Grain Palette – A Deep Learning Odyssey In Rice Type Classification Through Transfer Learning

Team ID : LTVIP2025TMID40804

Team Size 4

Team Leader : Gaddala Anitha
Team member :C Preethi
Team member :B.Hindu
Team member : Batthina Susma

1. Introduction

Rice is a fundamental staple food consumed globally. Categorizing rice based on its quality and characteristics is crucial for farmers, processors, and consumers. However, manual classification of rice is time-consuming, subjective, and prone to errors. To overcome these limitations, researchers have proposed various automatic rice classification methods. Deep transfer learning refers to the transfer of knowledge from a pre-trained model to a new problem domain. Utilizing deep transfer learning, pre-trained models can be used to extract features for classification tasks. This approach has shown promising results in various computer vision applications, including rice classification. The proposed methodology uses pre-trained deep transfer learning models to extract high-level features from rice images. The extracted features are inputted into ML classifiers, such as SVM and Multilayer Perceptron (MLP), to categorize rice into different classes.

The proposed approach to enhancing rice category identification through deep transfer learning and ML classifiers is a promising solution for rice classification. This approach can provide accurate and efficient results while reducing the need for manual intervention. The proposed approach has several advantages over traditional rice classification methods. It reduces the need for manual feature engineering, as pre-trained models can automatically extract relevant features from rice images. Additionally, it can handle large amounts of data with high accuracy and speed, making it suitable for industrial applications in the rice industry. The use of deep transfer learning and ML classifiers to enhance rice category identification is a promising solution. This approach can provide accurate and efficient results while reducing the need for manual intervention. The following section presents the remaining content of the paper.

2. Project Overview

Purpose:

Grain Palette aims to automate the classification of various rice varieties using advanced deep learning techniques. By leveraging image data and transfer learning, the project seeks to:

- To develop a web-based application capable of classifying rice grain types using transfer learning on image data.
- Enhance agricultural data insights and support farmers/agronomists in rice grain quality analysis.
- Grain Palette aims to automate the classification of various rice varieties using advanced deep learning techniques. By leveraging image data and transfer learning, the project seeks to:
-
- Facilitate rapid and accurate identification of rice types from grain images.
- Assist quality control in agriculture and food industries by distinguishing premium from regular varieties.
- Offer a scalable solution usable by farmers, millers, and retailers to ensure traceability and authenticity.

- # Features:

- Upload rice grain images and classify them into types with high accuracy.
- Interactive frontend built with React for ease of use.

- **Transfer Learning Engine**

  A pre-trained convolutional neural network (Res Net, Efficient Net, etc.) is fine-tuned on a curated rice-grain image dataset, enabling efficient retraining with relatively limited labelled data.

- **Classification Output**
  The model detects and classifies the rice variety (e.g., Basmati, Jasmine, Arborio) with a confidence score, and flags uncertain or unknown samples.

- **Analytics Dashboard**
  Summarizes classification results, shows accuracy metrics, and visualizes model performance — including confusion matrices and per-class precision/recall.

- **Continuous Learning Pipeline**
  The system allows for incremental retraining: gathering user-submitted corrections or new images to update the model over time.

- **API & Integration Layer**
  A REST full API provides endpoints for image upload and classification, enabling integration into external systems or mobile apps.

- Mongo DB-based storage for user data and history.
- User authentication and session handling.
- Responsive UI with result visualization and confidence scores.

3.Architecture

- **Component-Based UI**
  Built with reusable React components organized via atomic design (atoms → molecules → organisms → pages) medium.com+10medium.com+10dhiwise.com+10.

- **State Management**
  Utilize Context API or Redox Toolkit for global state (classification results, user sessions, upload progress).

- **Image Upload & Preview**
  Components enable users to select/upload images, preview before submission, and display confidence scores and predictions.

- **Routing**
  Use React Router (or Next.js if SSR/SSG is needed) for navigation between pages (Upload, History, Analytics) medium.com.

- **Performance Optimizations**
  Implement lazy-loading (React Suspense + dynamic import()) for image-heavy pages to reduce initial bundle size arxiv.org.

