Steelcase AI-Powered Color Validator

JIC Team 4307

Zuhair Al Araf, Rishi Manimaran, Benson Lin, Zhihui Chen, Ethan Kim

Client: Edward Vanderbilt

Repository: https://github.com/JDA-4307/JIC_4307_ColorValidation.git

Table of Contents

Table of Contents 2
Table of Figures 4
Terminology 5
Introduction 6
Background 6
Document Summary 6
System Architecture 7
Introduction 7
Static System Diagram 9
Figure 1: Static System Diagram 9
Dynamic System Diagram 10
Figure 2: Dynamic System Diagram to demonstrate validation and report retrieval workflows. 10
Workflow 1: Running a New Validation 11
Workflow 2: Viewing Saved Reports 11
Key Interactions 11
Component Design 12
Introduction 12
Static Element 13
Figure 3: Static Element Diagram for Component Design 13
Dynamic Element 16
Figure 4: Dynamic Element Diagram for Component Design 16
Data Design 18
Overview 18
Introduction 18
Entity Relationship Diagram 19
Figure 5: Entity Relationship Diagram. 19
File Use 20
Data Exchange 20
User Interface 20
Introduction 20
Home Screen 21
Home Screen Heuristic Analysis (Figure 6) 21
Figure 6: User Interface: Home Screen 22
Image Selection Screen 22
Image Selection Screen Heuristic Analysis (Figure 7) 22
Figure 7: User Interface: Image Picker Screen 23
Validation Screen 23

Figure 8: User Interface: Validation Screen 24

Figure 9: User Interface: Validation Results: Analyze 25

Figure 10: User Interface: Validation Results: After Analyze 25

Figure 11: User Interface: Validation Results: Full Report 26

Validation Screen Heuristic Analysis (Figures 8, 9, 10, 11) 26

Reports Screen 27

Figure 11: User Interface: Reports Screen 27

Report Screen Heuristic Analysis (Figure 12) 27

Appendix **29**

Members 29

Table of Figures

Figure 1: Static System Diagram 9
Figure 2: Dynamic System Diagram to demonstrate validation and report retrieval workflows. 10
Figure 3: Static Element Diagram for Component Design 13
Figure 4: Dynamic Element Diagram for Component Design 16
Figure 5: Entity Relationship Diagram. 19
Figure 6: User Interface: Home Screen 22
Figure 7: User Interface: Image Picker Screen 23
Figure 8: User Interface: Validation Screen 24
Figure 9: User Interface: Validation Results: Analyze 25
Figure 10: User Interface: Validation Results: After Analyze 25
Figure 11: User Interface: Validation Results: Full Report 26
Figure 11: User Interface: Reports Screen 27

Terminology

API (Application Programming Interface)

- Definition: A set of protocols and tools for building and interacting with software applications.
- Context: APIs in the backend (Python Flask) facilitate communication between the user interface, AI model, and Firestore.

Firestore

- Definition: A NoSQL cloud database for storing and syncing data.
- Context: Used for storing validation results and metadata in the Color Validation Application.

HTTPS (Hypertext Transfer Protocol Secure)

- Definition: A secure version of HTTP that encrypts data exchanged between a client and a server.
- Context: Ensures secure communication between the client and server in the system.

Python Flask

- Definition: A Python runtime environment for building server-side applications.
- Context: Used for the backend development of the Color Validation Application.

Metadata

- Definition: Data that provides information about other data, such as timestamps and validation results.
- Context: Stored in Firestore to facilitate efficient retrieval of reports.

React Native

- Definition: A cross-platform framework for building mobile applications using JavaScript and React.
- Context: Used for the front-end development of the Color Validation Application.

TensorFlow

- Definition: An open-source library developed by Google for numerical computation and large-scale machine learning.
- Context: Used to develop the custom AI model hosted locally for color validation.

.

Introduction

Background

Steelcase is a global furniture company committed to designing and manufacturing innovative furnishings and solutions to help people do their best work in many places, especially offices. One of the challenges the company is facing is validating the color quality of their wood veneer products. Ensuring that each piece of finished wood furniture matches the customer's exact specifications is a significant challenge for Steelcase. The current color validation process is time-consuming and prone to human error, leading to inconsistencies that can affect customer satisfaction and brand reputation. We are developing an application to assist Steelcase's quality assurance and field engineers in reducing visual bias and minimizing customer complaints related to mismatched wood veneer colors. The solution combines Artificial Intelligence with a simple, user-friendly interface. Built with React Native, Python, and Firestore, the application streamlines the inspection process while improving accuracy and consistency.

Document Summary

The System Architecture section outlines the high-level structure of the Color Validation Application, including its modularity, scalability, and efficiency. It is composed of the four major components: User Interface, Backend, Processing, and Data Storage and their roles and purpose in detail. It explains the reasons behind the chosen respective solutions for each component. The section also details how the combination of these components will help develop a robust optimal solution for wood veneer color validation.

The Component Design section provides a detailed breakdown of each of the following system components and its functionality. The main components that compose the frontend, backend, and data storage are described. The reasoning behind why each component was chosen and how it is structured is thoroughly explained.

The Data Storage Design explains how data is stored, retrieved and managed within the application.The content focuses on describing and explaining why Firestore is our primary data storage option. It goes on to describe how metadata (e.g. validation results and timestamps) is stored and retrieved. It also details how we minimize storage costs by taking advantage of Firestore's schema-less design, scalability, and its integration with Python Flask.

The UI Design section details the different components of the user interface. It discusses the key pages that compose the app. It also details how users will be able to seamlessly navigate across the pages. Additionally, the extensive testing done to fool proof the app is examined.

System Architecture

Introduction

The Color Validation Application's architecture is designed for modularity, scalability, and efficiency in validating veneer colors against Steelcase specifications and retrieving historical reports. Our goal is to create a robust, user-friendly solution that utilizes advanced AI for real-time color validation while ensuring seamless data persistence. By breaking the system into independent components—React Native for the front end, Python Flask for backend orchestration, and Firestore for data storage—we aim to deliver a reliable and performant system for concurrent users.

This architecture employs a layered design that separates concerns across the user interface, business logic, processing, and storage, allowing for future enhancements without disrupting system flow. It supports multiple workflows, enabling users to run new validations and access saved reports efficiently and securely.

Rationale

The architectural design balances efficiency, modularity, and user experience, with each component selected for its strengths:

- React Native (User Interface): Chosen for its cross-platform capabilities, allowing accessibility on both iOS and Android without separate development efforts. It handles image uploads, displays validation results, and retrieves saved reports, with a modular design for easy UI updates.
- Python Flask (Backend): Selected for its lightweight, event-driven architecture, ideal for managing concurrent requests. Flask orchestrates communication between components, passing images into our TFLite Model for validation, saving results to Firestore, and fetching user reports, ensuring clear separation of concerns.

- Model Processing: Locally hosted TFLite color validation model. The system analyzes images against Steelcase's specifications, providing results with confidence scores for transparency.
- Firestore (Data Storage): Efficiently stores structured metadata, such as validation results and timestamps, while avoiding the overhead of raw image storage. Its integration with Python Flask allows for seamless data retrieval for reports.

Security Considerations

While this version does not include security measures like authentication or encryption, the architecture is designed for future enhancements. Firestore can integrate with Firebase Authentication for secure, role-based access, and HTTPS ensures secure client-server communication.

Layered Architecture

The layered design enhances maintainability and scalability by separating the system into distinct components. This modularity allows for the integration of new AI models or storage options without affecting the front-end or back-end logic.

Future-Proofing

The architecture supports extensibility, with our backend processing system enabling scalability for increased user traffic and Firestore's schema-less design accommodating future data requirements. This architecture meets project objectives while allowing for growth and enhancements.

