

# AgriNova — Smart Agricultural Assistant

Aakash Radhakrishnan, Krishna Yadav, Om Kamble, Sanket Bihare, Mrs. Nisha Karolia

\*\*\*

**Abstract-** *Most agricultural tech tools are either too expensive for small Indian farmers or impossible to use in a field where the cell signal keeps dropping. AgriNova is our MSBTE 6th-semester project and it does two things: identifies leaf diseases from phone photos and recommends crops from soil nutrient data. Frontend is React, backend is Flask. We fine-tuned an EfficientNetB0 on corn and potato images from PlantVillage at 128×128 resolution for disease detection. Crop recommendation uses a Scikit-Learn Random Forest on NPK and rainfall inputs. The vision model got 96.5% validation accuracy. Soil model got 98.2%. Training was the easy part, honestly. The hard part was stopping the 200MB Keras file from crashing our 4GB laptop every time two people used the site at once. We fixed it by loading the model once at startup instead of reloading it per-request, and response times dropped from seven seconds to 1.2. This paper covers how we trained the models, how we connected everything together, and all the things that went wrong.*

**Keywords—***Convolutional Networks, EfficientNetB0, Random Forest, Deployment Bottlenecks, Full Stack Engineering.*

## I. INTRODUCTION

### A. The ground reality for small farms

Most potato farmers in rural Maharashtra cannot tell early blight apart from normal leaf aging. What they do is look at the brown spots, ask someone older in the family, then spray whatever chemical is already in the shed. Sometimes it helps. Most of the time it does not. Getting an agricultural officer out to the village takes three to five days because each officer covers something like thirty villages by themselves. Maharashtra's agricultural extension system operates at roughly that ratio across most districts. By day four or five, the *Alternaria solani* spores have already spread two or three rows over and half the crop is compromised.

Honestly, crop selection might be an even worse problem. Farmers across the state plant the same crop every year without testing the soil. Nitrogen levels drop after a bad monsoon. Phosphorus shifts when they switch fertilizer brands. Nobody notices because government soil test kits cost 300 rupees and take two weeks to come back. So the farmer skips it, yields fall 15-20%, and then everyone says it was the rain.

### B. What we wanted to build

MSBTE 6th-semester micro-project needed to solve a real problem. That was the brief. We spent the first week just scrolling through GitHub and Kaggle looking at what already existed. There are probably 200 repositories out there that classify plant diseases. Kaggle has even more notebooks doing the same thing. But almost every single one stops at the notebook. You run the cell, get a number, close the browser tab. No one actually deploys it somewhere a farmer can use it.

So that became the project. Get the model out of the notebook and onto a website, and make sure the whole thing loads in under three seconds on a phone browser. Once we picked that target everything else followed from it. Backend code had to be fast. The API could not be slow. Did not matter how good the model was if the user was going to sit there staring at a loading spinner for ten seconds.

## II. PRIOR WORK AND WHAT WENT WRONG

### A. Why drones don't help here

Most published crop diagnostic research assumes you have a multispectral drone flying over a 500-acre soybean field somewhere in Iowa. If you can drop money on a DJI Matrice, sure, that works. But it means nothing for a farmer working two acres of potatoes near Pune. We were only interested in proximal sensing, which really just means holding a regular phone camera about 10 centimeters from the leaf. No drone, no extra hardware.

Before CNNs took over this space, people were doing manual feature extraction. Around 2012 to 2014 the approach was to write code that isolated leaf edges using Canny filters, or to measure green channel pixel intensities, and then pass those hand-built vectors into an SVM. It sort of worked under lab lighting. Take the same setup outdoors though and it falls apart. One shadow, one over-exposed photo from afternoon sunlight, and the SVM starts flagging healthy leaves as infected.

### B. What changed with deep learning

PlantVillage changed everything. That dataset hit Kaggle and suddenly everyone was fine-tuning AlexNet or VGG16 and publishing papers showing 99% accuracy. The catch: PlantVillage images are shot against clean backgrounds with consistent lighting. Actual farm photos

have dirt, fingers, overlapping leaves, terrible compression. Try running those 99% models on real field images and accuracy drops to maybe 80%, sometimes lower.

