Design for Social Innovation

ML-Assisted HTP Interpretation Platform

Project Report

Team TelClub

Akmal
Arihant
Deepraj
Devansh
Rohan
Aktar

November 17, 2025

Abstract

This report details the architecture, design, and implementation of the ML-Assisted House-Tree-Person (HTP) Interpretation Platform, a project undertaken for the Design for Social Innovation course. The HTP test is a valuable psychological tool, but its manual interpretation is subjective, time-consuming, and requires extensive expertise. Our platform addresses these challenges by introducing a two-stage machine learning pipeline to automate the preliminary analysis of HTP drawings. The system architecture is a containerized, service-oriented design comprising a React frontend, a Node.js backend, and a Python-based ML service. The ML pipeline features a fine-tuned EfficientNet model (the "Seer") for objective visual feature detection and a Large Language Model (Google Gemini) enhanced with Retrieval Augmented Generation (the "Interpreter") to produce clinically grounded psychological reports. This "expert-in-the-loop" solution is designed to augment, not replace, the clinical psychologist, thereby increasing efficiency, improving objectivity, and allowing professionals to focus on higher-level interpretation and patient care. This document provides a comprehensive overview of the project's problem statement, technical and ML architecture, data models, user workflows, API specifications, and deployment procedures.

Contents

1 Problem Statement

The House-Tree-Person (HTP) projective test is a widely utilized psychological tool for inferring aspects of an individual's personality, emotional state, and psychological well-being. However, the interpretation process is inherently subjective, labor-intensive, and demands extensive expertise, often leading to inconsistencies. The development of a technological solution for HTP analysis is hindered by several significant barriers:

- **Domain Ambiguity & Label Noise:** HTP interpretation is highly theoretical and varies across clinicians. The lack of standardized criteria for "risk" markers leads to inconsistent labels, making it difficult to train reliable models.
- **Data Scarcity and Quality:** High-quality, expertly annotated HTP datasets are not readily available. Existing data is often heterogeneous, complicating the training and validation of robust predictive models.
- **Subjectivity and Risk of Bias:** An automated system risks perpetuating or even amplifying biases present in the training data. The ethical implications of a misdiagnosis based on a flawed algorithm are significant.

Our project, the **ML-Assisted HTP Interpretation Platform**, aims to address these limitations by providing an objective, automated preliminary analysis, specifically focusing on tree drawings. The goal is not to replace the psychologist but to create an "expert-in-the-loop" system that handles the initial, repetitive screening. This empowers clinicians by saving significant time, providing an objective second opinion, and allowing them to dedicate more resources to nuanced interpretation and direct patient interaction.

2 System Architecture

The platform is built on a modular, service-oriented architecture, fully containerized with Docker for consistency, scalability, and ease of deployment. It consists of three primary, decoupled services: a React Frontend, a Node.js Backend, and a Python ML Service.

2.1 Technical Architecture

The three core services communicate via REST APIs over a shared Docker network, ensuring a clear separation of concerns and allowing each component to be developed and scaled independently.

- **Frontend:** A React single-page application served by Nginx. It provides a dynamic and role-based user interface for all three stakeholders (Uploader, Assessor, Admin) and communicates exclusively with the Backend API.
- **Backend (Node.js/Express):** The application's central nervous system. It was chosen for its non-blocking I/O, which is highly efficient for managing concurrent API requests and database interactions. It handles all business logic, user authentication (JWT), role-based access control (RBAC), database modeling (MongoDB with Mongoose), and asynchronous job queuing for ML analysis.
- **ML Service (Python/Flask):** A dedicated microservice that encapsulates the entire machine learning pipeline. Python was chosen due to its extensive ecosystem of ML libraries (PyTorch, Timm, etc.). This isolation prevents ML dependencies from bloating the backend and allows the ML component to be updated or scaled independently.