Static System Diagram

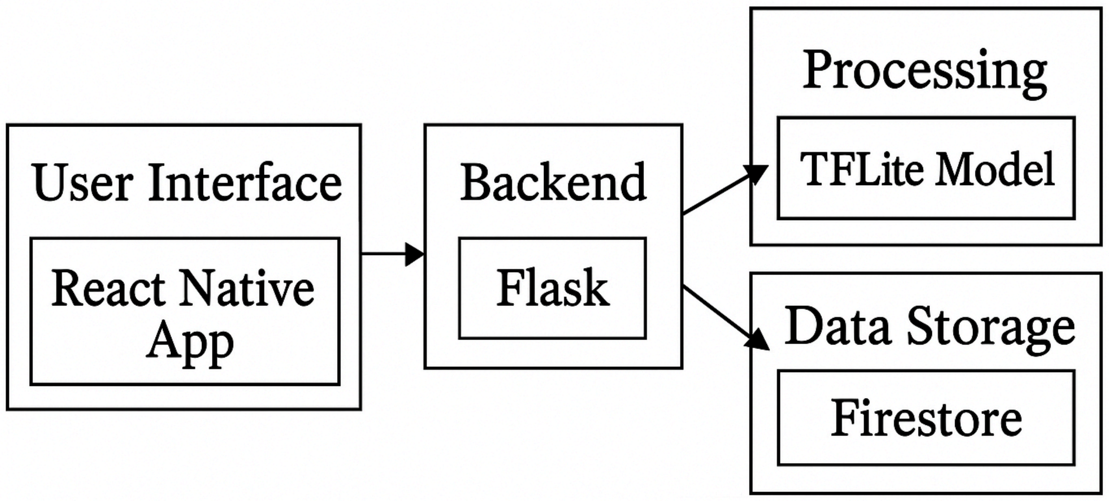


Figure 1: Static System Diagram

Our architecture comprises four major components: User Interface, Backend, Processing, and Data Storage. Each component is critical to enabling the functionality of the color validator. The first component is the interface the user interacts with. The interface will be built with React Native. This allows us to make a mobile app that is compatible with both iOS and Android devices. Typescript will be utilized to create the frontend features necessary to accomplish the stated functionality of the app. A few of these features will be taking scans, uploading scans, and viewing previous reports. The front end will communicate with the second major component, the back end. The backend will be built with Python Flask. It will host several APIs that the frontend and backend will utilize to communicate. One major API will be sending images to the backend and receiving the model prediction response. The backend will utilize a custom AI model processing system locally hosted which is the third major component of our architecture. Our system will utilize tensorflow lite to compile a model from a dataset of various wood veneers and predict the inputted veneer's color. During development, we will build an app that enables both local and cloud-based processing and we'll evaluate which of the two has better accuracy and response time for our final app. The winner will determine how this component will be structured. Our last component will be the Data Storage. We will utilize Firestore to save wood veneer reports. The backend will retrieve previous reports at their user's request as well as add new reports to the database.

Dynamic System Diagram

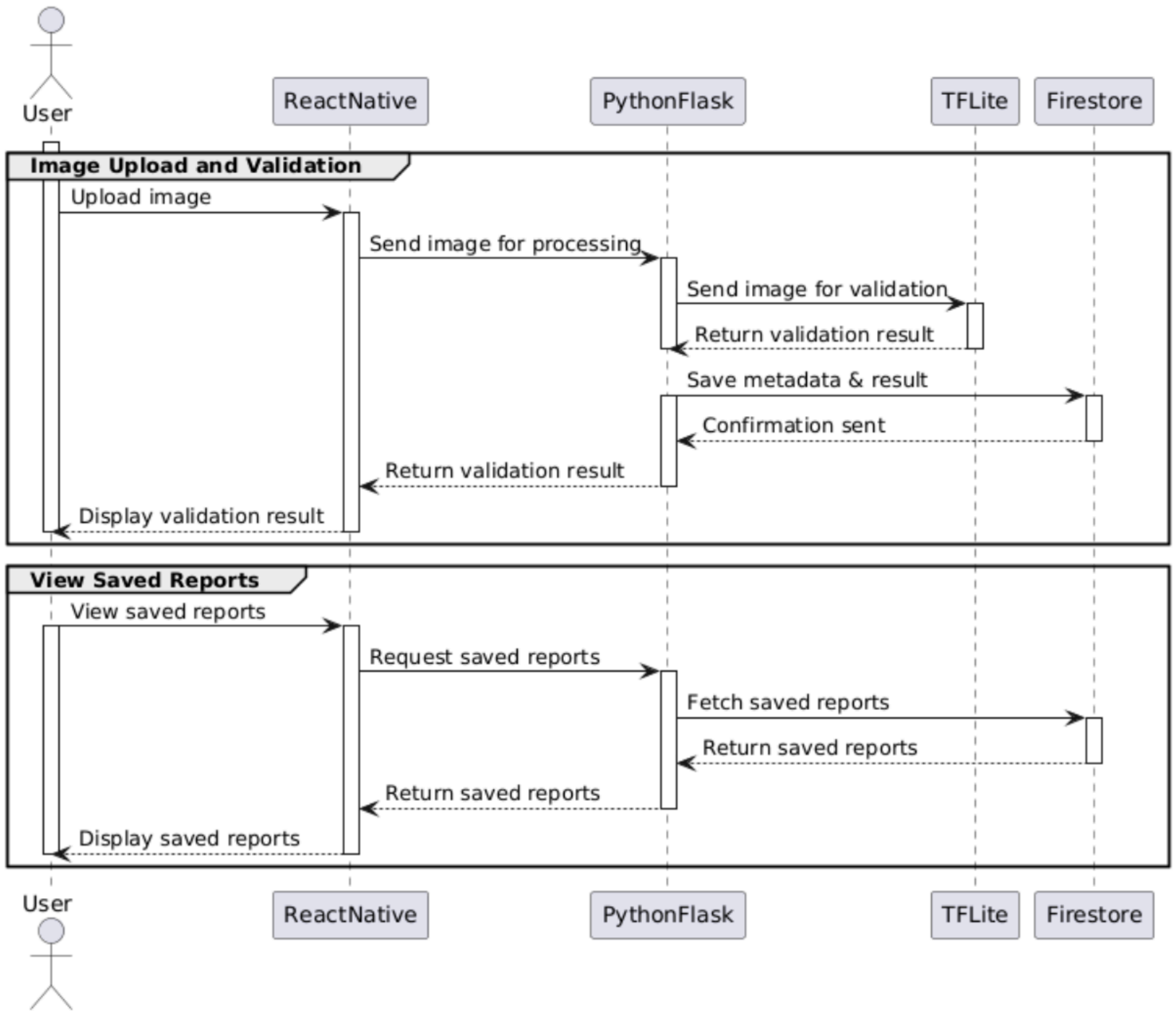


Figure 2: Dynamic System Diagram to demonstrate validation and report retrieval workflows.

The dynamic system diagram illustrates the runtime behavior of the application, focusing on two core workflows: running a new validation and viewing saved reports. This diagram was chosen to demonstrate the sequential interactions between system components—React Native, Python Flask, and Firestore—during these real-time processes. These workflows were selected because they represent the application's most critical functionalities from the user’s perspective.

Workflow 1: Running a New Validation

Upon initiating the validation process, the user interacts with the ReactNative front-end by uploading an image of the veneer. This triggers a request from ReactNative to the python flask back-end, which serves as the central coordinator. The back-end sends the image to TFLite, where the color validation model processes the input to determine if the veneer matches the specified Steelcase color.

Once processing is complete, TFLite returns the validation results to a flask endpoint, including the verdict (pass/fail) and the model's confidence level. These results are stored as metadata in Firestore, ensuring the information is saved for future access. After successfully storing the data, flask returns the results to the ReactNative front-end, which then displays the verdict to the user, completing the workflow.