We started with ResNet50. Took 4.8 seconds per image on our laptop. Completely unusable for anything real-time. Then we tried MobileNet which was fast but kept confusing early blight and late blight on potato leaves. Those two need different treatments so getting them mixed up is genuinely dangerous. Ended up going with EfficientNet because it was fast enough and accurate enough. Not the best at either, but the best balance we could find without buying a better GPU.

For the soil recommendation side of things, our first try was actually a small feedforward neural network. It scored worse than a basic Random Forest on the exact same NPK dataset. Tree-based models just handle messy tabular inputs better. pH at 6.5 sitting next to rainfall at 1200mm, the forest splits wherever it needs to split. No normalization step needed.

### C. Why existing solutions don't translate to India

We read a lot of papers before starting. Almost every one optimizes for Western farms. Big acreage, stable broadband, GPU servers in a data center somewhere. None of those assumptions hold for a 2-acre potato farm in Maharashtra where the only internet is a Jio 4G connection that drops every few minutes. Any system that needs constant server contact is broken in that context. Just broken.

Offline capability is treated as a feature request in most of these papers. For us it is the end goal. We kept inference on the server for now because converting to TFLite would have taken time we did not have, but we know that is where this has to go eventually.

And then there is the language problem that basically no one talks about. Every plant disease interface we looked at shows the raw disease name right on the screen. "Cercospora zae-maydis" on a results card. That is meaningless to a farmer who never studied plant pathology. We wrote plain-language descriptions for every disease class that our model can detect. What to look for, what to spray, in normal words. Small thing, but small things decide whether someone actually opens the app a second time.

Barbedo wrote about dataset bias back in 2018 [3] and people cite that paper constantly. But the recommended fixes are always about collecting more controlled images. Real photos from cheap Android phones have JPEG compression artifacts and inconsistent white balance and half the time another leaf is blocking the one you are trying to photograph. Our augmentation pipeline was designed

around exactly this kind of noise, not around making accuracy numbers look better on paper.

### III. HOW WE BUILT IT

Fig. 1 shows the overall layout. Two prediction pipelines running inside one Flask process. The image side works like this: user uploads a JPEG, React reads it as a binary blob, packs it into FormData, and Axios fires a multipart POST. On the server, PIL opens the file, resizes to 128x128, converts to a Float32 array, and hands it off to EfficientNetB0. The response is a JSON object with probabilities for all six disease classes. The soil pipeline is more straightforward. User types in NPK, pH, temperature, humidity, and rainfall. Flask passes those through a StandardScaler, the Random Forest does its thing, and a crop name comes back as JSON. No state is held between requests. Every POST contains everything the model needs.

MobileNetV2 sits right at the entrance of the image pipeline. If it recognizes a car or a face or a building at above 70% confidence, the request gets killed before the disease model ever loads the pixels. Both models live in memory from the moment Flask starts up, which eats about 800MB of RAM, but it means no request has to wait for a model to load. On one laptop running everything locally, that seemed like a fair trade.