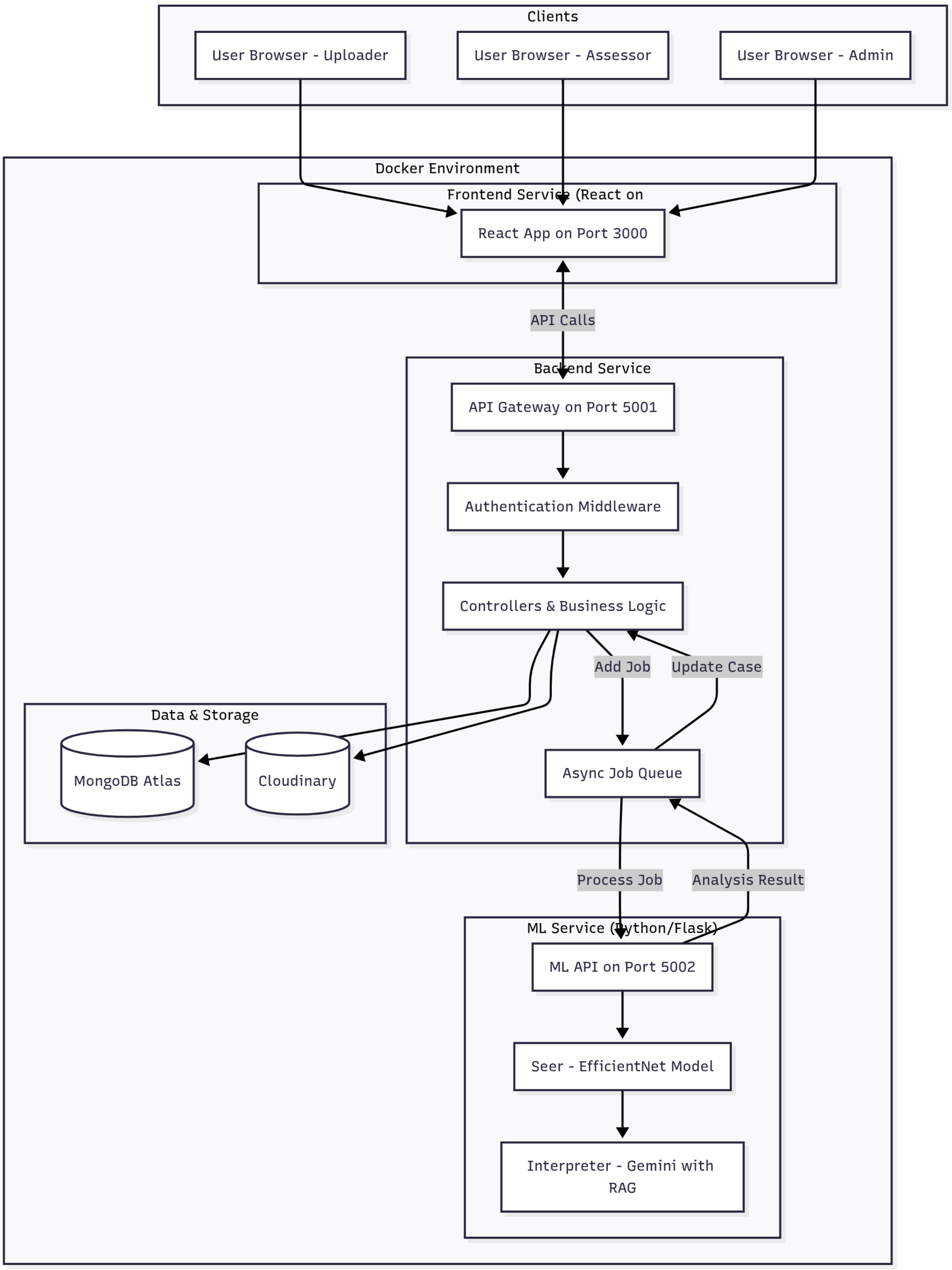


Figure 1: Overall Technical Architecture Diagram

Architectural Diagram Explanation

The diagram above illustrates the flow of information through the system. Users interact with the **Frontend**, which sends API requests to the **Backend**. The backend authenticates these requests and performs business logic, interacting with external services like **MongoDB Atlas** for data persistence and **Cloudinary** for image storage. For drawing analysis, the backend does not process the request synchronously. Instead, it adds a job to an in-memory **Async Job Queue** and immediately responds to the user. A worker process picks up jobs from this queue and sends them to the **ML Service**. The ML service performs its two-stage analysis and returns the result, which is then used to update the case record in the database. This asynchronous pattern ensures the user interface remains responsive and the system can handle long-running ML tasks without timing out.