Workflow 2: Viewing Saved Reports

In the second workflow, the user requests to view saved validation reports through the ReactNative front-end. This prompts a request to the Python Flask backend, which fetches the corresponding data from Firestore. Once the saved reports are retrieved, the Flask backend sends the data back to the ReactNative front-end, where it is displayed to the user in an accessible format.

Key Interactions

- ReactNative serves as the user-facing interface, facilitating input collection and output display.
- Python Flask orchestrates interactions, acting as a mediator between the front-end, TFLite, and Firestore.
- TFLite Model processing system performs the core image validation, leveraging a machine learning model for accurate analysis.
- Firestore provides persistent storage for metadata and validation results, enabling efficient retrieval for report viewing.

Component Design

Introduction

This document provides a detailed breakdown of the Component Design for the Steelcase AI-Powered Color Validator. It builds upon the System Architecture section by zooming into the low-level components of the system, their static relationships, and their dynamic interactions during runtime. The purpose is to reinforce the conceptual integrity of the system architecture while providing a deeper understanding of how the system operates at the component level.

The document is divided into two main sections: Static Elements: A structural diagram (class or component diagram) showing low-level components, their attributes, methods, and relationships. Dynamic Elements: A runtime interaction diagram (sequence or interaction overview diagram) illustrating how the static components interact during key workflows. Both sections maintain conceptual integrity with the system architecture and provide a clear correlation between high-level and low-level design decisions.

Static Element

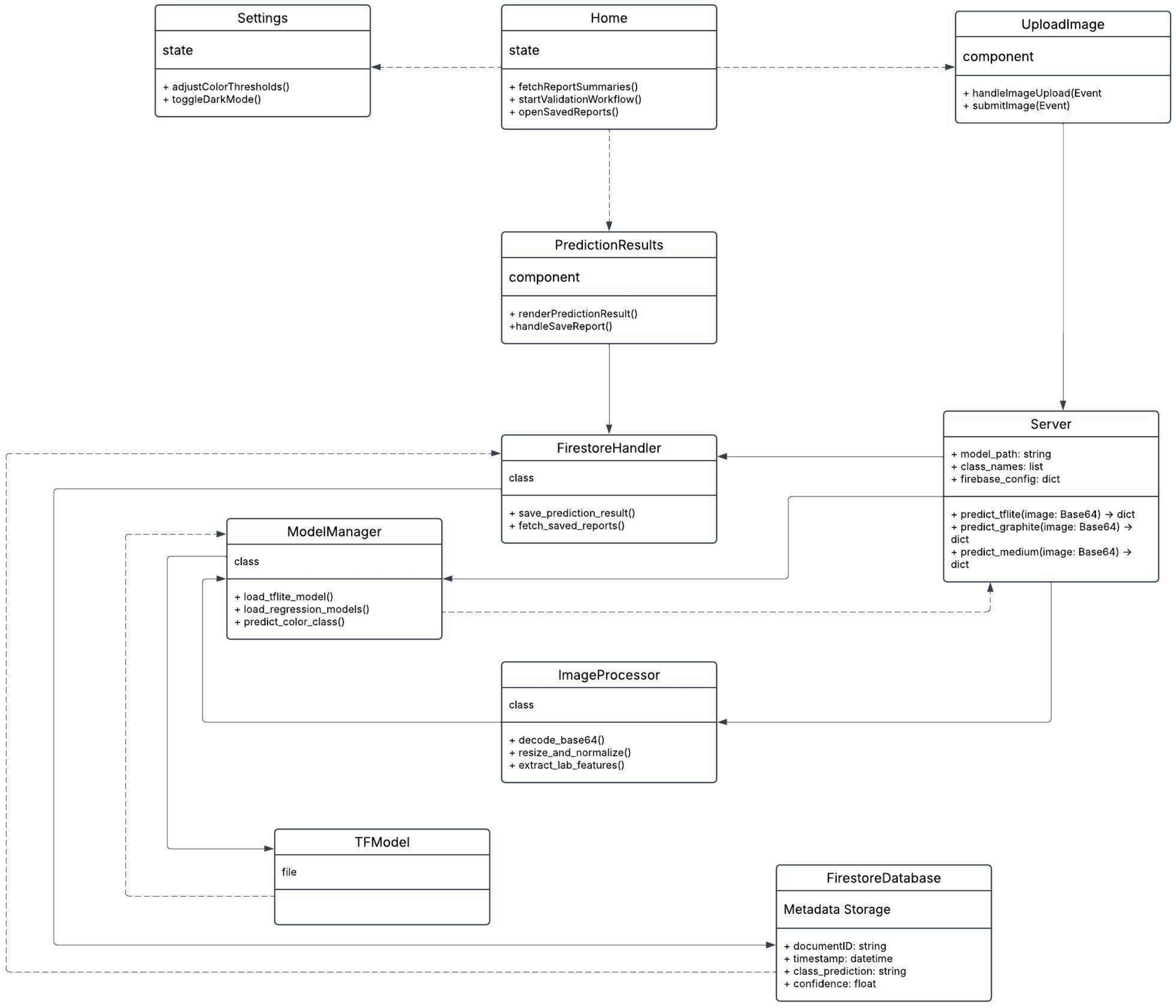


Figure 3: Static Element Diagram for Component Design

The static component design diagram shows the architecture of the color verification application by detailing the components involved in the application's functionality and their interactions. The diagram is divided into three primary layers: the web view, the application logic, and the data storage layer.

The web view layer contains four components: Home, Settings, UploadImage, and PredictionResults. The Home component represents the main user interface for initiating validation workflows, fetching report summaries, and accessing saved reports. It is connected to the UploadImage component via a dashed arrow, indicating an indirect relationship where the home interface can trigger the image upload process. The Home component also has a dashed arrow pointing toward the PredictionResults component, suggesting a connection for navigating to the results display. The Settings component has two methods, adjustColorThresholds() and toggleDarkMode(), which allow users to modify color validation parameters and switch between light and dark modes. The UploadImage component contains methods handleImageUpload(Event) and submitImage(Event), which manage the selection and submission of images for analysis. This component has a solid arrow directed toward the Server component, indicating a direct interaction where the image data is transmitted for processing. The PredictionResults component includes methods renderPredictionResult() and handleSaveReport() and has a solid arrow pointing to the FirestoreHandler, representing the action of saving the validation results to the database.

The application logic layer comprises four core components: Server, ImageProcessor, ModelManager, and FirestoreHandler. The Server component is central to this layer, with attributes like model_path, class_names, and firebase_config. It provides three primary methods: predict_tflite(image: Base64), predict_graphite(image: Base64), and predict_medium(image: Base64). From the server, there is a solid arrow leading to the ImageProcessor, indicating the delegation of tasks related to decoding base64-encoded images, resizing them, and extracting LAB color features through methods like decode_base64(), resize_and_normalize(), and extract_lab_features(). Another solid arrow extends from the Server to the ModelManager, showing the invocation of the color prediction functionality. The ModelManager class includes the methods load_tflite_model(), load_regression_models(), and predict_color_class(). A solid arrow runs from the ModelManager to the TFModel component, illustrating the interaction with the trained TensorFlow Lite model for inference. Additionally, the Server connects to the FirestoreHandler through a solid arrow, indicating the process of saving prediction results. The Firestore Handler provides the methods save_prediction_result() and fetch_saved_reports() and serves as the intermediary between the application logic and the data storage layer.

The data storage layer includes two components: TFModel and FirestoreDatabase. The TFModel is a file that stores the trained machine learning model used for wood veneer color classification. It is connected to the ModelManager with a solid arrow, representing the process of loading the model weights during inference. The Firestore Database component is responsible for persisting prediction metadata and includes attributes like documentID, timestamp, class_prediction, and confidence. A solid arrow links the Firestore Handler to the FirestoreDatabase, indicating the storage of report metadata.

