AgriNexus: An Intelligent Framework for Weather-Based Crop Recommendation and Agricultural Market Analysis

1st Maseeha Bhanu
Assistant Professor, Dept. of CSE (AI&ML)
Malnad College of Engineering
Hassan, India
mb@mcehassan.ac.in

2nd Akshatha M M
B.E., Dept. of CSE (AI&ML)
Malnad College of Engineering
Hassan, India
akshathamm1813@gmail.com

3rd Impana R H
B.E., Dept. of CSE (AI&ML)
Malnad College of Engineering
Hassan, India
impana17gowda@gmail.com

4th R Deepika Patil
B.E., Dept. of CSE (AI&ML)
Malnad College of Engineering
Hassan, India
deepudollylucky@gmail.com

5th Yashaswini M N
B.E., Dept. of CSE (AI&ML)
Malnad College of Engineering
Hassan, India
yashaswiniyashu2212006@gmail.com

***Abstract*—Choosing what to grow means looking at soil, water, weather and prices together, yet farmers usually have to search for each of these in a different place. AgriNexus, a Farmer Helper Chatbot created for Karnataka, brings crop recommendations, weather updates and market prices into one web application. The server is written in Python with Flask, the pages are built with HTML and CSS, SQLite stores structured records, and Pandas loads the price data. A farmer enters the district, taluk, village, land area, soil type and water availability; the system then scores a set of crops using hand-written rules and lists the top matches with short explanations. Weather data is fetched from the OpenWeather API, and prices are read from a CSV file. No machine-learning model has been trained yet, so the quality of the recommendations depends on how well the rules match real conditions, the weather display depends on the external service being reachable, and the price display is only as up to date as the CSV file. The tool is meant for learning and early planning, not as a replacement for advice from agricultural experts. We outline the motivation, related work, design, method, implementation, a testing plan, limitations and future directions. No performance figures are reported because a formal evaluation has not yet been conducted.**

Keywords—Agricultural decision support, crop recommendation, weather information, market prices, Karnataka, Flask, rule-based system.

I. INTRODUCTION

Farming decisions rarely depend on one thing alone. The soil decides what can be planted, the water on hand limits the options further, and the weather shapes both the work in the field and the growth of the crop. Prices matter too, because a farmer wants to know roughly what the harvest will earn. In reality this information is scattered across government portals, traders, neighbours and personal memory, and putting it together is slow, especially when each source presents things in its own way.

A web application that collects the basics in one place can ease that burden. The system described here, AgriNexus (built as the Farmer Helper Chatbot for Karnataka), takes a few details about the farmer's location and land, uses a set of predefined rules to rank suitable crops, and displays current weather from an external service next to price records from a CSV file.

We call the interface a chatbot because it talks the user through the steps, but the crop suggestions are not learned from data, since no model has been trained on any dataset. The scores only put crops in order according to the configured conditions; they do not tell the farmer how likely a crop is to succeed. How helpful the results are depends on how closely the rules match the needs of each crop and the conditions in the field.

The work had four objectives. The first was to collect location and land details through a simple form. The second was to rank crops with one consistent scoring method and show the reasons behind each ranking. The third was to include weather data from the OpenWeather API. The fourth was to display market data from a structured dataset, so that all three kinds of information can be viewed in one place.

The main contribution is a working software prototype that shows how a Flask application can bring together user input, rule-based logic, an external API, a local database and tabular price data. We are not saying that its suggestions are better than those of experts or other tools. Such a claim would need a clear baseline, verified data and proper testing.

The rest of the paper is organised as follows. Section II reviews related work. Section III describes the architecture, and Section IV explains the method, including how inputs are handled, how weather is collected, how crops are scored and how market data is used. Section V lists the technologies, Section VI discusses how the system should be tested and where its limits lie, and Sections VII and VIII cover future plans and conclusions.

II. LITERATURE SURVEY

A. Digital Technologies in Agriculture

Digital agriculture relies on computers, sensors and communication tools to help manage farms. Wolfert et al. [1] examined how big data is used in smart farming. They noted that linking information along the whole farming chain can help, but that poor data quality and slow real-world adoption remain problems. Their conclusions encourage tools that combine several data sources, although the advantage only appears when the data are reliable, fit the farmer's local situation and come with enough background to make sense.