2.2 Machine Learning Pipeline

Our core AI pipeline, hosted in the ML Service, operates in a two-stage process conceptually named "Seer" and "Interpreter". This hybrid approach was deliberately chosen to maximize objectivity while leveraging the nuanced understanding of modern LLMs.

1. **Stage 1: The "Seer" - Visual Feature Extraction**
    - **Purpose:** To objectively identify the presence of specific, predefined HTP drawing features from an uploaded image.
    - **Model:** A fine-tuned `EfficientNet-B0` model, pre-trained on ImageNet. We employed transfer learning, freezing the convolutional base and training only the final classification head. This is a highly effective technique for adapting powerful models to specialized tasks with limited data.
    - **Output:** A binary prediction vector indicating which of the 26 defined features are present.
2. **Stage 2: The "Interpreter" - Psychological Report Generation**
    - **Purpose:** To translate the raw visual features from the "Seer" into a human-readable psychological report that is grounded in established HTP interpretation guidelines.
    - **Model:** Google's Gemini Flash model.
    - **Approach: Retrieval Augmented Generation (RAG).** This approach was critical to mitigate the risk of LLM "hallucination" and ensure the generated interpretations are traceable and clinically relevant.
        (a) **Knowledge Base:** A meticulously compiled text document (`knowledge_base.txt`) serves as the ground truth.
        (b) **Retrieval:** The detected features are used as a query to retrieve the most relevant text snippets from the knowledge base using semantic search (cosine similarity on text embeddings).
        (c) **Generation:** The retrieved snippets are injected directly into the LLM prompt along with the detected features. The model is explicitly instructed to base its interpretation *only* on the provided context, transforming a general-purpose LLM into a specialized, auditable HTP interpreter.

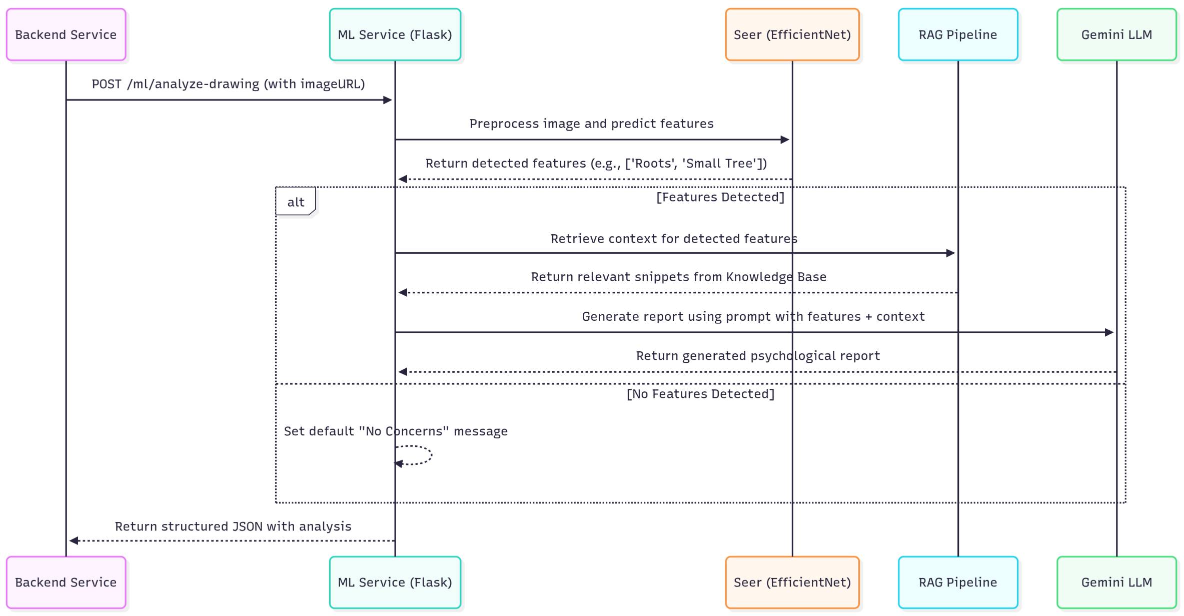

Backend Service
ML Service (Flask)
Seer (EfficientNet)
RAG Pipeline
Gemini LLM
POST /ml/analyze-drawing (with imageURL)
Preprocess image and predict features
Return detected features (e.g., ['Roots', 'Small Tree'])
alt
[Features Detected]
Retrieve context for detected features
Return relevant snippets from Knowledge Base
Generate report using prompt with features + context
Return generated psychological report
[No Features Detected]
Set default "No Concerns" message
Return structured JSON with analysis