The arrows in the diagram follow clear patterns to represent interactions between components. Solid arrows denote direct method calls or data flows, such as between UploadImage and Server or Server and ModelManager. Dashed arrows, such as those between Home and PredictionResults, indicate indirect relationships or event-based triggers.

Dynamic Element

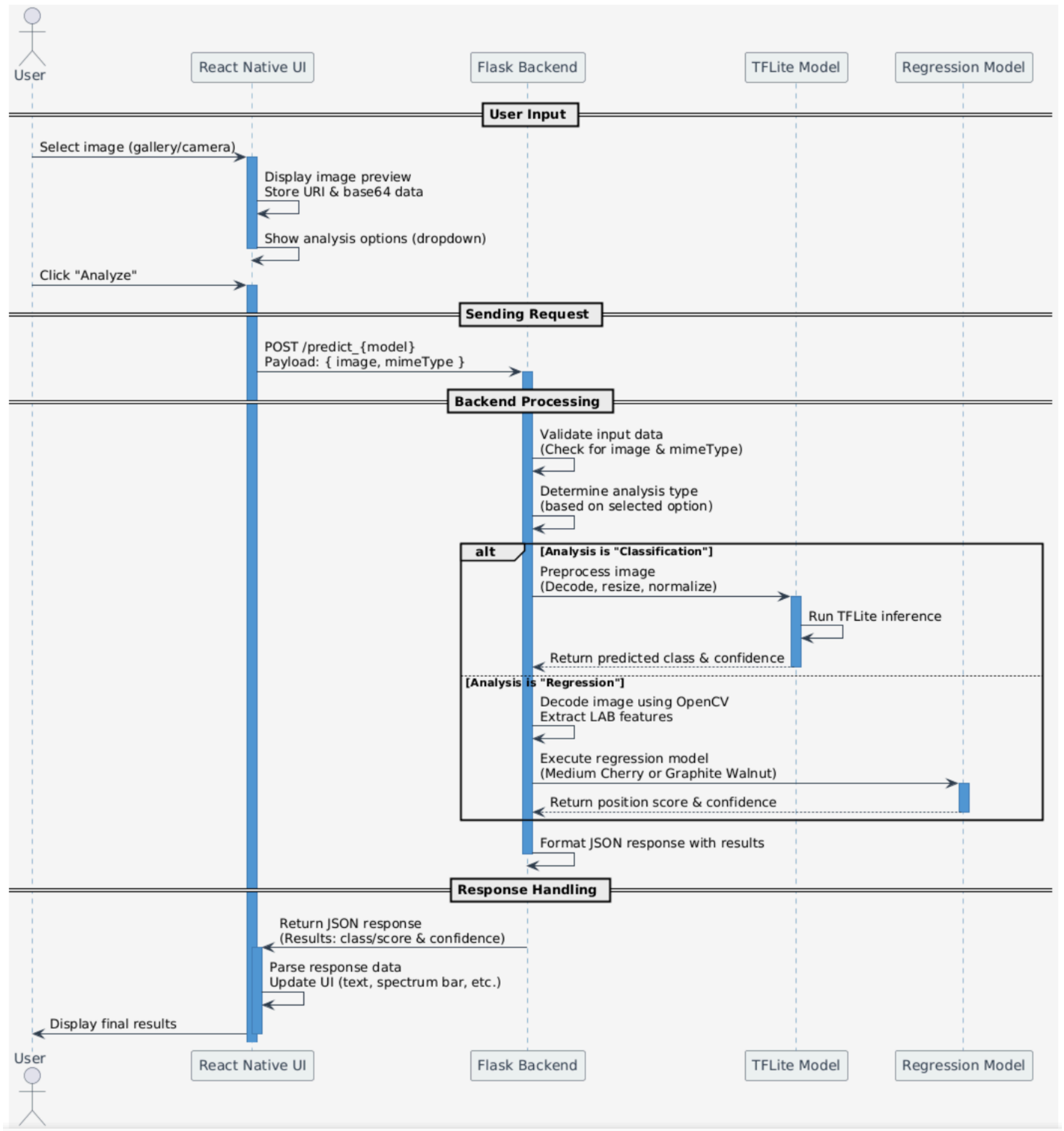

User
React Native UI
Flask Backend
TFLite Model
Regression Model
User Input
Select image (gallery/camera)
Display image preview
Store URI & base64 data
Show analysis options (dropdown)
Click "Analyze"
Sending Request
POST /predict_{model}
Payload: { image, mimeType }
Backend Processing
Validate input data
(Check for image & mimeType)
Determine analysis type
(based on selected option)
alt
[Analysis is "Classification"]
Preprocess image
(Decode, resize, normalize)
Run TFLite inference
Return predicted class & confidence
[Analysis is "Regression"]
Decode image using OpenCV
Extract LAB features
Execute regression model
(Medium Cherry or Graphite Walnut)
Return position score & confidence
Format JSON response with results
Response Handling
Return JSON response
(Results: class/score & confidence)
Parse response data
Update UI (text, spectrum bar, etc.)
Display final results

Figure 4: Dynamic Element Diagram for Component Design

The sequence diagram represents the overview of the process the application undertakes when the user attempts to analyze an image. First, the user is prompted to select an image. This can be done either by selecting an image from the user's gallery or by taking a photo. The user uses the native photo app on their device, enabling them to edit or crop the image if necessary. Once the user uploads the image, a preview is displayed on the screen. After that, the user can hit the analyze button to send the image to the prediction process. The image is sent through a POST request to the Flask backend, with the payload being a sixty four byte encoded version of the image. The backend decodes the image and runs it through the classification process. This process involves sending the preprocessed image to the tfLite model script. The prediction is then utilized by the backend to run the respective regression model script. The model returns a positional score on a spectrum from -1 to 1 on lightness. It also returns a confidence rating. These two values are formatted in a JSON response and sent back in the same POST request. The results are then displayed through the React Native UI page. This streamlined process enables the user to get accurate predictions on their submitted images.

Data Design

Overview

This section outlines the data storage, file formats, and data exchange methods within the Steelcase AI-Powered Color Validator application. The system employs Firestore as a NoSQL cloud database, TensorFlow models hosted locally, and secure data transfer protocols to ensure efficient and secure operation. The goal of this section is to provide external developers with sufficient understanding of how the data is stored, exchanged, and protected across the system.

Introduction

This section provides an overview of how data is managed within the Color Validation Application. The system is designed to validate veneer colors against Steelcase specifications using an AI model and store metadata such as validation results and timestamps in Firestore. The data design is intended to support the modular and scalable nature of the application while ensuring secure and efficient handling of user interactions and data retrieval.

In Figure 3, we present an Entity Relationship (ER) diagram tailored to the requirements of our web application. This diagram illustrates the various entities, their attributes, and the relationships between them. The database utilized in this project is Firestore, a NoSQL cloud database, which supports efficient data storage and retrieval. Following the diagram, we discuss file usage, data exchange mechanisms, and data security considerations to ensure the integrity and protection of the system's data.