B. Crop Recommendation Approaches

Crop recommenders can be built from rules, statistical models or machine learning. The usual inputs are soil characteristics, weather, region, water supply and past farm records. Liakos et al. [2] reviewed machine learning across many agricultural tasks and showed how often it is used. These models still need enough training data, a good evaluation plan and proof that the results hold up on new situations. Rule-based methods are different because the conditions are written out directly, so it is easy to see why a particular crop is recommended.

 ©IJSRED: All Rights are Reserved

AgriNexus follows the rule-based path. It gives a starting point that can be improved as more domain knowledge becomes available. The downside is that rules can miss factors or assume conditions that are not true in every district and season, so the results should be seen as a first suggestion until experts have reviewed them and they have been tested on local cases.

C. Weather Services for Decision Support

Weather data gives useful background for planning, but it has to be read carefully. Weather APIs provide current conditions, recent rainfall or forecast values through separate endpoints, and each field has its own meaning and time window [3]. For example, rainfall for the last hour should not be shown as a daily or seasonal total, and a present reading does not replace a forecast or a long-term climate study.

Our application pulls weather from the OpenWeather API, so we do not have to keep our own observation records. The downside is that it needs internet access, an API key and an online service, and it is subject to request limits. If a request fails and default values are used instead, the interface must label them as defaults and not as live data.

D. Market-Price Information

Showing prices on screen lets users see what has been recorded, but only if the numbers can be understood. A price without its commodity, market, date and unit can be confusing. Likewise, a static CSV file does not become a live feed just because a web page displays it. How fresh it is depends on where the records come from and how often the file is updated.

In AgriNexus the price data are read from a CSV file with Pandas. This keeps the design simple for a project of this size and lets the file be opened or replaced easily. Before real use, the data should be checked for source, units, duplicates, gaps and update dates, and if the file contains demonstration data, that must be stated clearly.

E. Research Motivation

Taken together, these themes make a good case for an integrated prototype. Field inputs, weather and prices can share one interface, and explicit rules keep the crop ranking easy to inspect. We are not proposing a new prediction algorithm; the focus is on bringing things together, keeping them easy to use, and building a workflow that can later be tested and improved with verified agricultural data.

III. SYSTEM ARCHITECTURE

AgriNexus is built as a modular web application. Farmers work through a browser interface, and a Flask backend takes their requests and passes them to the right component. There are six parts in all: the user interface, the request-handling layer, the rule-based recommendation module, the weather integration, the SQLite data layer and the CSV-based market module.

A. User Interface

The front end is written in HTML and CSS. It has forms for entering location and land details, plus separate pages for recommendations, weather and market records. Every field carries a clear label and unit so nobody has to guess what to type, and when something is missing or wrong the message says exactly what to fix instead of showing a raw technical error.

B. Flask Backend

Flask handles the routes and connects the pages to the logic behind them. It receives the submitted values, makes sure the required fields are filled in, calls the recommendation routine and sends back the finished page or an API response. Since request handling is kept separate from the scoring rules, each function can be updated and tested on its own.

C. Recommendation Module

This module compares every candidate crop against predefined conditions. Depending on the rules in place, the conditions may involve soil type, water availability, location and other user inputs. It computes a score for each crop, sorts them, and returns a short list with an explanation for each entry. The precise rules and weights should be documented directly from the source code.

D. Weather and Data Components

The weather component calls the OpenWeather API with a key kept outside the source code. SQLite holds structured application records wherever the project needs them, and the market module loads the configured CSV with Pandas before applying its filtering and display logic. Each of these sources has its own life cycle: weather arrives over the network on request, database records persist locally, and price records change only when someone updates the CSV.