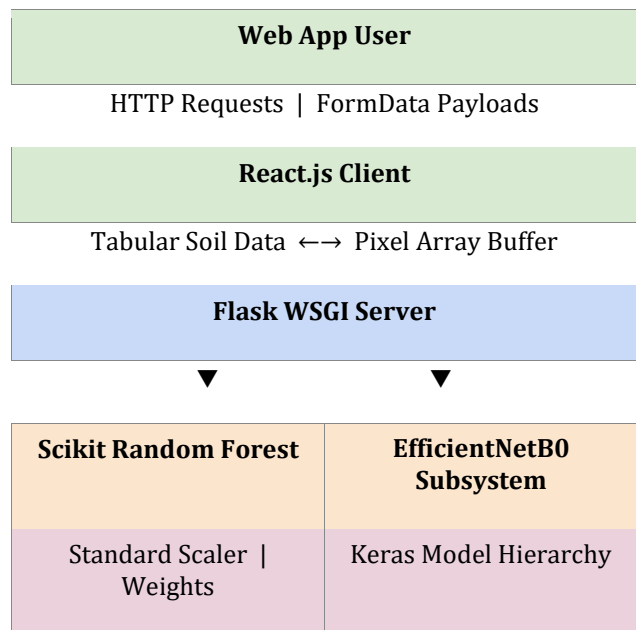


Fig. 1. End-To-End System Architecture of Mobile AgriNova Platform.

### A. The frontend

React 19 with Vite for bundling. We started with Create-React-App but hot reload was taking 8 to 10

seconds every time we saved a file. When you are sitting there fixing CSS bugs for three hours straight, that wait gets painful. Switching to Vite cut the reload time to under a second. Seems like a minor thing but over a long work session it made a real difference.

Image uploads were surprisingly painful. The flow sounds simple when you describe it: user picks photo, React reads the blob, renders a thumbnail, wraps it in FormData, sends it via Axios. Should have taken maybe an afternoon. Instead it took us two full days. Turns out certain Samsung phones embed EXIF orientation data in their JPEGs, so the image would arrive on the server rotated 90 degrees. We had no idea this was even a thing until we started Googling why the uploaded image was arriving sideways.

Validation on forms was another thing we did not think about until it bit us. Users would hit submit with blank input fields and Flask would crash on the null values. We added checks in the React components so the submit button stays grey until every required field has something in it. Server error rate went from roughly 40% of all requests down to basically zero after that. Felt disproportionately satisfying for what was really just a few if-statements.

### B. The backend and the crashes

We went with Flask, not Django. Django has a full ORM and an admin panel built in and we needed none of that. We needed something that could accept a JPEG and return JSON. Flask does that in 30 lines.

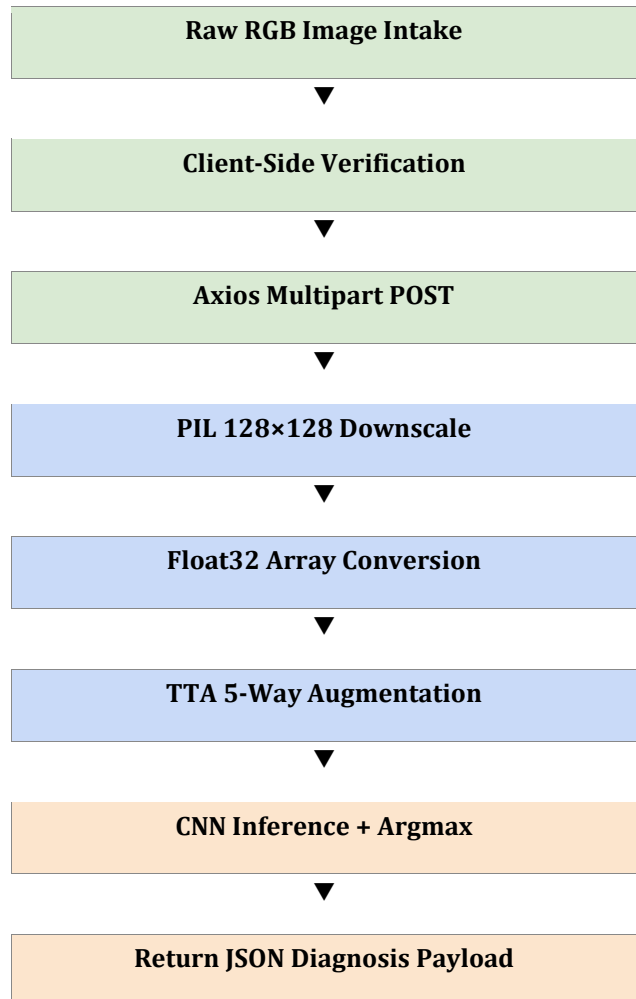
First real problem was CORS. React's dev server runs on port 5173, Flask listens on 5000, and browsers block cross-origin requests by default. Every API call just died silently. We spent about four hours going through Stack Overflow threads on Flask-CORS before getting the right header configuration. Looking back it seems obvious but at the time we had no idea what was happening.