Entity Relationship Diagram

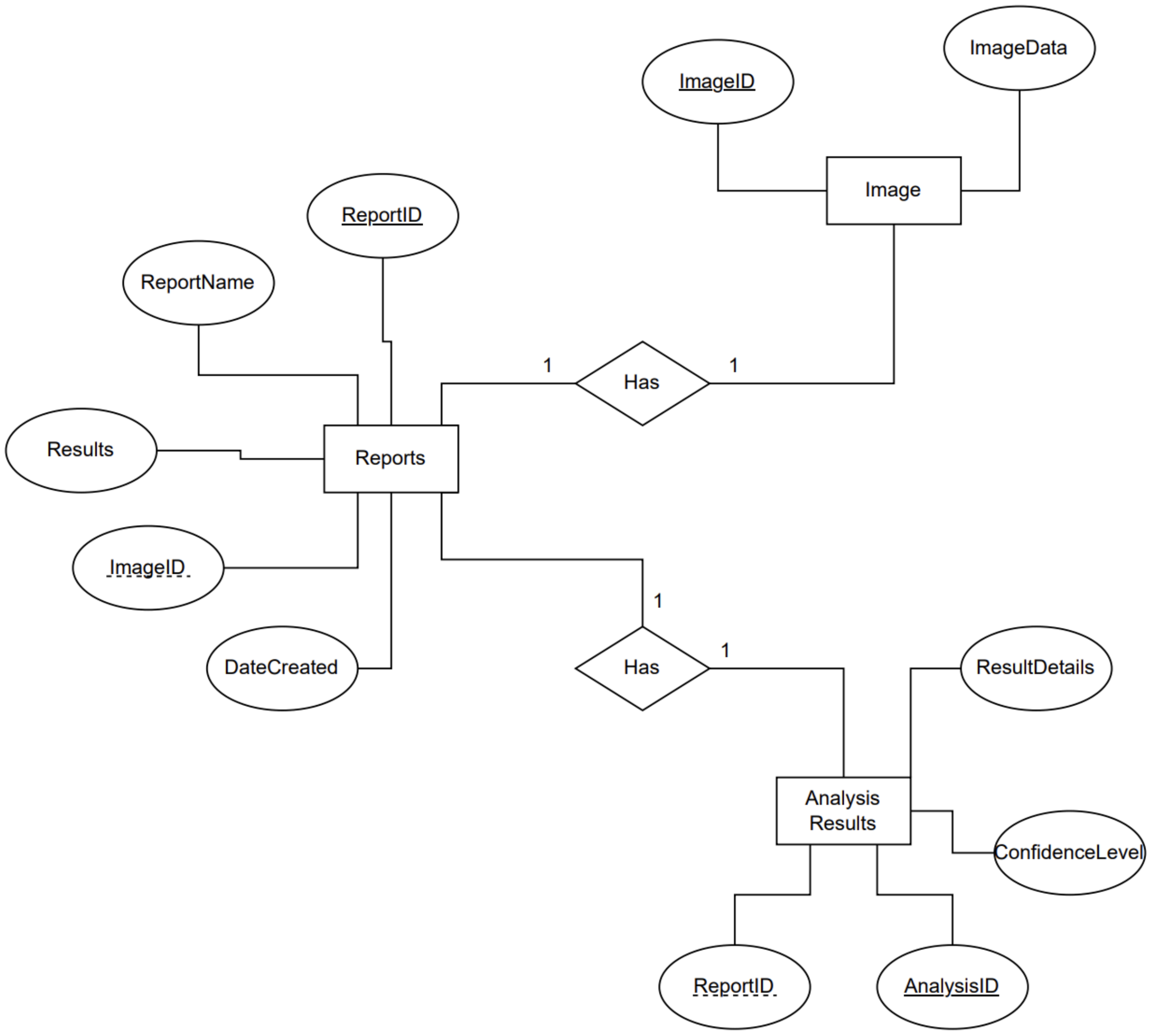


Figure 5: Entity Relationship Diagram.

List of Entities:

1. **Reports**: Stores information about individual reports, including the report name, results, associated image ID, and date of creation.
2. **Image**: Represents image data associated with reports, identified by an Image ID and containing image-related information. The image data should be a jpeg file.
3. **Analysis Results**: Contains detailed analysis results linked to a report, including specific result details, confidence levels, and a unique analysis ID.

File Use

The images used in the models are jpeg format, and the photos taken should also be jpeg. This ensures uniform file usage because the models are trained on jpegs and any other image type may cause errors. Other information such as results, identifiers, and timestamps are directly stored in the Firestore as primitives.

Data Exchange

Currently, there are no security measures taken within the app since it is a research project and does not need to reflect real world usability in terms of app functions. If necessary, Firebase authentication service can be used to validate login and sign ups. There should not be any sensitive information stored within the app, so users will not have to worry about their information being leaked. Likewise, there is no need for data encryption or secure communication. For future plans, we may have quality assurance engineers that have specific clients which a title or name may be given to the report, thus data exchanging and keeping the client's details kept private and they should be able to have personal logins with their own data stored. This would not only ease the QA engineers and field engineers with finding their own validations, but also help keep their client's information private.

User Interface

Introduction

The User Interface (UI) of the Steelcase AI-Powered Color Validator is designed to provide a simple and intuitive experience for quality assurance and field engineers. As the primary point of interaction, the UI allows users to upload images of wood veneer samples, initiate color validation and analysis, and access past validation reports with ease. Built using **React Native**, the interface ensures cross-platform compatibility, making it accessible on both iOS and Android devices without requiring separate development efforts.

When users open the application, they are greeted by the **home screen**, as shown in **Figure 6**, which prominently displays the Steelcase logo. From this screen, a navigation bar at the bottom provides quick access to two key features: **Reports** and **Image Picker**.

The Reports page allows users to review past validation reports, while the Image Picker page enables users to begin a new color validation process.

Home Screen

Home Screen Heuristic Analysis (Figure 6)

1. **Visibility of System Status**
   - Application: The home screen displays the Steelcase logo prominently and includes a navigation bar with clearly labeled buttons for "Reports" and "Image Picker." Users immediately understand where they are in the application and what actions they can take next, ensuring they don’t feel lost.

2. **Match Between System and the Real World**
   - Application: The navigation bar uses standard icons and terminology (e.g., "Reports" and "Image Picker") that align with real-world concepts of selecting images and reviewing past analyses. By using familiar terms and symbols, the app reduces cognitive load and makes navigation intuitive, especially for users unfamiliar with the system.

3. **Consistency and Standards**
   - Application: The bottom navigation follows common UI patterns found in mobile applications, ensuring a consistent experience across platforms. Users familiar with other apps can quickly learn how to navigate this one, leading to a smoother user experience.

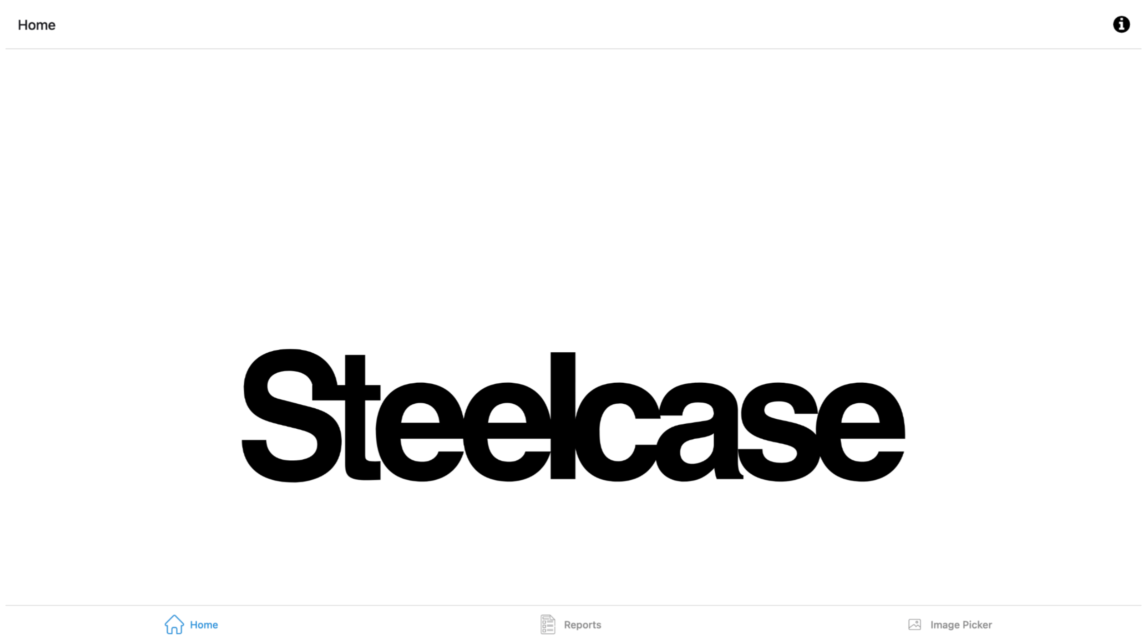


Figure 6: User Interface: Home Screen

Image Selection Screen

Upon selecting the **Image Picker** button, users are directed to the **Image Picker Screen** (Figure 7). Here, they are presented with two options:

- **Take a New Picture** – If the user chooses this option, the app will request camera access (if not previously granted). Once permission is provided, the device's camera opens, allowing the user to capture an image of the wood veneer sample.