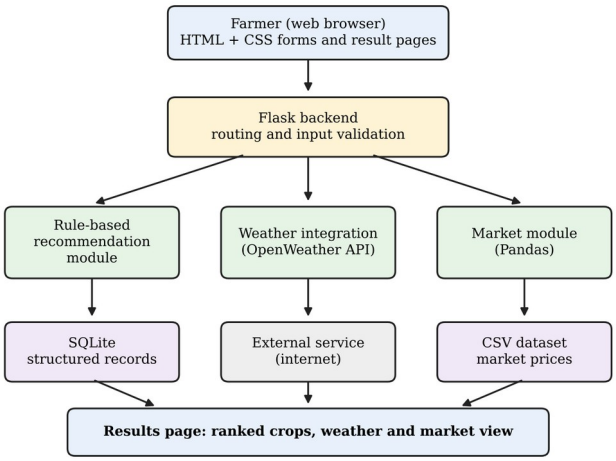


Fig. 1. System workflow of AgriNexus.

The design should also keep the origin of data visible. Weather output should reflect whether the API call succeeded, and market records should carry their verified source and update date. Wherever default weather values are used, they need to be clearly distinguished from retrieved ones.

IV. METHODOLOGY

A. Input Collection and Validation

The application asks for the location and field details that its rules need: district, taluk, village, land area with its unit, soil type and water availability. The previous crop and soil-test status can be added optionally. Soil can be chosen as red, black, sandy, clay, loamy or unknown, and water availability as good, medium, low or rainfed. All inputs are validated before scoring. Land area, for instance, must be a valid positive number, categorical fields are limited to the supported options, and blank optional fields must not cause failures. When a value is unknown, the scoring step should not quietly assume a particular soil or water condition.

 ©IJSRED: All Rights are Reserved

The form groups location fields separately from land characteristics. In the demonstrated entry, the user selects Hassan district and supplies the taluk and village before entering the cultivated area, soil type and available water.

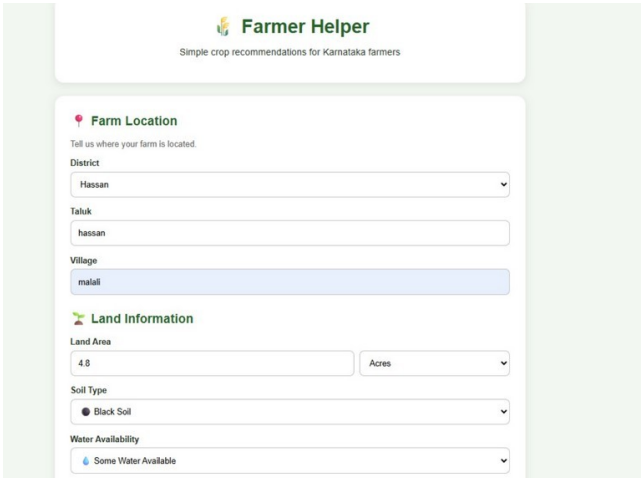


Fig. 2. Farmer Helper form for entering the farm location and land details.

The previous-crop field and soil-test response provide additional context for interpreting the recommendation. The weather panel also explains that the application retrieves local conditions automatically, so the farmer does not need to enter weather measurements manually.

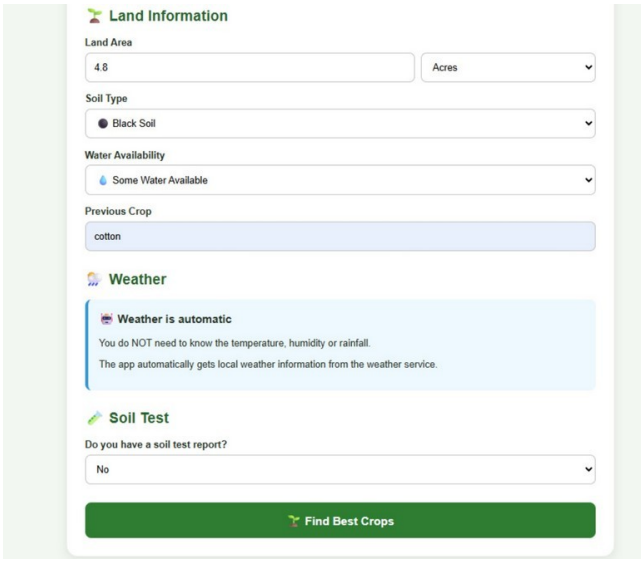


Fig. 3. Remaining farm inputs, weather guidance and soil-test selection.