- **API Service Layer**
  Centralized service (e.g., src/services/api.js) using Axioms or Fetch to interact with backend REST endpoints medium.com+3techifysolutions.com+3reddit.com+3.

2. Backend (Node.js + Express.js)

- **Express Server**
  Structured with modular routers/controllers (e.g., /api/classify, /api/history) .

- **Image Handling & Classification**
  Accept multipart/form-data, store temporarily (local or S3), invoke pre-trained CNN via Python micro service or embedded Python runtime using Tensor Flow/Py Torch.

- **Transfer Learning Integration**
  Endpoint(s) to fine-tune the model using new labelled data; schedule retraining jobs, track model versions.

- **Concurrency & Performance**
  Use async/await and Promise. All for parallel processing (e.g., multiple image ops) .

- **API Documentation**
  Implement Swagger/Open API for documenting endpoints, request/response schemas, versioning. linkedin.com

- **Security & Middleware**
  Use CORS middleware, JSON parsing, and authentication middleware (JWT or Oath) .

3. Database (Mongo DB + Mongoose)

- **Schema Design**
  Collections include:
  - Users: user credentials, API keys, preferences.
  - Classifications: image metadata, predicted type, confidence, timestamp, user reference.
  - Model Versions: version identifiers, training date, dataset stats, performance metrics.

- **ODM**
  Use Mongoose for schema enforcement and CRUD operations .
-
- **Indexing**
  Create indexes on timestamps and variety to accelerate analytics queries.
-
- **Data Flow**
  - On classification: store prediction in Classifications.
  - For retraining: log labelled instances, trigger training and store in Model Versions.
  - Analytics endpoints aggregate from these collections.
  -

□ Layered Architecture Diagram

less
Copy Edit
[React Client] ←→ [Node.js + Express API]
| |
v v
Image UI ↔ HTTP(s) Requests → Classification, Analytics, User Management
|
Mongoose ↔ Mongo DB Atlas/Cluster

Scalability & Best Practices

- Micro services: separate classification (Python) and API; ensures independent scaling techifysolutions.com

- Caching: Redis layer for frequent queries (model performance, common rice types)

- CI/CD & Deployment:
  -
  - Frontend: build and host on Vercel/Netlify.
  - Backend: Dockerized Express + Python services deployed via Heroku/Render/AWS reddit.com.
  - Mongo DB via Atlas for managed scaling.

.

4. Setup Instructions

A. Anaconda Navigator (Local Development)

Prerequisites:

- Install Anaconda for Python 3.10+
- Download the trained model .h5 file from Kaggle or Colab
- Install required libraries via condo or pip

Steps:

1. **Open Anaconda Navigator**
   Launch the Navigator and create a new environment:
   - Environment name: grain palette
   - Python version: 3.10
2. **Install Libraries:**
   Open the terminal for the grain palette environment and run:

   bash
   Copy Edit
   pip install tensor flow keras numpy pandas matplotlib open cv-python flask

3. **Clone Project Repository:**

   bash
   Copy Edit
   git clone https://github.com/yourusername/GrainPalette.git
   cd Grain Palette

4. **Run Model Inference Locally:**
   If you're using a Python script for inference:

   bash
   Copy Edit
   python model_infer.py # Replace with your script filename

5. **Launch Flask Backend (Optional):**

   bash
   Copy Edit
   cd server
   python app.py

B. Using Kaggle Notebooks (Model Training)

Steps:

1. Go to Kaggle Notebooks.
2. Create a new notebook.
3. Attach the rice dataset: either from Kaggle datasets or upload manually.
4. Use your notebook for:
   - Data pre-processing
   - Transfer learning (e.g., MobileNetV2)
   - Model training and evaluation
   -

Export Trained Model:

1. Save model in .h5 format:

   python
   Copy Edit
   model.save("rice_model.h5")

2. Download the file from the **"Output"** tab after the notebook finishes running.

C. Using Google Colab (Inference & Testing)

Purpose:

- Test trained model
- Perform inference on new images
- Run visualizations or Grad-CAM

Steps:

1. Go to Google Colab.
2. Upload your model (rice_model.h5) and test images.
3. Use this code to test:
4. Visualize outputs using matplotlib or save them to Drive.