- **Select an Existing Image** – This option opens the device's photo library, enabling the user to choose a previously taken image.

Image Selection Screen Heuristic Analysis (Figure 7)

1. **User Control and Freedom**
    - Application: The user can either take a new picture or select an existing image, and if they make a mistake, they can always return to this screen. Providing multiple options ensures users are not forced into one rigid workflow and can easily correct mistakes.

2. **Error Prevention**
    - Application: Before taking a new picture, the app asks for camera permissions to ensure the process runs smoothly without unexpected failures. By prompting for permissions in advance, the system prevents confusion and avoids user frustration caused by camera access errors.

3. **Recognition Rather Than Recall**
    - Application: When selecting an image from the library, the user is presented with a visual preview of their images rather than having to remember file names. By displaying thumbnails, users can easily recognize the image they want, reducing the need for memory recall.

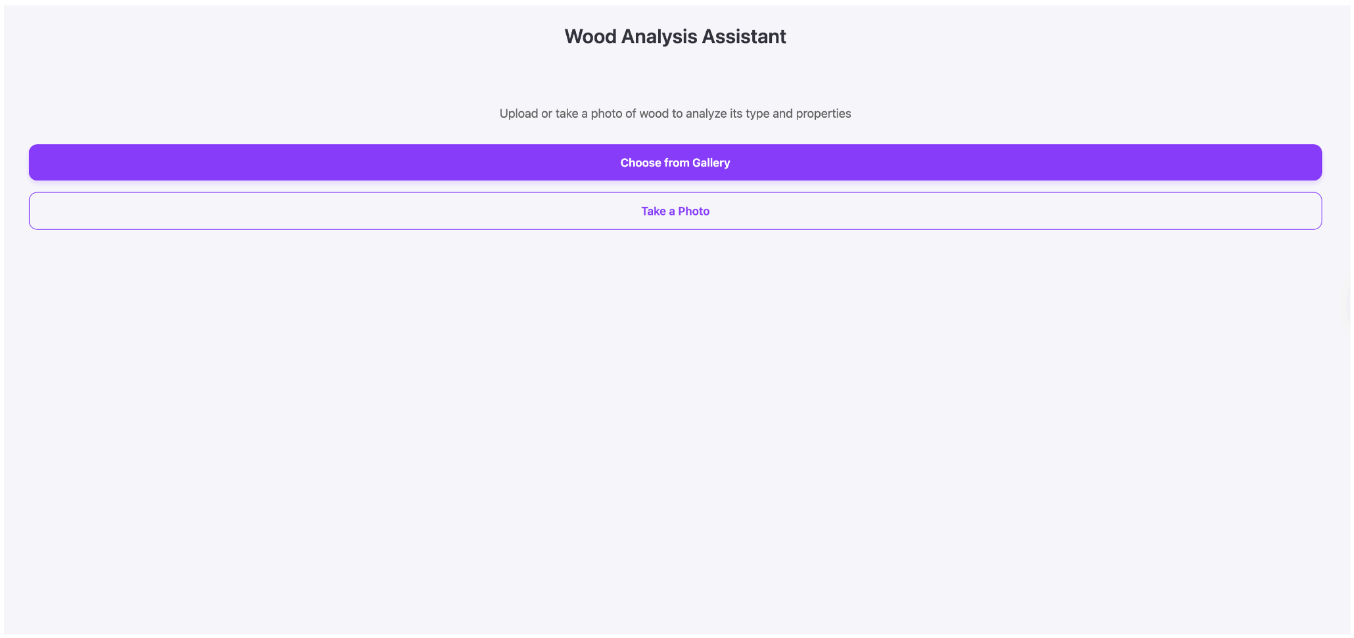

Wood Analysis Assistant
Upload or take a photo of wood to analyze its type and properties
Choose from Gallery
Take a Photo

Figure 7: User Interface: Image Picker Screen

Validation Screen

Once an image is selected or captured, the user is redirected to the **Validation Screen** (Figure 8), where they can preview the chosen image. If the selected image is incorrect or unsatisfactory, users can click the **New Image** button, which takes them back to the **Image Picker Screen (Figure 7**) to select a different image

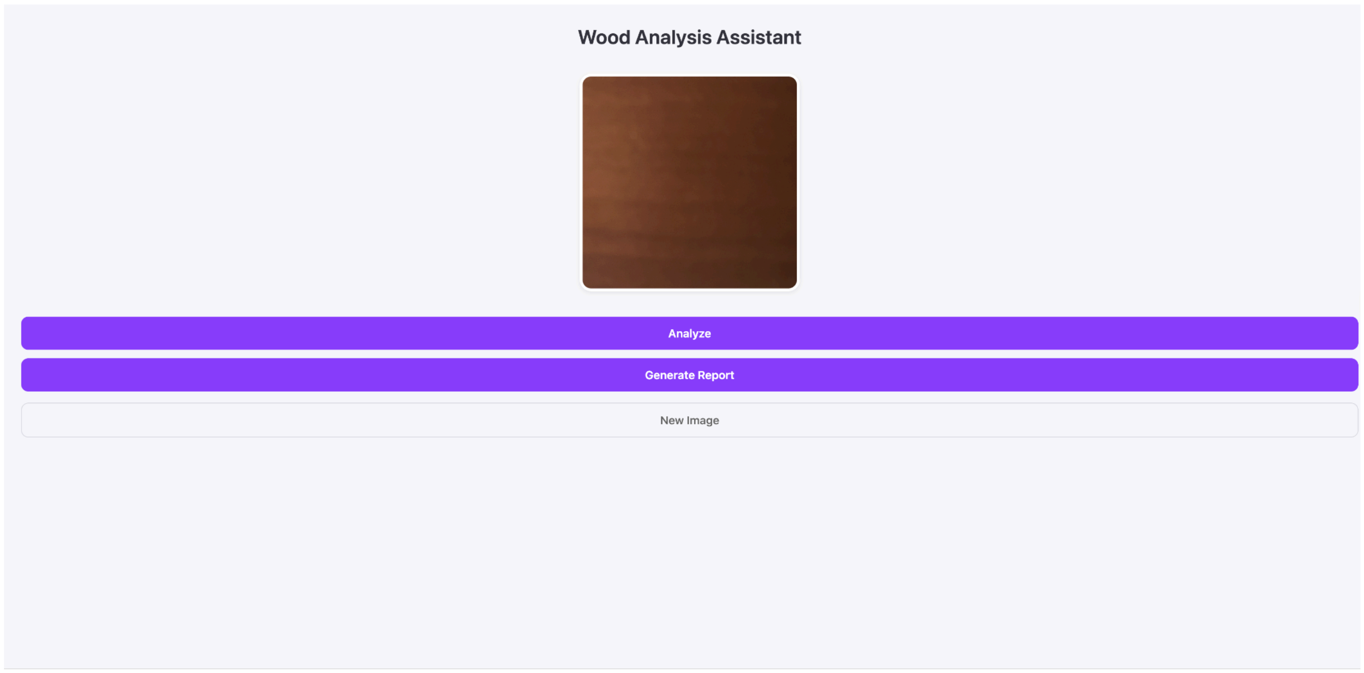


Figure 8: User Interface: Validation Screen

If the user proceeds with the selected image, they are presented with two validation options:

1. **Analyze Based on Classified Color** – The image is compared against a predefined veneer classification. The results would give the classified wood veneer type and the confidence level for that type. This is a quick way to analyze a wood veneer without saving the report. The result for the analyze button is demonstrated in **Figure 10** after you have selected one of the wood types for analysis shown in **Figure 9**.