The bigger disaster was model loading. Akash had put the `load_model('model.keras')` call inside the `/predict` route itself. Seemed reasonable. But that Keras file is 200MB and Flask was reloading it from disk on every single request, which meant a seven-second wait just to process one image. We did not realize how bad it was until Akash and Om both had the app open in separate browser tabs at the same time. The laptop has 4GB of GPU memory. Two loads running in parallel exceeded that and Python just crashed. Memory allocation error, no graceful fallback, the process was gone.

Once we figured out what was actually happening, the fix was not complicated. Move `load_model()` to the top of the script, outside any route. Flask loads it once on startup

and then `model.predict()` runs against the weights that are already in memory. Seven seconds went down to 1.2.

## IV. ENGINEERING THE DEEP LEARNING MODEL



**Fig. 2.** Granular Leaf Disease Image Processing Execution Pipeline.

### A. The training images

We pulled about 15,000 corn and potato leaf images from PlantVillage on Kaggle. CNNs need all inputs at the same resolution, so Keras ImageDataGenerator resized every image to 128 by 128 pixels before feeding it in.

Farm photos look nothing like dataset photos. Leaves are off-center, lighting varies from one image to the next, and sometimes there is literally a thumb covering part of the frame. We used data augmentation to make the model robust to this. Random rotation up to twenty degrees, horizontal flips, slight shearing at the edges. We deliberately left out vertical flips because leaves do not grow upside down in practice and flipping the lighting

gradient just confuses the convolutional filters. Whole point of augmentation here was to teach the model what a rust spot actually looks like, not just that a rust spot happened to always appear in the same position in the training set.

**B. The CNN setup**

Pre-trained EfficientNetB0 as the backbone. EfficientNet uses compound scaling, meaning it adjusts the network's depth and width and input resolution all at the same time rather than just adding more layers on top. Keeps inference speed manageable on consumer-grade hardware.

We chopped off the original classification head and replaced it with a Dense layer at 256 neurons followed by Dropout at 0.5. Dropout randomly zeroes out half the connections on each forward pass during training. Accuracy takes a hit in the short term but the model can not just memorize images anymore, which was the problem we kept running into. Whole thing ran for 10 epochs using Categorical Crossentropy as the loss function. Took about 38 minutes on our RTX 2050. We expected it to take longer honestly.

**C. The soil model**

Scikit-Learn Random Forest for the crop recommendations. Not a neural network. Random Forests build hundreds of individual decision trees and then average their votes together. One tree might say no to rice because rainfall is below 100mm. A different tree says no to wheat because nitrogen is too low. Averaging across all of them handles edge cases that any single tree would get wrong.

First version overfit badly. Training accuracy was 100%, test set was at 89%. Classic overfitting. We ran GridSearchCV over about 300 parameter combinations and found that setting max\_depth to 15 fixed it. Test accuracy went from 89% to 98.2% with that one change. Pickled the final model along with its StandardScaler weights so Flask could load both files on startup.

**V. RESULTS**

**A. What the numbers say**

EfficientNet landed at 96.5% validation accuracy. Higher than we expected going in. The confusion matrix showed one problem that kept coming up: early blight and normal age-related browning on potato leaves look almost identical. Brown patches, concentric ring patterns, same general shape. An agronomist can tell them apart by feeling the leaf texture but the model only sees pixels, so it gets them mixed up. The soil model came in at 98.2% after we ran grid search. Tabular classification is just a simpler

problem than image recognition so that number was not a surprise.

