dataset. Additionally, confusions between visually similar species were reported. Consequently, the errors observed in the TensorFlow Lite implementation exhibit a pattern consistent with limitations already described in the original *DeepWeeds* evaluation, although it is not possible to directly attribute every error observed in this study to a specific visual feature.

Lighting conditions, camera distance, background vegetation, plant orientation, and the presence of multiple plants in the same image can influence classification scores. Future versions of *PlantScan* could incorporate object detection, image segmentation, or explicit rejection mechanisms for unknown classes to improve operation under more complex field conditions. A systematic sensitivity analysis of the threshold and validation of the application using independently collected Brazilian field images are also recommended.

6. Conclusion

This study presented the design, implementation, and applied evaluation of *PlantScan*, consolidating an edge computing-based mobile architecture for weed classification. The solution integrated the local execution of a pre-trained ResNet-50 model, adapted for mobile devices through conversion to the *TensorFlow Lite* format.

Empirically, the findings are strictly limited to: (1) functional laboratory tests on a single Android tablet equipped with a Qualcomm Snapdragon 732G processor to verify database and georeferencing pipeline continuity, and (2) offline classification evaluation using 3,507 static images from the Australian *DeepWeeds* test subset.

While the model was successfully integrated and executed locally within the Flutter application, this study does not provide broad validation for weed species beyond those present in the *DeepWeeds* training data. Consequently, these results do not establish the operational effectiveness of the application under dynamic agricultural conditions. Further field research is necessary to evaluate the system's capabilities in the face of physical challenges, including lighting variations, wind-induced camera motion blur, complex weed-crop overlaps, and hardware performance scaling across diverse mobile chipsets in real-world environments.

The proposed architecture achieved the goal of integrating the classification workflow into a mobile application, enabling a workflow where classification, georeferencing, and record storage occur independently of network infrastructure. Auxiliary features, such as real-time weather queries and conversational assistant interactions, require network access to communicate with external services. Future work should prioritize: (i) validation using independently collected Brazilian agricultural images, (ii) expansion of supported weed classes, (iii) evaluation of unknown class rejection strategies, (iv) standardized, multi-device benchmarking on physical mobile hardware, and (v) systematic comparison with lighter quantized architectures (e.g., quantized MobileNetV3) designed for edge deployment to optimize local resources.

7. Declarations

The complete source code of the *PlantScan* application, including the Dart/Flutter codebase, image preprocessing pipeline, and TensorFlow Lite model configuration, is available for peer review and

research reproducibility at <https://github.com/ChagasEng/detector_ervas>.

To support long-term reproducibility and open science practices, the software version used in this study has been archived on Zenodo and is available at [10.5281/zenodo.22235557](https://doi.org/10.5281/zenodo.22235557).

References

FARMIS. **Agrobase – Weed, Disease**. 2025. Google Play Store. Available at: <<https://play.google.com/store/apps/details?id=lt.farmis.apps.farmiscatalog>>.

FERENTINOS, K. P. Deep learning models for plant disease detection and diagnosis. **Computers and Electronics in Agriculture**, v. 145, p. 311–318, 2018. Available at: <<https://doi.org/10.1016/j.compag.2018.01.009>>.

HORVATH, D. P. *et al.* Weed-induced changes in the maize root transcriptome reveal transcription factors and physiological processes impacted early in crop–weed interactions. **AoB Plants**, v. 15, n. 3, p. plad013, 2023. Available at: <<https://doi.org/10.1093/aobpla/plad013>>.

Invasive Animals Ltd. **WeedScan**. 2025. Google Play Store. Available at: <<https://play.google.com/store/apps/details?id=com.weedscan>>.

KALYANI, Y.; COLLIER, R. A systematic survey on the role of cloud, fog, and edge computing combination in smart agriculture. **Sensors**, v. 21, n. 17, p. 5922, 2021. Available at: <<https://doi.org/10.3390/s21175922>>.