B. Weather Retrieval

On a weather request, the backend contacts the configured OpenWeather endpoint and parses the response. The values shown can include temperature, humidity and the last hour's precipitation when the response contains it, although the exact fields vary with the endpoint. Wherever possible, the page should state the location and time to which the reading refers.

Retrieval can fail for several reasons: a network problem, an invalid key, a rate limit or a service that is simply down. In that case the prototype may fall back on default values, but it labels them as defaults and never passes them off as measured weather. A production version would also log why the call failed, without exposing any credentials, and show the user a friendly message.

The interface presents temperature, humidity, rainfall and the current condition as separate weather indicators. These values give the farmer a quick view of the retrieved conditions before considering the crop ranking.



Fig. 4. Weather information displayed with the crop recommendation page.

C. Rule-Based Crop Scoring

Let C = {c1, c2, …, cn} be the set of candidate crops and x the vector of user inputs. For each crop ci the system checks a group of predefined rules and combines the outcomes into a weighted sum:

$$Score(c_i) = \Sigma_j \, w_j \cdot r_j(c_i, x) \quad (1)$$

where rj is the outcome of rule j and wj is the weight configured for it. This formula is a conceptual description and would need to be adjusted if the implemented calculation differs. Crops are sorted by score and the highest-ranked ones are shown. The reason displayed next to each crop must correspond to conditions that actually matched; for instance, the system must not claim a crop suits a soil type unless the rules encode that relationship. Scores are ranking values and must not be read as probabilities, yield forecasts or accuracy percentages. To judge this component, a reviewed set of input cases with expected recommendations is required.

For the illustrated farm entry, the displayed ranking places Tur (pigeon pea) and cotton above the next option. Each card includes a short suitability explanation and a score, allowing the user to see the reason for the ordering rather than receiving only a crop name.

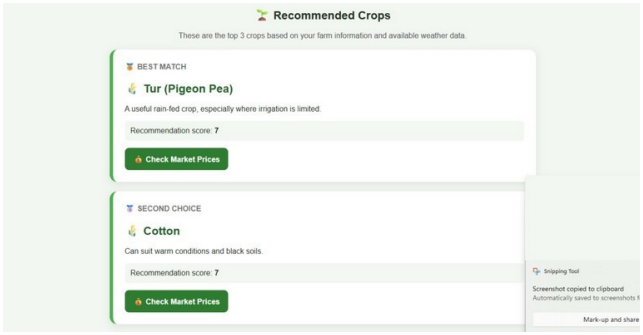


Fig. 5. Crop recommendation cards with rank, explanation and score.

The final recommendation card is followed by a notice clarifying that suitability guidance is not a guarantee of yield or profit. This distinction is important because the current prototype uses configured rules rather than a trained predictive model.

 ©IJSRED: All Rights are Reserved

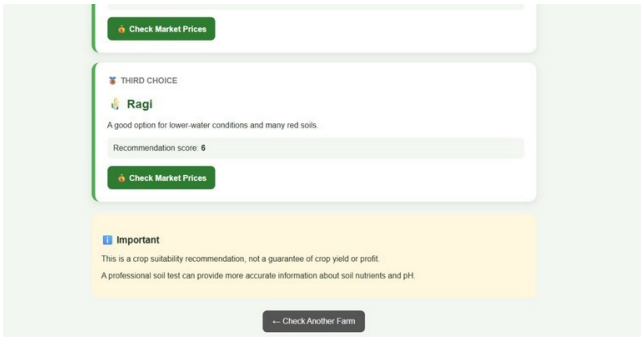


Fig. 6. Further recommendation details and the crop-suitability notice.

D. Market-Data Processing

The market page loads the configured CSV through Pandas, optionally filters the records by crop or another field, and displays what matches. The pipeline should confirm that the expected columns exist, cope with empty or malformed files, and retain the meaning of the original values, including price units and dates.

Source and freshness remain the main limitations of this module. Data that are illustrative or prepared by hand must be labelled as sample data. If official records are adopted in a later version, each entry should store its source, retrieval date, market, commodity and unit so that users can interpret it correctly.