Figure 2: ML "Seer" and "Interpreter" Pipeline (Placeholder)

ML Pipeline Diagram Explanation

The diagram details the internal workflow of the ML service. When the **Backend** sends a request, the **ML Service** first passes the image to the **Seer (EfficientNet)** model, which outputs a list of raw visual labels. If any labels are detected, the pipeline proceeds to the RAG stage. The labels are used to query a vector store built from our clinical knowledge base. The top-matching interpretations are retrieved and combined with the labels into a detailed prompt for the **Gemini LLM**. The LLM synthesizes this information into a coherent report. This structured JSON output, containing both the raw labels and the generated interpretation, is then sent back to the Backend.

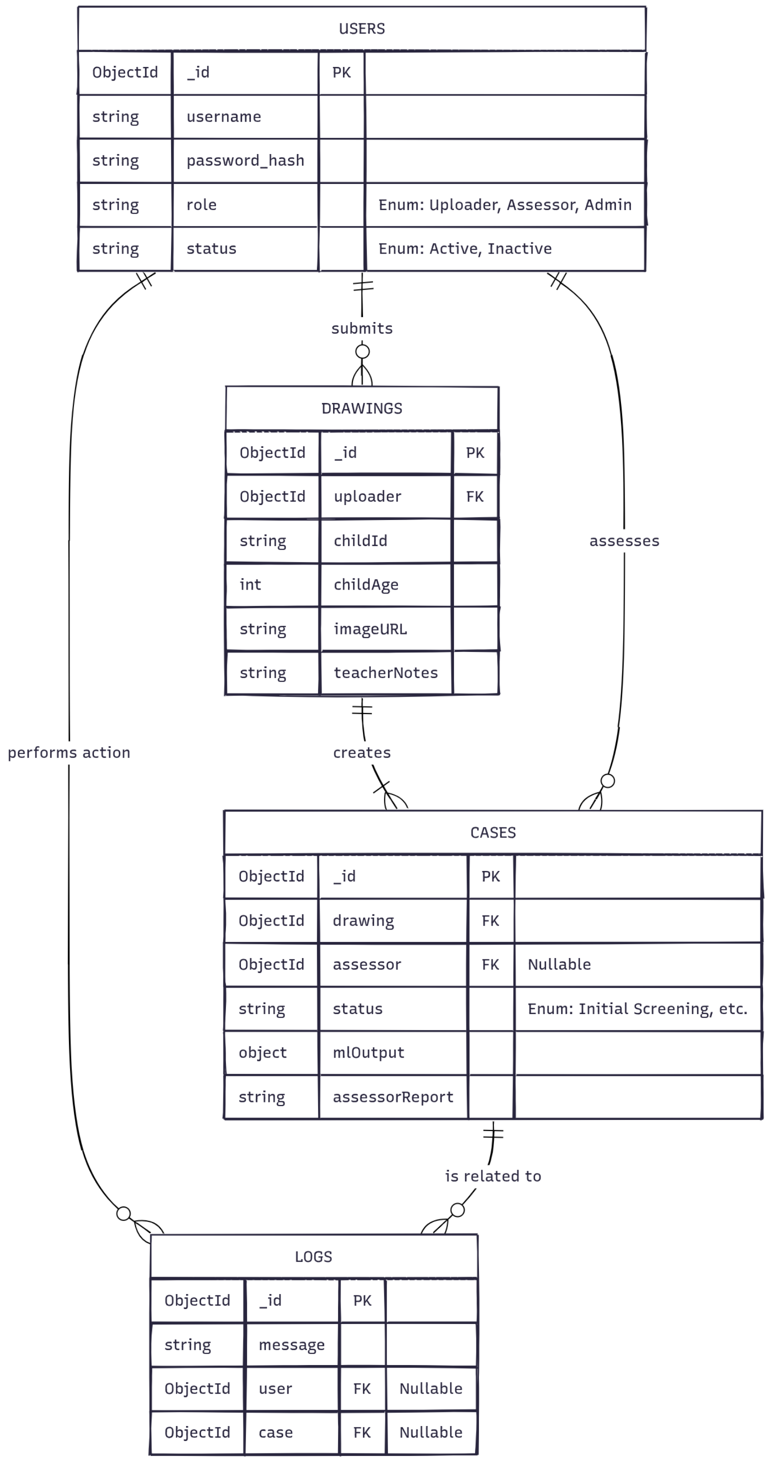
2.3 ML Model Training and Performance

- **Dataset:** A custom multi-label dataset of approximately 1000+ HTP tree drawings was sourced from Roboflow. Each image was annotated with the presence or absence of 26 distinct HTP features.
- **Training Process:** We used transfer learning with an `EfficientNet-B0` model. The convolutional layers were frozen, and only the final linear classification head was trained for 30 epochs. To address the significant class imbalance (some features are very rare), we used a weighted loss function, `nn.BCEWithLogitsLoss`, with a `pos_weight` tensor calculated from the training data distribution.
- **Performance Analysis:** The provided training log reveals key insights:
  - **Training Loss:** Decreased steadily from 1.3298 to 0.4143, indicating that the model was effectively learning from the training data.
  - **Validation Loss:** Increased significantly over time (from 1.5174 to 2.3436), diverging from the training loss. This is a clear indicator of **overfitting**. The model learned the training data too well and struggled to generalize to unseen validation data.
  - **Validation F1-Score:** The F1-score (micro-average) peaked at **0.3849** around epoch 28 before plateauing. While the model shows some predictive capability, this score is