LeafSnap. **LeafSnap: Plant Identification**. 2025. Google Play Store. Available at: <<https://play.google.com/store/apps/details?id=plant.identification.snap>>.

MASSON, S. *et al.* Reducing tillage and herbicide use intensity while limiting weed-related wheat yield loss. **European Journal of Agronomy**, Elsevier, v. 160, p. 127284, 2024. Available at: <<https://doi.org/10.1016/j.eja.2024.127284>>.

MOHANTY, S. P.; HUGHES, D. P.; SALATHÉ, M. Using deep learning for image-based plant disease detection. **Frontiers in Plant Science**, v. 7, p. 1419, 2016. Available at: <<https://doi.org/10.3389/fpls.2016.01419>>.

NEGEWO, T. *et al.* Weed science research achievements on maize in ethiopia: a review. **F1000Research**, v. 12, n. 880, 2024. Available at: <<https://doi.org/10.12688/f1000research.135210.4>>.

OLSEN, A. *et al.* DeepWeeds: A multiclass weed species image dataset for deep learning. **Scientific Reports**, v. 9, p. 2058, 2019. Available at: <<https://doi.org/10.1038/s41598-018-38343-3>>.

PEAT GmbH. **Plantix: Your Crop Doctor**. 2024. Google Play Store. Available at: <<https://play.google.com/store/apps/details?id=com.peat.GartenBank>>.

PICON, A. *et al.* Deep convolutional neural networks for mobile capture device-based crop disease classification in the wild. **Computers and Electronics in Agriculture**, v. 161, p. 280–290, 2019. Available at: <<https://doi.org/10.1016/j.compag.2018.04.002>>.

PlantNet. **Pl@ntNet: Identify Plants**. 2025. Google Play Store. Available at: <<https://play.google.com/store/apps/details?id=org.plantnet>>.

REGINALDO, L. T. R. T. *et al.* Weed interference in carrot yield in two localized irrigation systems. **Revista Caatinga**, v. 34, n. 1, p. 119–131, 2021. Available at: <<https://doi.org/10.1590/1983-21252021v34n113rc>>.

REHMAN, A. *et al.* Critical competition period of *Parthenium hysterophorus* l. in spring maize (*Zea mays* l.). **Planta Daninha**, v. 38, p. e020214143, 2020. Available at: <https://doi.org/10.1590/S0100-83582020380100085>.

SOKOLOVA, M.; LAPALME, G. A systematic analysis of performance measures for classification tasks. **Information Processing & Management**, v. 45, n. 4, p. 427–437, 2009. Available at: <https://doi.org/10.1016/j.ipm.2009.03.002>.

SOLTANI, N. *et al.* Economic impact of glyphosate-resistant weeds on major field crops grown in Ontario. **Weed Technology**, Cambridge University Press, v. 36, n. 5, p. 629–635, 2022. Available at: <https://doi.org/10.1017/wet.2022.61>.

SOWMYA, P.; BHAT, S. A comprehensive review of weed detection through advanced image processing and deep learning. **Global Journal of Research in Engineering & Computer Sciences**, v. 4, n. 1, p. 59–66, 2024. Available at: <https://doi.org/10.5281/zenodo.10685044>.

TOO, E. C. *et al.* A comparative study of fine-tuning deep learning models for plant disease identification. **Computers and Electronics in Agriculture**, v. 161, p. 272–279, 2019. Available at: <https://doi.org/10.1016/j.compag.2018.03.032>.

WeedAI. **WeedAI: Machine Learning for Weed Recognition**. 2025. Available at: <https://weedai.org>.

WS Apps. **WeedWise**. 2024. Google Play Store. Available at: <https://play.google.com/store/apps/details?id=app.web.wsappsdevs.weedwise>.

YANG, N. *et al.* Rapid image detection and recognition of rice false smut based on mobile smart devices with anti-light features from cloud database. **Biosystems Engineering**, v. 218, p. 229–244, 2022. Available at: <https://doi.org/10.1016/j.biosystemseng.2022.04.009>.