The market view compares the listed modal prices across the displayed markets and estimates travel expense and net profit using the entered harvest, cultivation-cost and transport assumptions. The page explicitly identifies the displayed prices as demo samples, so the estimates should not be treated as current market quotations.

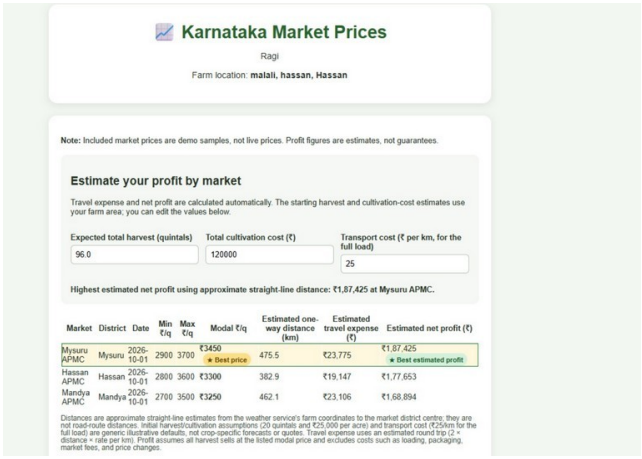


Fig. 7. Market comparison and illustrative profit estimate for the selected crop.

E. Security and Reliability Considerations

The OpenWeather key is kept in an environment variable and stays out of public repositories. Errors are handled so that secrets, file paths and stack traces never reach the user, and the application copes with missing inputs, unavailable services and absent data files without crashing. These steps make the system sturdier, but they do not tell us whether the recommendations themselves are agronomically sound.

V. IMPLEMENTATION

The server side of the prototype is written in Python with Flask [4], while HTML and CSS provide the browser interface. SQLite stores structured records where required, Pandas [5] processes the market-price CSV, and OpenWeather [3] supplies weather data whenever the external request succeeds. Table I lists the technologies and what each one does.

TABLE I
TECHNOLOGIES USED AND THEIR ROLES

Layer	Technology	Role
Backend	Python, Flask	Routing, request validation and application logic
Frontend	HTML, CSS	Input forms, navigation and display of results
Structured data	SQLite	Stores structured records where used
Data processing	Pandas	Reads and filters market-price CSV records
Weather service	OpenWeather API	Retrieves weather observations
Market data	CSV dataset	Provides the displayed market records

The code base is split into backend modules, templates, static files and a data folder, so the interface or the data can be changed without disturbing everything else. Credentials stay outside the repository, all required Python packages are listed, and the SQLite database is created with a schema that matches the current code.

The report page brings the submitted farm information into a compact summary, making it easier to review the selected district, taluk, village, land area, soil and water availability alongside the weather section. It serves as a final overview of the information used during the recommendation workflow.



Fig. 8. Consolidated farm report summarizing farm details and local conditions.

VI. RESULTS AND DISCUSSION

A. Functional Validation

A proper functional test plan would cover valid inputs, missing required values, every soil type, every water-availability level, successful and failed weather calls, crop ranking, market filtering, and missing or malformed CSV files. Each test records the input, the expected result, the observed result and a pass or fail verdict, so anyone can repeat it and see whether the system behaves as intended.

The final version of this paper should state how many tests were run, how many passed and what failures were found. We do not give a test count, accuracy or success rate here because no such measurements have been recorded yet. If a labelled agronomic test set is built, agreement with expert-reviewed

 ©IJSRED: All Rights are Reserved

recommendations can also be reported, together with an explanation of how the cases were chosen.

B. Response-Time Evaluation

Response time should be measured separately for crop scoring, market-data loading and weather retrieval. Scoring runs locally, while weather retrieval depends on the network and an outside service, so lumping them together would hide where the delay comes from. A good report names the test machine, the network conditions, the number of repeated runs and the mean or median time, because one unrecorded run proves nothing.

C. Qualitative Comparison

Table II contrasts the prototype with looking up each kind of information separately.