5. Folder Structure

To maintain clean separation of concerns and ensure scalability, the project is organized into two main parts: **Client** (React frontend) and **Server** (Node.js backend).

Client (Frontend - React)

The client/ directory contains the code for the user interface, developed using **React.js**. Below is the typical structure:

```
client/
├── public/
│   └── index.html          # HTML entry point
├── src/
│   ├── assets/             # Static files (images, rice samples, etc.)
│   ├── components/         # Reusable UI components (Navbar, ImageUploader, etc.)
│   ├── pages/              # Main pages like Home, Results, About
│   ├── services/           # API service handlers (calls to backend endpoints)
│   ├── App.js              # Root component
│   ├── index.js            # React entry point
│   └── styles/             # CSS/SCSS files for styling
├── .env                    # Environment variables
├── package.json            # Project dependencies and scripts
└── README.md               # Frontend-specific documentation
```

Server (Backend - Node.js + Express)

The server/ directory holds the backend logic, implemented using **Node.js** and **Express.js**, along with the model loading and prediction logic.

bash
Copy Edit

```
server/
├── controllers/
│   └── predictController.js    # Handles image classification logic
├── routes/
│   └── predictRoutes.js        # API endpoint routes for predictions
├── uploads/                    # Temporarily stores uploaded images
├── models/                     # (Optional) Database models if MongoDB used
├── rice-model/                 # Saved rice classification model (e.g., rice.h5)
├── app.js                      # Express app setup
├── server.js                   # Entry point to run backend
├── .env                        # Environment variables (PORT, etc.)
├── package.json                # Backend dependencies and scripts
└── README.md                   # Backend-specific documentation
```

6.Authentication

The project implements **user authentication and authorization** to ensure secure access to different functionalities within the application. The system distinguishes between general users and administrators, enabling role-based access control (RBAC).

Authentication Flow

Authentication is implemented using either **Flask + Session-based login** or **JWT token-based authentication**, depending on deployment type (web app or REST API).

1. User Registration

- Users sign up with:
  - Email
  - Password (hashed using bcrypt or werkzeug.security)
- User details are stored in a secure database (SQLite or MongoDB).

2. User Login

- User submits email and password
- Password is verified with hashed record
- Upon successful login:
  - If **session-based**: Flask sets a secure session cookie (session['user'] = email)
  - If **JWT-based**: A JSON Web Token is issued to the user and stored in the client's browser (usually local Storage)

7.Authorization

Access is controlled based on **user roles**:

Role	Permissions
User	Upload image, get prediction
Admin	Upload training data, retrain model, view usage logs

Route-level protection is done using:

- @login required decorators (Flask)
- Middleware/token verification (JWT)