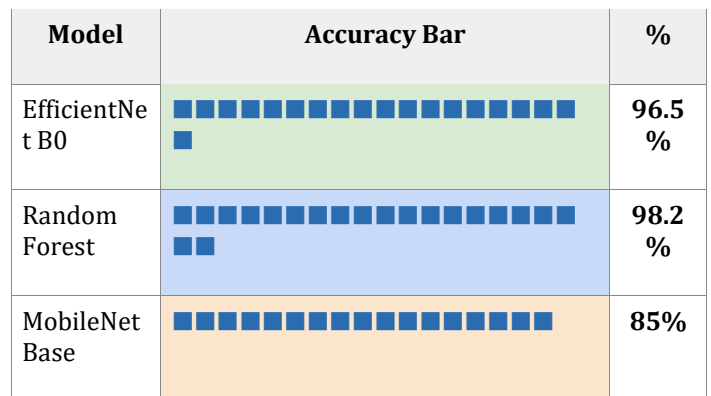


Fig. 3. Validation Accuracy Comparison Between Model Architectures.

**TABLE I. TRAINING PROGRESS OVER EPOCHS**

Epoch	Val Loss	Train Acc	Val Acc
1	0.218	91.7%	92.1%
5	0.061	97.9%	97.4%
10	0.028	99.1%	96.8%

Table II has the per-class breakdown. Weakest classes are Corn Northern Leaf Blight at 94.1% and Potato Late Blight at 93.5%. Makes sense when you look at the images. Both produce these elongated brown lesion patterns that look confusingly similar to each other, especially in early stages. The confusion matrix confirmed it, most misclassifications are between those two classes. Healthy leaf detection is close to perfect across both crops because a clean green leaf looks nothing like a diseased one. That part was easy for the model.

**TABLE II. PER-CLASS VALIDATION ACCURACY**

Class	Accuracy
Corn — Healthy	99.2%
Corn — Common Rust	97.5%
Corn — Northern Leaf Blight	94.1%
Potato — Healthy	98.9%

Potato — Early Blight	95.8%
Potato — Late Blight	93.5%

Table III has the latency comparison. MobileNetV2 is fastest at 8.7ms per inference but we did not reject it because of speed. The problem was that it kept swapping early blight and late blight on potatoes. Those two diseases require completely different fungicide treatments, so a wrong prediction there is worse than being slow. EfficientNetB0 at 14.2ms was the middle option. Not as fast as MobileNet, nowhere near as slow as ResNet50, and it does not confuse the two blight types nearly as often.

**TABLE III. MODEL INFERENCE LATENCY COMPARISON (RTX 2050)**

Model	Latency (ms)
ResNet50	22.4
MobileNetV2	8.7
EfficientNetB0	14.2

**B. The gatekeeper problem**

We found a real problem during testing. Upload a photo of a car dashboard and EfficientNet still classifies it as some crop disease. The softmax layer just picks whichever class has the highest score, even if that score is 12%. To deal with this we put MobileNetV2 in front of the disease model as a gatekeeper. MobileNetV2 was already trained on ImageNet so it can tell the difference between a plant and a car. If it tags a non-plant object at over 70% confidence, Flask rejects the image and tells the user to upload an actual leaf photo. Does not catch everything, some weird photos still get through, but it blocks the obvious cases.

**VI. SYSTEM INTERFACE**

Detection page has two panels. On the left there is the image upload area with a green "Analyze Crop" button and a "Clear" button below it. After the model returns a result, a card pops up showing the confidence as a big number, the disease name, crop type, and a badge that says HIGH, MEDIUM, or LOW confidence. Green progress bar fills proportionally. So for a Gray Leaf Spot detection at 94%, the card shows "Corn (Maize)" with "Diseased Leaf" and a HIGH CONFIDENCE badge next to it.

The right panel has a "Top Predictions" card that shows every class and its probability, not just the winner. Cercospora leaf spot 93.6%, Northern Leaf Blight 5.8%, Potato Early Blight 0.6%, and so on. We did this intentionally. If the model gives 52% to one class and 47% to another, the farmer can see that and know to get a second opinion rather than just trusting whichever class happened to edge ahead.