TABLE II
COMPARISON WITH SEPARATE MANUAL LOOKUP

Feature	Separate manual lookup	Proposed prototype
Crop ranking	User compares information manually	Rule-based crop ranking
Weather information	Needs a different source	OpenWeather API integration
Market records	Needs a different source	Local CSV-based display
Recommendation explanation	Left to the user's interpretation	Reasons tied to matched rules
Data freshness	Depends on the source	Depends on the API and the CSV update process

This comparison covers design characteristics only and does not show that the prototype is more accurate, faster or more useful than other systems. A stronger study would define the competing approaches, apply identical test cases and measure the same outcomes.

D. Limitations

The ranking component is limited by how completely and correctly its rules are written. It may overlook factors such as crop variety, sowing season, pest pressure, input costs, irrigation facilities, local agronomic recommendations or sudden extreme weather. The weather feature depends on the API being available and may give only current conditions, not forecasts, and default values can mislead unless they are labelled. The market module relies on the quality and freshness of the local CSV; without confirmed origin and update information, the prices shown should be regarded as illustrative and not authoritative.

In addition, the prototype still needs usability testing with real users and expert review of its rules. It is best seen as an educational decision-support tool and not as a replacement for agricultural extension services or professional advice.

VII. FUTURE SCOPE

There is plenty of room to make the system more useful and dependable. The first step would be linking the market module to a verified official source with a documented update schedule, so that every record shows its commodity, market, date, unit and source. Validation checks could also catch missing values, duplicates and inconsistent units before anything reaches the screen.

Second, the weather module could include forecasts and seasonal rainfall context, keeping observed and forecast values visibly separate, and the interface could report failed requests instead of silently showing default data as current. Third, agricultural experts could review the rules, which could then be refined with region- and season-specific guidance. Fourth, support for Kannada and other regional languages, a mobile-friendly layout, a simpler input flow and optional voice interaction would improve accessibility. Fifth, later evaluation could rely on a documented set of test cases with expert-reviewed expected outcomes, with metrics chosen to suit the task and all test conditions reported.

Finally, a trained machine-learning model could be explored if the team gains a verified dataset with relevant features and reliable labels. That step would call for a documented train/test split, suitable metrics, comparison with the current rule-based baseline and checks for data leakage. It should not be described as part of the present system unless it is actually built and evaluated.

VIII. CONCLUSION

This paper presented AgriNexus, a web-based Farmer Helper Chatbot for Karnataka that combines rule-based crop recommendation, weather information and market-price display. The backend runs on Python and Flask, the interface on HTML and CSS, SQLite stores structured records, Pandas processes the CSV data and the OpenWeather API provides weather. The system takes basic location and field details, ranks candidate crops through predefined rules and shows the results together in one interface.

The prototype shows that several agricultural information services can live together in one lightweight application. Still, its suggestions are only as good as its rules, its weather view relies on an outside service, and its prices are only as trustworthy as the dataset behind them. Until more implementation and verification are done, it should not be called a trained machine-learning model or a live official price service. Our next steps are expert validation, verified data sources, documented functional tests and measured performance, all before anyone relies on the tool for real decisions.

ACKNOWLEDGMENT

The authors thank the Department of CSE (AI&ML) at Malnad College of Engineering for the facilities and academic support that made this mini-project possible. They also thank their project guide and the faculty members for their advice and feedback throughout the design and implementation of the application.

REFERENCES

[1] S. Wolfert, L. Ge, C. Verdouw, and M.-J. Bogaardt, "Big data in smart farming—A review," Agric. Syst., vol. 153, pp. 69–80, 2017.

[2] K. G. Liakos, P. Busato, D. Moshou, S. Pearson, and D. Bochtis, "Machine learning in agriculture: A review," Sensors, vol. 18, no. 8, Art. no. 2674, 2018.

[3] OpenWeather, "Current weather data: API documentation." [Online]. Available: https://openweathermap.org/current. Accessed: Oct. 2026.

[4] Pallets Projects, "Flask documentation." [Online]. Available: https://flask.palletsprojects.com/. Accessed: Oct. 2026.

[5] The pandas development team, "pandas documentation." [Online]. Available: https://pandas.pydata.org/docs/. Accessed: Oct. 2026.

 ©IJSRED: All Rights are Reserved