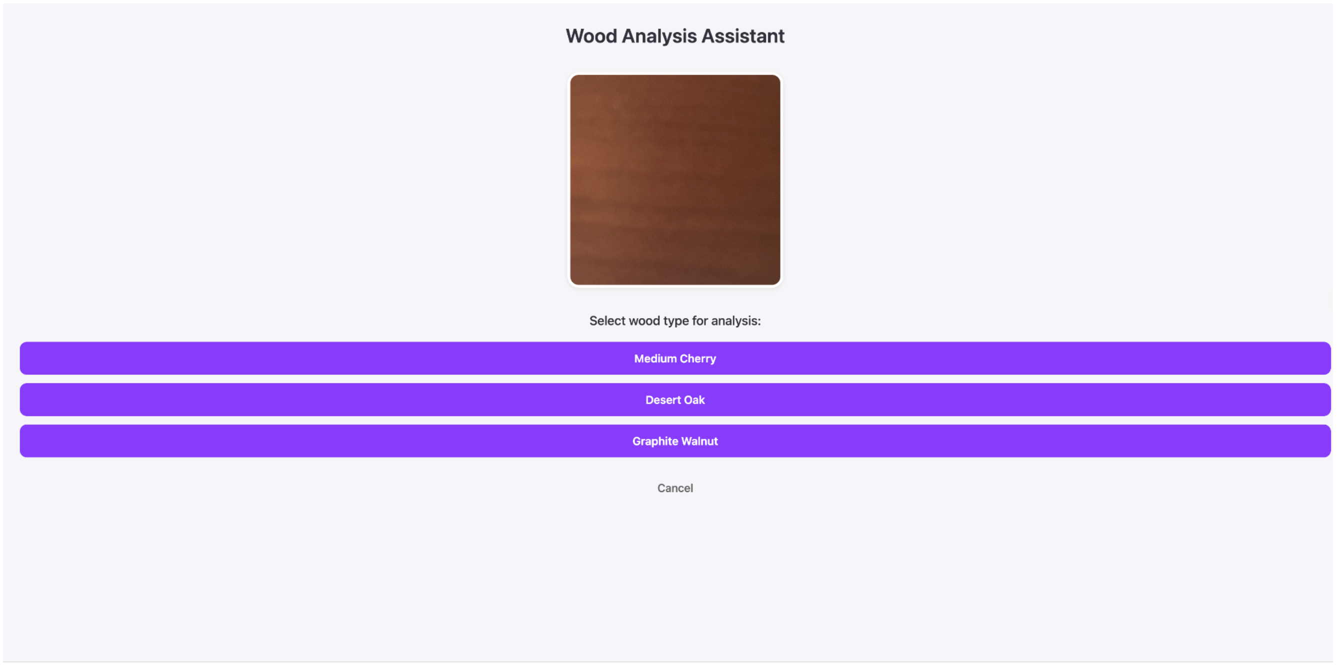
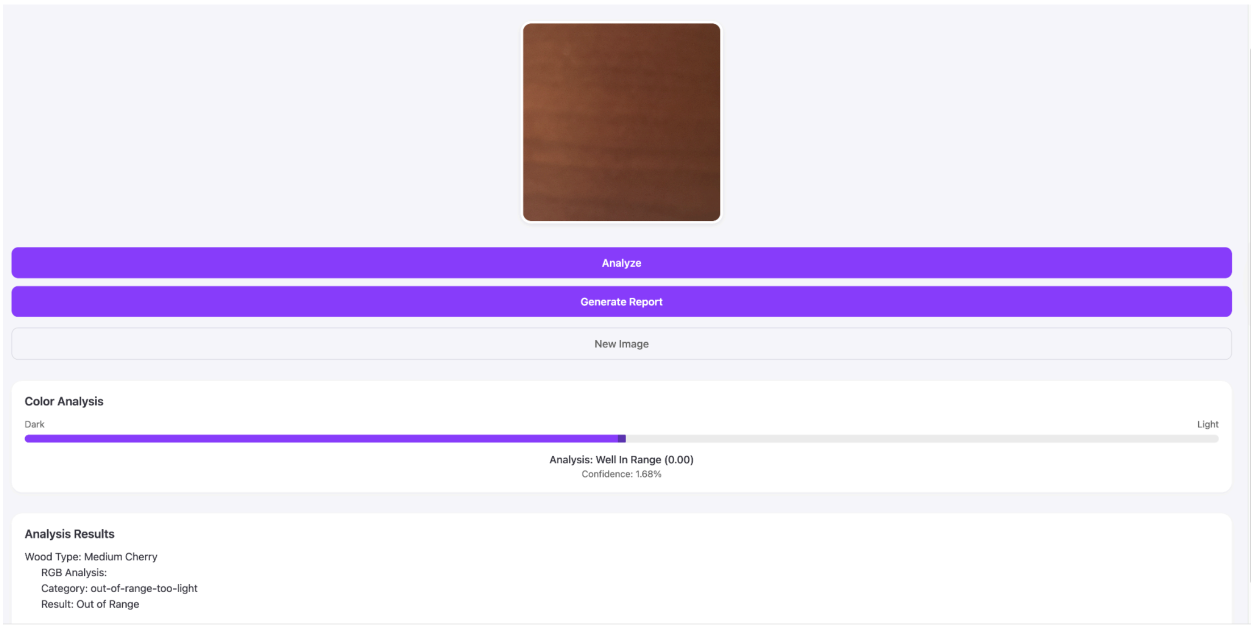


Figure 9: User Interface: Validation Results: Analyze



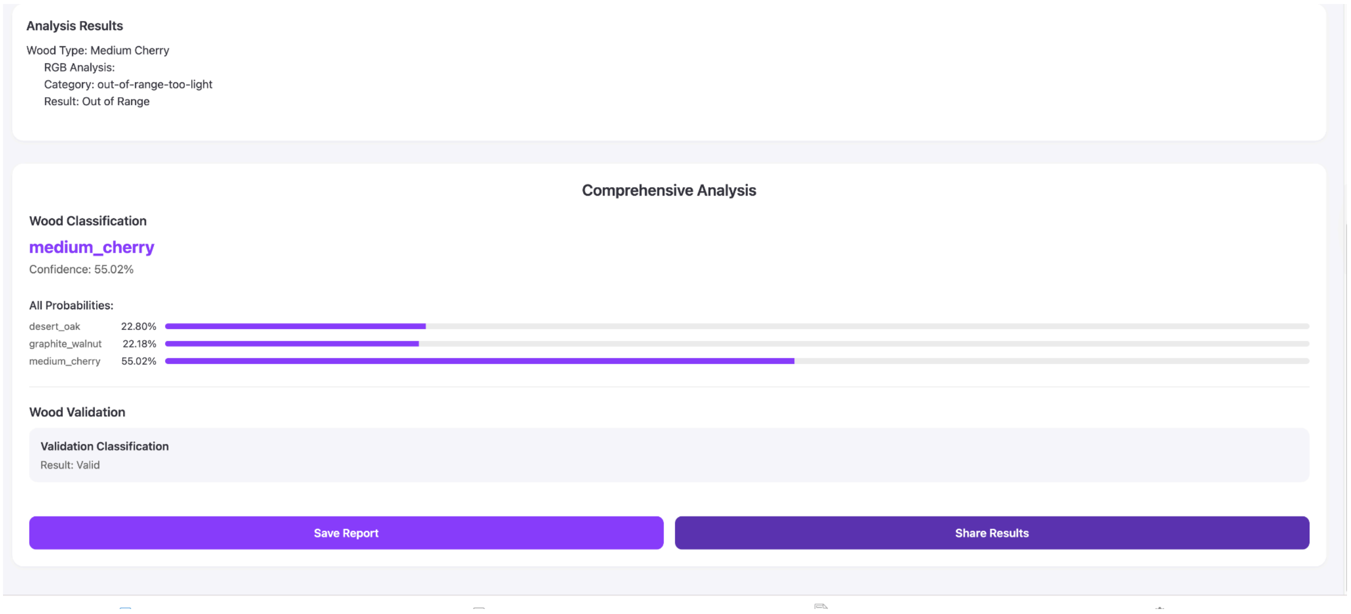
Analysis: Well In Range (0.00)
Confidence: 1.68%