relatively low for a production system, a direct consequence of the small and specialized dataset. The final F1-score on the unseen test set was **0.3729**, confirming the generalization challenge.

- **Conclusion:** The "Seer" model is a functional proof-of-concept but is currently overfitting. Its primary limitation is the small dataset size. Future work must focus on data augmentation, acquiring more data, or exploring more advanced regularization techniques to improve its generalization performance.

3 Database ER Diagram



USERS			
ObjectId	_id	PK	
string	username		
string	password_hash		
string	role		Enum: Uploader, Assessor, Admin
string	status		Enum: Active, Inactive

DRAWINGS		
ObjectId	_id	PK
ObjectId	uploader	FK
string	childId	
int	childAge	
string	imageURL	
string	teacherNotes	

CASES			
ObjectId	_id	PK	
ObjectId	drawing	FK	
ObjectId	assessor	FK	Nullable
string	status		Enum: Initial Screening, etc.
object	mlOutput		
string	assessorReport		

LOGS			
ObjectId	_id	PK	
string	message		
ObjectId	user	FK	Nullable
ObjectId	case	FK	Nullable

Figure 3: Entity-Relationship Diagram

The system uses MongoDB, a NoSQL database, but the data is structured relationally using Mongoose schemas to enforce consistency and define relationships. The four core collections are Users, Drawings, Cases, and Logs.

ER Diagram Explanation

The diagram shows the relationships between the main data entities:

- A **USER** can submit many **DRAWINGS**.
- Each **DRAWING** creates exactly one **CASE**.
- A **CASE** is assessed by at most one **USER** (with the 'Assessor' role). The 'assessor' field is nullable because cases are initially unassigned.
- **LOGS** are created to track significant system events, linked to the **USER** who performed the action and the **CASE** it relates to, providing a complete audit trail.

4 User Flows

Each stakeholder (Uploader, Assessor, Admin) has a distinct workflow tailored to their responsibilities within the platform.

4.1 Uploader User Flow

The Uploader's primary role is data ingestion. The workflow is designed to be simple and efficient, suitable for mobile or web interfaces. The key architectural feature here is the asynchronous processing, which provides immediate feedback to the Uploader while the heavy lifting happens in the background.