Bottom right has two info cards next to each other. One for symptoms, one for treatment. Both are written in simple English. We wrote "Small rectangular brown to gray lesions running parallel to veins" instead of something like "Cercospora zeae-maydis infection presenting as necrotic striations." Nobody reading this screen in a field has a pathology background. They want to know what to look for on the leaf and what chemical to buy. So that is what we tell them.

**VII. CONCLUSION**

Whole thing runs on one laptop with a budget GPU. No cloud services, no paid APIs. Getting from a Jupyter notebook to a real React and Flask application was harder than any of us expected. CORS debugging took a full day. The memory crash took two days to figure out. The EXIF rotation bug was another two days on top of that. If someone had told us at the start that connecting the frontend to the backend would be harder than training the neural network, we probably would not have believed them. But it was.

It works now though. Open the site on a phone browser, upload a leaf photo, and a diagnosis comes back in 1.2 seconds. Soil recommendations are faster than that. For a semester project built by four students, we are all pretty satisfied with where it ended up.

**VIII. WHAT WE WOULD DO NEXT**

If we had more time, first thing would be Docker. Setting up the Python environment from scratch on a new machine takes about 45 minutes because TensorFlow versions keep conflicting with NumPy versions and CUDA versions. Docker would make the whole deployment a single command. We would also add Marathi and Hindi translations for the interface because the farmers who need this most are not reading English confidently enough to act on disease names like "Septoria leaf spot."

Real end goal is TFLite. The Keras model is 200MB. TensorFlow Lite quantization would compress that to maybe 15 or 20MB, small enough to ship inside an Android APK. At that point the phone handles inference locally. No Flask server, no internet connection required. That is where this project needs to end up if it is going to actually help anyone in a village where cell coverage is unreliable.

We also want to build a feedback loop. Right now the model is trained once and never updated. If a farmer could tap a button that says "this diagnosis was wrong" and submit a correction, those corrected photos could go into a retraining queue. The MobileNetV2 gatekeeper would keep junk out of that queue. We did not have time to build this before submission and it would need a database layer that we currently do not have, but the concept is straightforward enough that a future team could add it.

Crop coverage is the last thing. PlantVillage has labeled images for 14 crops. We trained on two, corn and potato, because of the semester deadline. Wheat, rice, and tomato would cover most of what gets grown in Maharashtra. The code changes would be minimal, just add the image folders and bump the output layer from 6 neurons to however many classes the new dataset has. Not a hard engineering problem, just a time problem.

## REFERENCES

- [1] A. Kamilaris and F. X. Prenafeta-Boldú, "Deep learning in agriculture: A survey," *Computers and Electronics in Agriculture*, vol. 147, pp. 70-90, 2018.
- [2] D. J. Mulla, "Twenty five years of remote sensing in precision agriculture: Key advances and remaining knowledge gaps," *Biosystems Engineering*, vol. 114, no. 4, pp. 358-371, 2013.
- [3] J. G. A. Barbedo, "A review on the main challenges in plant disease data sets for machine learning," *Frontiers in Plant Science*, vol. 9, p. 1531, 2018.
- [4] S. P. Mohanty, D. P. Hughes, and M. Salathé, "Using deep learning for image-based plant disease detection," *Frontiers in Plant Science*, vol. 7, p. 1419, 2016.
- [5] E. You, "Vite: Next Generation Frontend Tooling," vitejs.dev. [Online]. Available: <https://vitejs.dev>.
- [6] R. Behmann, A. K. Mahlein, T. Rumpf, C. Römer, L. Plümer, "A review of advanced machine learning methods for the detection of biotic stress in precision crop protection," *Precision Agriculture*, vol. 16, pp. 239-260, 2015.
- [7] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," arXiv preprint arXiv:1704.04861, 2017.
- [8] M. Tan and Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," *International Conference on Machine Learning (ICML)*, 2019, pp. 6105-6114.
- [9] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5-32, 2001.