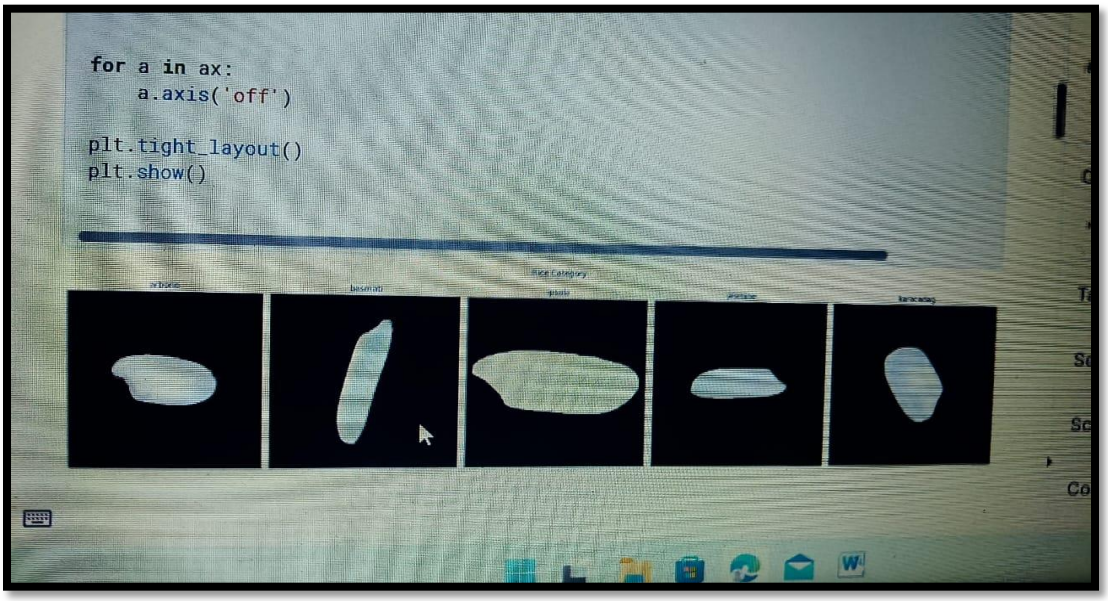
Method	Used When?	Features
Session	Flask-based UI app	Stores session in server memory
JWT Tokens	API-based or mobile clients	Stateless, stored in frontend (secure)