Analysis Results
Wood Type: Medium Cherry
RGB Analysis:
Category: out-of-range-too-light
Result: Out of Range

Figure 10: User Interface: Validation Results: After Analyze

2. **Generate a Full Report** – The image is analyzed across all three veneer types (**Medium Cherry, Desert Oak, and Graphite Walnut**) to provide a

comprehensive assessment telling you whether the veneer is valid or not (in range or out of range). After the comprehensive assessment a new button, "save report" is prompted. This allows you to save the report to firebase. This is demonstrated in **Figure 11**.



Analysis Results

Wood Type: Medium Cherry
RGB Analysis:
Category: out-of-range-too-light
Result: Out of Range

Comprehensive Analysis

Wood Classification

medium_cherry

Confidence: 55.02%

All Probabilities:

desert_oak	22.80%
graphite_walnut	22.18%
medium_cherry	55.02%

Wood Validation

Validation Classification
Result: Valid

Figure 11: User Interface: Validation Results: Full Report

Validation Screen Heuristic Analysis (Figures 8, 9, 10, 11)

1. **Flexibility and Efficiency of Use**
   - Application: The user is provided with two validation options: analyze based on classified color or generate a full report. This allows both novice users (who want a simple classification) and expert users (who want detailed analysis) to efficiently use the application.

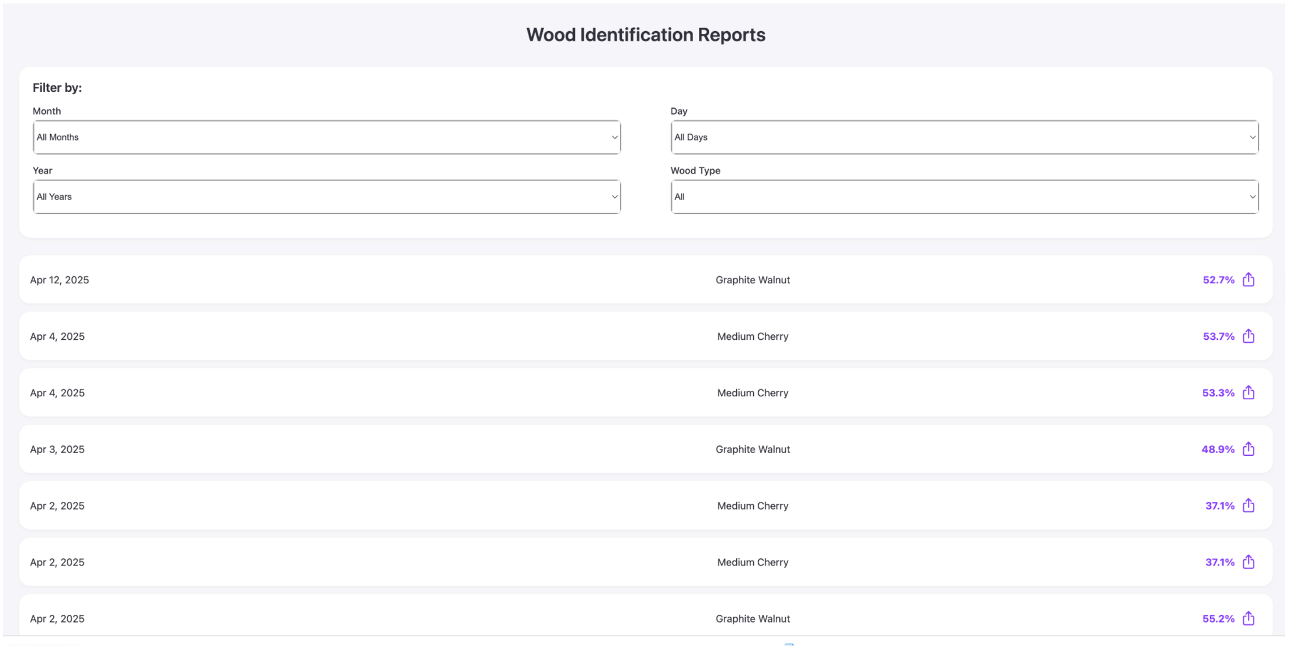
2. **Error Recovery and Prevention**
   - Application: The Reupload button lets users return to the Image Picker screen if they realize they selected the wrong image. This prevents users from submitting incorrect images for validation, reducing errors in the analysis process.

3. **Aesthetic and Minimalist Design**
   - Application: The Validation Screen focuses on the image preview and two clear options for analysis, avoiding unnecessary distractions. A clean and

simple interface helps users focus on the task at hand without being overwhelmed by excessive buttons or text.

Reports Screen

Clicking on the middle tab (**Reports**) brings you to the report screen. The Reports screen maintains a complete history of your wood validation analyses, displaying them in reverse chronological order for easy access. An intuitive scrollable interface lets you browse results, while powerful filters allow you to pinpoint specific validations by date (day/month/year) or wood type. This organized system ensures you can always retrieve and review past analyses with precision.



Wood Identification Reports

Filter by:

Month: All Months	Day: All Days
Year: All Years	Wood Type: All

Apr 12, 2025	Graphite Walnut	52.7%
Apr 4, 2025	Medium Cherry	53.7%
Apr 4, 2025	Medium Cherry	53.3%
Apr 3, 2025	Graphite Walnut	48.9%
Apr 2, 2025	Medium Cherry	37.1%
Apr 2, 2025	Medium Cherry	37.1%
Apr 2, 2025	Graphite Walnut	55.2%

Figure 11: User Interface: Reports Screen

Report Screen Heuristic Analysis (Figure 12)

- **Flexibility and Efficiency of Use**
    - Users can quickly access past reports and utilize search and filter functions to find relevant data efficiently. This function allows both general users who may browse reports casually and expert users (who require precise

historical data).

- **Error Prevention**
    - The structured format ensures that users do not lose saved reports, allowing them to revisit previous analyses anytime. The persistent storage of reports and precise filtering options prevent data misinterpretation by maintaining historical accuracy and enabling targeted analysis of specific results.
- **Aesthetic and Minimalist Design**
    - The interface follows a clean and distraction-free layout, ensuring clarity in navigation and focusing on core functionalities. Key elements, such as filtering options, are intuitive and accessible without cluttering the screen.

Appendix

Members

Zuhair Al Araf - Programmer, Backend

Email: zaraf3@gatech.edu

Rishi Manimaran - Programmer, Backend

Email: rishimj@gmail.com

Benson Lin - Programmer, Frontend

Email: linbenson0401@gmail.com

Zhihui Chen - Programmer, Backend

Email: zchen0630@gmail.com

Jihoon Kim - Programmer, Frontend

Email: jikim1803@gmail.com