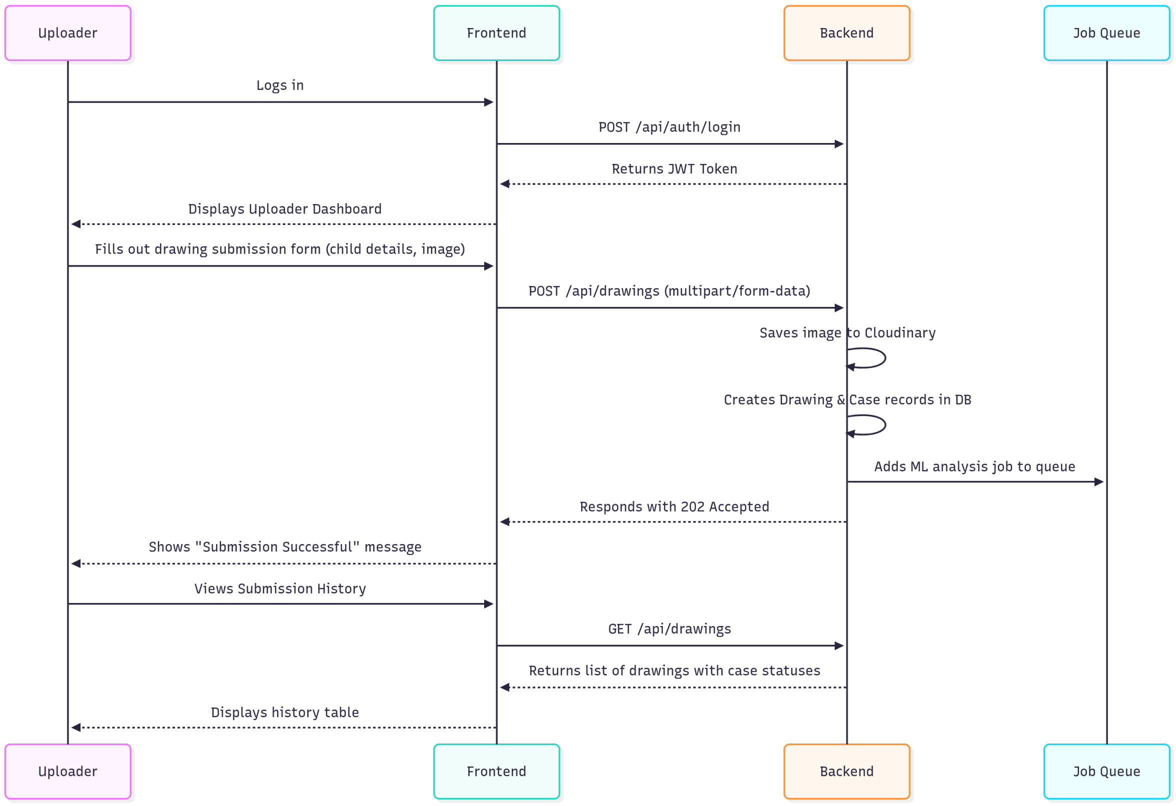

Uploader
Frontend
Backend
Job Queue
Logs in
POST /api/auth/login
Returns JWT Token
Displays Uploader Dashboard
Fills out drawing submission form (child details, image)
POST /api/drawings (multipart/form-data)
Saves image to Cloudinary
Creates Drawing & Case records in DB
Adds ML analysis job to queue
Responds with 202 Accepted
Shows "Submission Successful" message
Views Submission History
GET /api/drawings
Returns list of drawings with case statuses
Displays history table

Figure 4: Uploader Workflow

4.2 Assessor User Flow

The Assessor is the "expert-in-the-loop," responsible for reviewing ML-flagged cases and providing a final professional judgment. This workflow is centered around a dedicated review workspace. A round-robin assignment service in the backend automatically distributes new flagged cases to the assessor with the lightest workload, ensuring balanced case distribution.