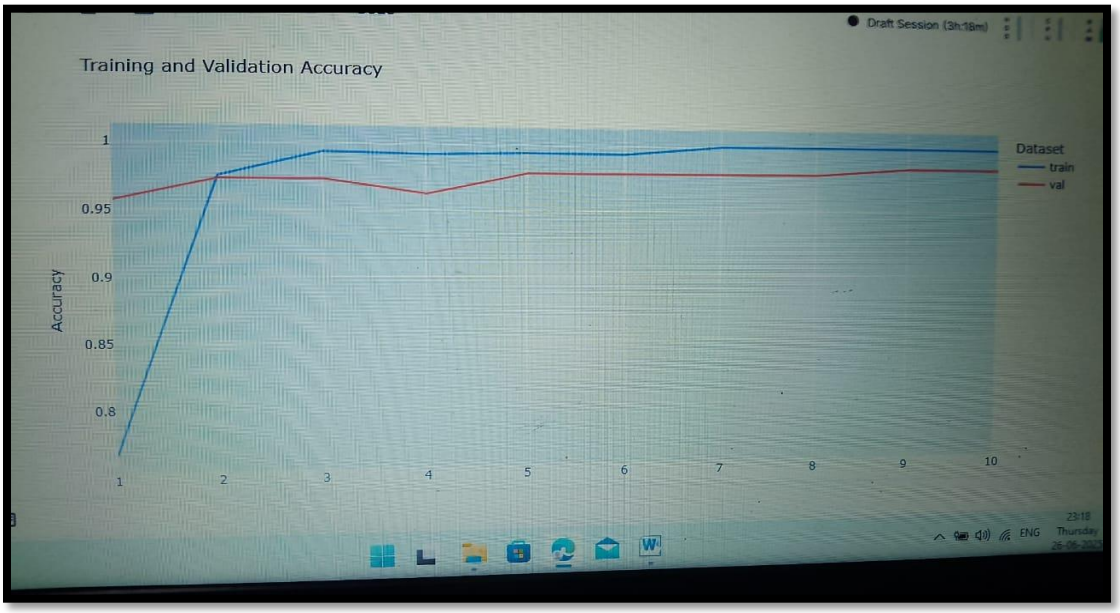
Security Measures

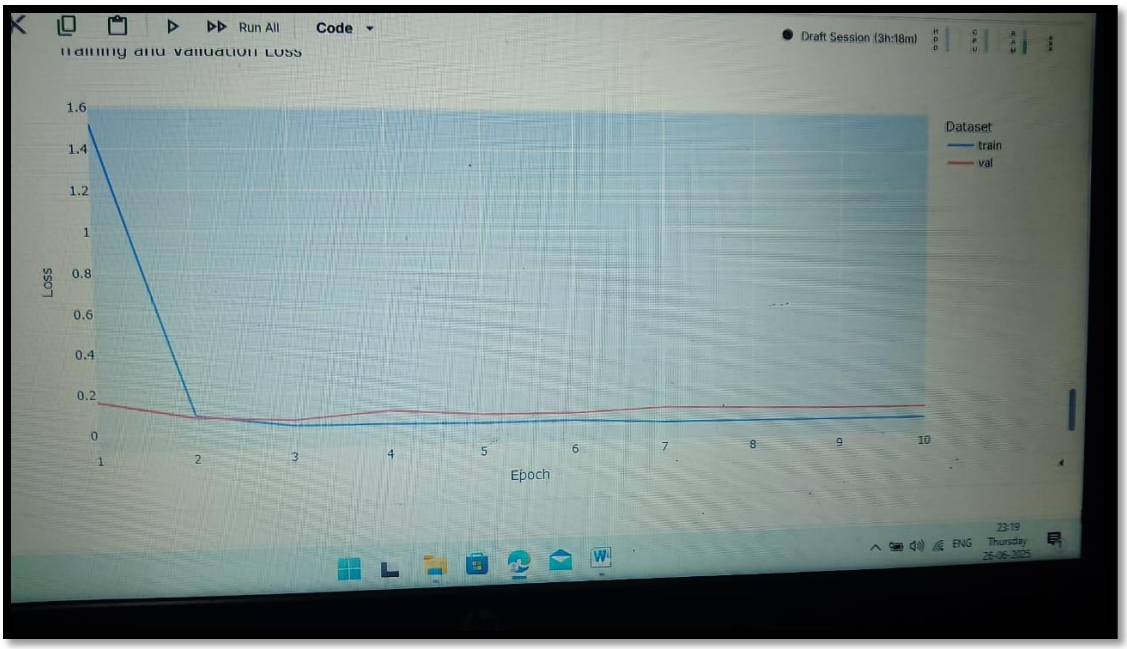
- Passwords are hashed (not stored in plain text)
- HTTPS enforced for production deployment
- Tokens have expiry (e.g., 1 hour)
- CSRF protection (Flask-WTF if using forms)
- Admin APIs are protected with role checks

