Detail Design:

# Architecture:

For my project skin disease detection using picture combining deep learning and machine learning for skin lesion analysis, a modular and layered architecture is ideal.

### High-Level Architecture

1. **User Interface Layer:**

**Purpose**: enable user interactions with the system such as uploading images, viewing results and exporting reports.

**Components:**

Android app: Built with Java using Android Studio.

**Web Interface:** users upload pre-captured images using a browser.

1. **Communication Layer:**

Both platforms communicate with a **Backend Server** using **RESTful APIs**. This ensures consistency and centralizes the image processing and disease detection logic.

1. **Backend Layer:**

The backend handles the core functionality of the system, including preprocessing, disease detection, and result generation:

**Image Preprocessing**: Uses OpenCV for resizing, normalization, and segmentation.

**Model Inference**: Combines predictions from:

**Deep Learning Models** (CNN with PyTorch).

**Machine Learning Models** (Random Forest, SVM with Scikit-learn).

And it may use ensemble techniques to aggregate predictions.

**Result Generation**: Generates confidence scores, disease names, and visualizations (e.g., heatmaps).

1. **Data Layer MongoDB stores:**

The data layer handles data storage and retrieval:

* User uploaded images
* Preprocesses images.
* Model analysis results.
* Patient record and reports.

1. **Reporting Layer:**

This layer provides the following functionalities:

**Mobile app:** Displays results such as disease type, confidence score and allows report downloads.

**Web Interface:** Allows report download and visualization on larger screens

# Graphic Description:

**Frontend:**

* **Sign In/Up**: Allows users to create an account or sign in to access personalized features. This process connects to the **Communication Layer** to handle user credentials and account creation.
* **Guest Sign In**: Provides a way for users to interact without creating an account, likely with limited functionality.
* **Request Previous Result**: Users can query previously processed results, sent through the Communication Layer to fetch saved analysis results.
* **Upload Image**: Enables users to upload an image for analysis. This action passes the image to the Communication Layer for preprocessing and analysis.

**Communication Layer:**

* **Sign In/Up and Guest Sign In**: Authenticates the user or creates a new account, passing the request to the Backend Layer for validation and account handling.
* **Upload Image**: Handles the upload request by passing the image data to the backend's preprocessing module for further analysis.
* **Request Previous Result**: Fetches user-specific analysis results from the database by interacting with the Backend Layer.

**Backend:**

* **Preprocess Image**: Prepares the uploaded image for machine learning analysis by resizing, normalizing, or extracting key features.
* **ML Server (Machine Learning Model)**: The ML server processes the preprocesses image using models like CNN (ResNet50 or InceptionV3). The analysed data is sent back to the backend.
* **Analyze Image**: Coordinates with the ML server to process the image, interprets results, and formats them for storage or display.
* **Retrieve User Result**: Fetches past analysis data from the database based on user requests. Useful for displaying history or trends.
* **Analysis Results**: Manages and organizes results before they are saved in the database or displayed to the user.
* **Check User:** Verifies if a user exists in the database and fetches their data or creates a new user account if not exists.

**Data Layer:**

* **Stores User Data**: Manages storage for user credentials, uploaded images, and processed analysis results. Interacts with the backend to save and retrieve information.

**Reporting Layer:**

* **Display Result**: Presents processed analysis results to the user in a clear and interpretable format. This step concludes the user's workflow.

## Class Diagram:

**Classes**

**User**: Represent a user in the system, responsible for managing user accounts, authentication, and user-specific data.

**Image**: represent an image uploaded by a user, containing both the raw image data and associated metadata such as the user who uploaded it and the upload date.

**Result**: represents the outcome of an image analysis process. It stores the analysis data generated after an image is processed by a machine learning model in the database through the backend.

**Backend**: The Backend class acts as a processing engine that handles the various tasks related to image manipulation and analysis. Its main responsibility is to preprocess images, making them suitable for analysis, and to invoke machine learning models to analyze the preprocessed images.