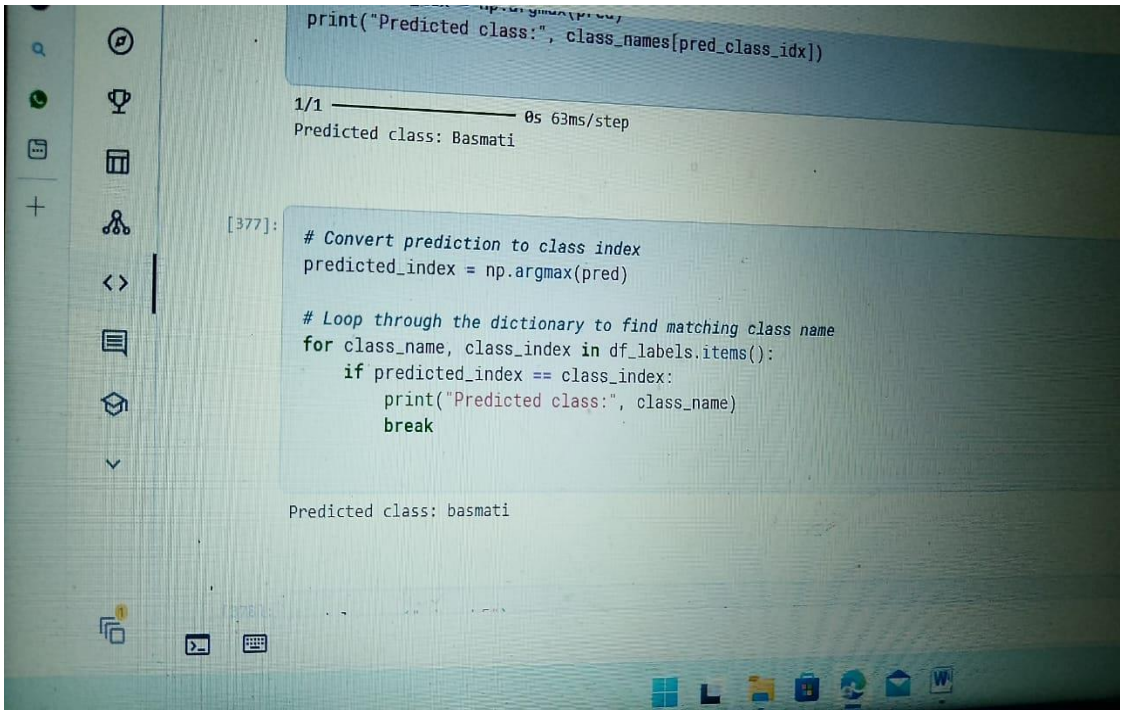
8. User Interface

The user interface of Grain Palette is designed to offer a smooth, user-friendly experience for both general users and administrators. Built with [*Streamlet/Flask/React* – depending on your stack], the UI focuses on simplicity, clarity, and functionality.






Run All
Code
Draft Session (3h:18m)
Training and Validation Loss
Dataset
train
val
Loss
Epoch
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
9
10
ENG
23:19
Thursday
26-06-2025


print("Predicted class:", class_names[pred_class_idx])
1/1 ——————— 0s 63ms/step
Predicted class: Basmati
[377]:
# Convert prediction to class index
predicted_index = np.argmax(pred)
# Loop through the dictionary to find matching class name
for class_name, class_index in df_labels.items():
if predicted_index == class_index:
print("Predicted class:", class_name)
break
Predicted class: basmati

9. Testing

The testing strategy for *Grain Palette* was designed to ensure the robustness, accuracy, and generalizability of the rice classification model. We used a combination of **unit testing**, **validation testing**, and **performance evaluation** techniques to assess both the frontend and backend components of the system, along with the trained deep learning model.

For the deep learning model, the dataset was split into **training (70%)**, **validation (15%)**, and **testing (15%)** sets. During model training, validation data was used to monitor over fitting, while the test dataset—completely unseen during training—was used to evaluate the final model performance. Metrics like **accuracy**, **precision**, **recall**, **F1-score**, and **confusion matrix** were used to interpret model effectiveness across different rice types.

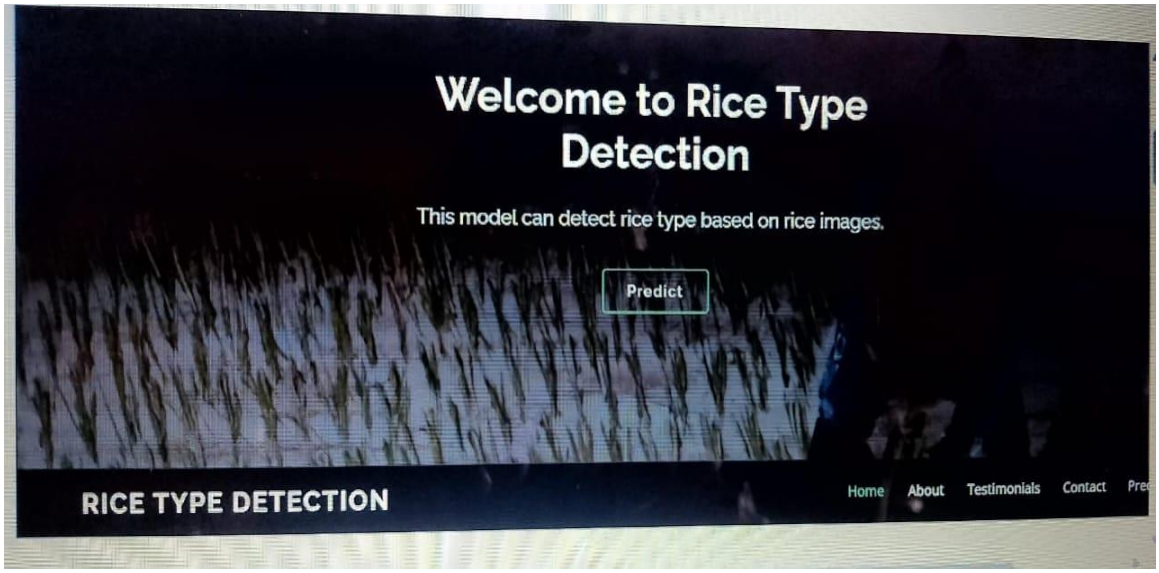
Tools Used:

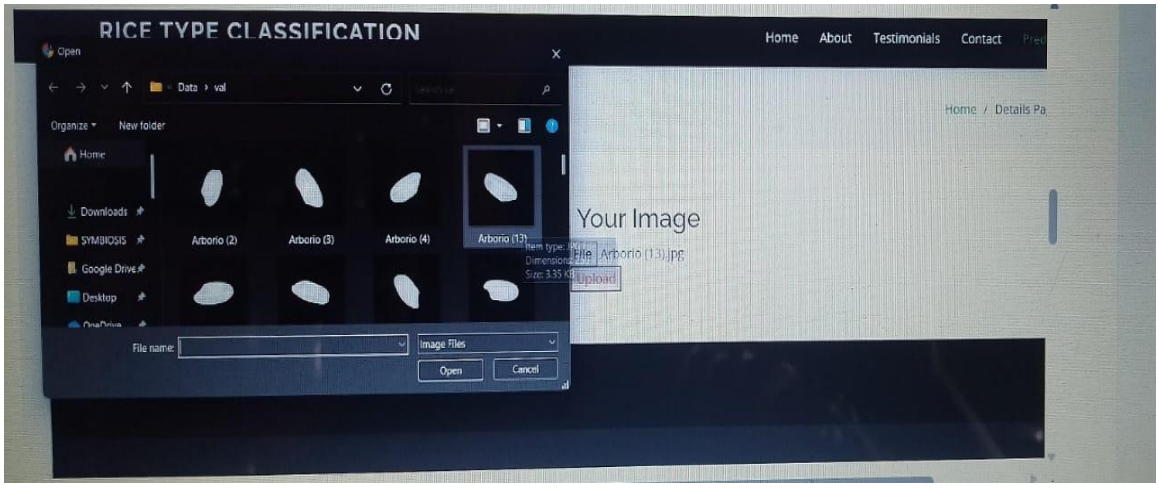
- Jest for frontend unit tests
- Mocha + Chai for backend testing
- Postman for manual API testing

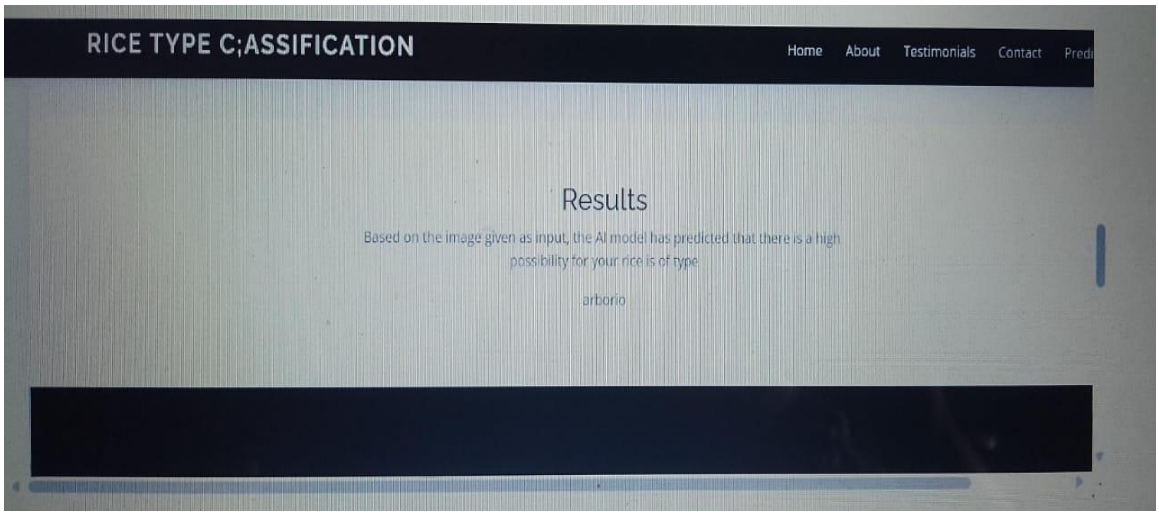
Strategy:

- Unit tests for components and routes
- Integration tests for API endpoints
- End-to-end testing of workflow from upload to prediction

10.Screenshots or Demo






RICE TYPE C;ASSIFICATION
Home
About
Testimonials
Contact
Results
Based on the image given as input, the AI model has predicted that there is a high possibility for your rice is of type
arborio

11. Known Issues

While *Grain Palette* aims to provide accurate rice type classification through transfer learning, the following known issues and limitations have been identified:

- Model loading time may be slow on first request due to cold start.
- Limited to rice types present in training dataset; fails on unseen categories.
- No drag-drop image support yet.

- **Limited Dataset Diversity:**
  The model was trained on a dataset with limited rice varieties and image conditions. This may reduce accuracy when users upload rice

- **Image Quality Sensitivity:**
  Low-resolution or blurry images can cause misclassification, as the model relies on clear grain features.

- **Model Loading Time:**
  The initial load time of the deep learning model on the backend can be slow, especially on servers with limited computational resources.

- **File Upload Restrictions:**
  Currently, only .jpg, .jpeg, and .png file formats are supported. Uploading unsupported formats results in an error.

- **Scalability Constraints:**
  The backend server is a single-instance deployment and may experience delays under heavy concurrent requests.

- **Error Handling Gaps:**
  Some edge cases, such as corrupted image files or network interruptions during upload, may not be gracefully handled and could cause the app to crash or behave unexpectedly.

- **Browser Compatibility:**
  The frontend has been primarily tested on modern browsers (Chrome, Firefox). Older browsers may experience layout or functionality issues.

- **No Offline Support:**
  The application requires an internet connection to communicate with the backend and perform predictions.

12. Future Enhancements

To further improve the performance, user experience, and scalability of the *Grain Palette* project, the following future enhancements are proposed:

1. Expand Dataset with More Rice Varieties

- Incorporate additional rice types from diverse geographical regions.
- Include images taken under different lighting, angles, and backgrounds to improve model generalization.

2. Model Optimization for Speed and Size

- Use techniques such as **model quantization** or **Tensor Flow Lite conversion** to reduce model size and loading time.
- Deploy models via GPU/TPU for faster real-time predictions.

3. Incorporate Explainable AI (XAI)

- Add visual explanations (e.g., Grad-CAM) to show which parts of the image influenced the prediction.
- This would help users understand the model's decision-making process.

4. Bulk Image Upload & Batch Prediction

- Allow users to upload multiple images at once and receive a consolidated report.
- Useful for farmers, researchers, and mill operators analyzing large batches.

5. Multilingual Support

- Add language support (e.g., Hindi, Telugu, Bengali) to improve accessibility for non-English speaking users.

6. Result Analytics Dashboard

- Provide users with downloadable reports and graphical summaries of classification results.

- Include metrics like classification confidence, usage history, etc.

7. Mobile App Integration

- Develop a mobile application (Android/iOS) to allow rice classification on the go using smartphone cameras.

8. Cloud Deployment and Scalability

- Deploy the system using scalable services like **AWS**, **Google Cloud**, or **Azure**.
- Implement load balancing and auto-scaling for high user traffic.

9. Searchable Rice Database

- Build a feature where users can compare classified rice types with a curated database that includes characteristics like grain length, aroma, and market price.

10. Enhanced Error Handling and Logging

- Improve backend error handling for corrupted files, failed uploads, or prediction errors.
- Add logging and monitoring for debugging and maintaining service uptime.

These future enhancements aim to make *Grain Palette* more powerful, scalable, and accessible while continuing to serve researchers, farmers, and agricultural industries with intelligent grain classification tools.