**MLModel**: the MLModel class encapsulates the machine learning model that performs the analysis on images. Its purpose is to abstract the model’s functionality and provide an interface for interacting with different types of machine learning models, such as deep learning architectures like ResNet50 or InceptionV3.

**Database**: The Database class is responsible for all interactions with the database, ensuring that data is correctly stored, retrieved, and managed.

It provides a centralized mechanism for saving users, images, and analysis results to the database, ensuring data consistency and integrity across the system.

**APIHandler**:

The APIHandler class acts as an intermediary between the system's frontend and backend, handling HTTP requests from external sources such as client applications.

**Arrows:**

**Result<->User**: This indicates that a User class is associated with the Result class, meaning users receive multiple results after processing their uploaded images.

**User<->Image** : This indicates that the User class is associated with the Image class, meaning users interact with images by uploading them.

**Backend<->Image/Result:** The Backend class depends on the Image and Result classes because it processes images and generates results. This relationship shows that changes in the Image or Result classes might affect the Backend class's functionality.

**Backend<-> ML Model:** The Backend class depends on the MLModel class to perform predictions. The ML models are the core processing unit that analyzes data (images) and generates predictions (results).

**Backend<->Database:** The Backend class interacts with the Database class to store and retrieve information, such as uploaded images, user data, and results.

**APIHnadler<->Backend:** The APIHandler class serves as a bridge between frontend requests and the backend. It depends on the Backend class to perform operations like uploading images, retrieving results, or handling user data.

## Algorithm description

**Deep Learning Algorithm**

**Convolutional Neural Networks (CNNs)**

CNNs form the backbone of modern image analysis tasks. They are particularly well-suited for handling unstructured data, like images, and excel at automatically learning hierarchical features from raw data. Key advantages include:

* **Feature Extraction**: CNNs learn low-level features (edges, textures) in early layers and complex patterns (shapes, objects) in deeper layers.
* **Spatial Hierarchies**: By using convolutional layers and pooling layers, CNNs capture spatial dependencies, making them ideal for analyzing images.

In this project, CNNs can process user-uploaded images to:

* Detect specific conditions, lesions, or abnormalities.
* Generate **heatmaps** to visualize areas of interest in an image, aiding interpretability.
* Provide accurate predictions by leveraging their ability to recognize patterns.

**ResNet (Residual Networks)**

ResNet, or Residual Networks, is a CNN architecture designed to address the **vanishing gradient problem** in deep networks. By introducing **residual connections**, ResNet enables deeper networks to train effectively without performance degradation.

Key features:

* **Residual Blocks**: ResNet introduces shortcut connections that skip one or more layers, allowing the network to learn identity mappings. This ensures that deep networks don't suffer from degradation problems.
* **Scalability**: ResNet can scale to hundreds or thousands of layers (e.g., ResNet-50, ResNet-101).
* **Transfer Learning**: Pre-trained ResNet models on large datasets (e.g., ImageNet) can be fine-tuned for specific tasks, drastically reducing training time and improving accuracy.

**Use in my project**:  
ResNet can identify subtle patterns in high-resolution images, making it ideal for detecting abnormalities, such as medical conditions or lesions. Its robustness ensures high accuracy and reliability in predictions.

**Inception (GoogLeNet)**

Inception networks, particularly **Inception-v3**, are known for their innovative use of **multi-scale feature extraction** and parameter efficiency. Instead of stacking layers sequentially, Inception networks use **Inception modules** to process multiple filter sizes simultaneously.

Key features:

* **Inception Modules**: These modules allow the network to capture patterns at different scales by applying 1x1, 3x3, and 5x5 convolutions in parallel.
* **Parameter Efficiency**: By factorizing convolutions and using 1x1 convolutions to reduce dimensionality, Inception achieves high performance with fewer parameters.
* **Auxiliary Classifiers**: Intermediate outputs during training help prevent vanishing gradients and improve convergence.

**Use in my project**:  
Inception is well-suited for image analysis tasks where features at different scales are important. For instance, it can detect lesions of varying sizes and generate detailed heatmaps, improving both performance and interpretability.