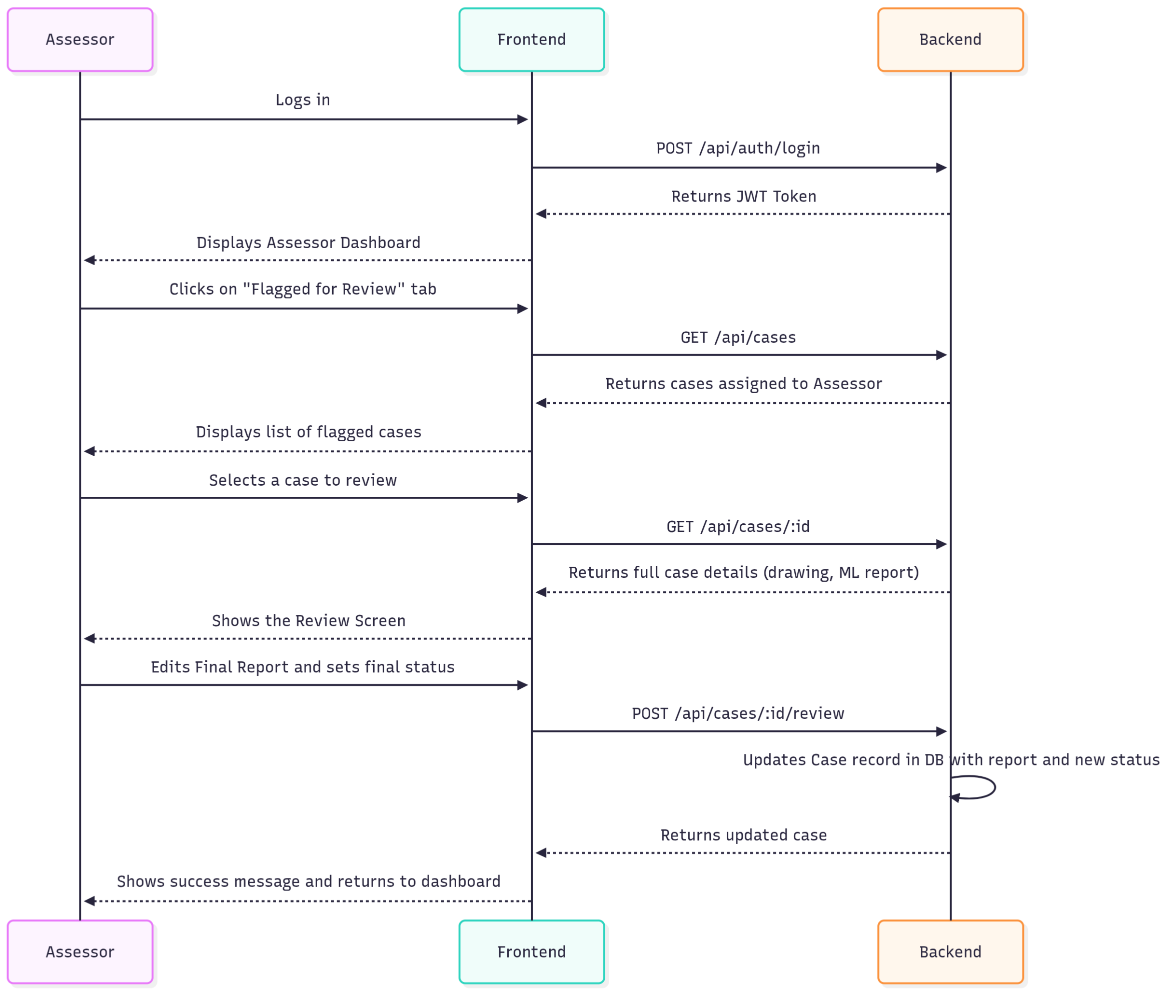

Assessor
Frontend
Backend
Logs in
POST /api/auth/login
Returns JWT Token
Displays Assessor Dashboard
Clicks on "Flagged for Review" tab
GET /api/cases
Returns cases assigned to Assessor
Displays list of flagged cases
Selects a case to review
GET /api/cases/:id
Returns full case details (drawing, ML report)
Shows the Review Screen
Edits Final Report and sets final status
POST /api/cases/:id/review
Updates Case record in DB with report and new status
Returns updated case
Shows success message and returns to dashboard
Assessor
Frontend
Backend

Figure 5: Assessor Workflow

4.3 Admin User Flow

The Admin has complete oversight of the entire system, including user management, case auditing, and monitoring system health via an analytics dashboard.

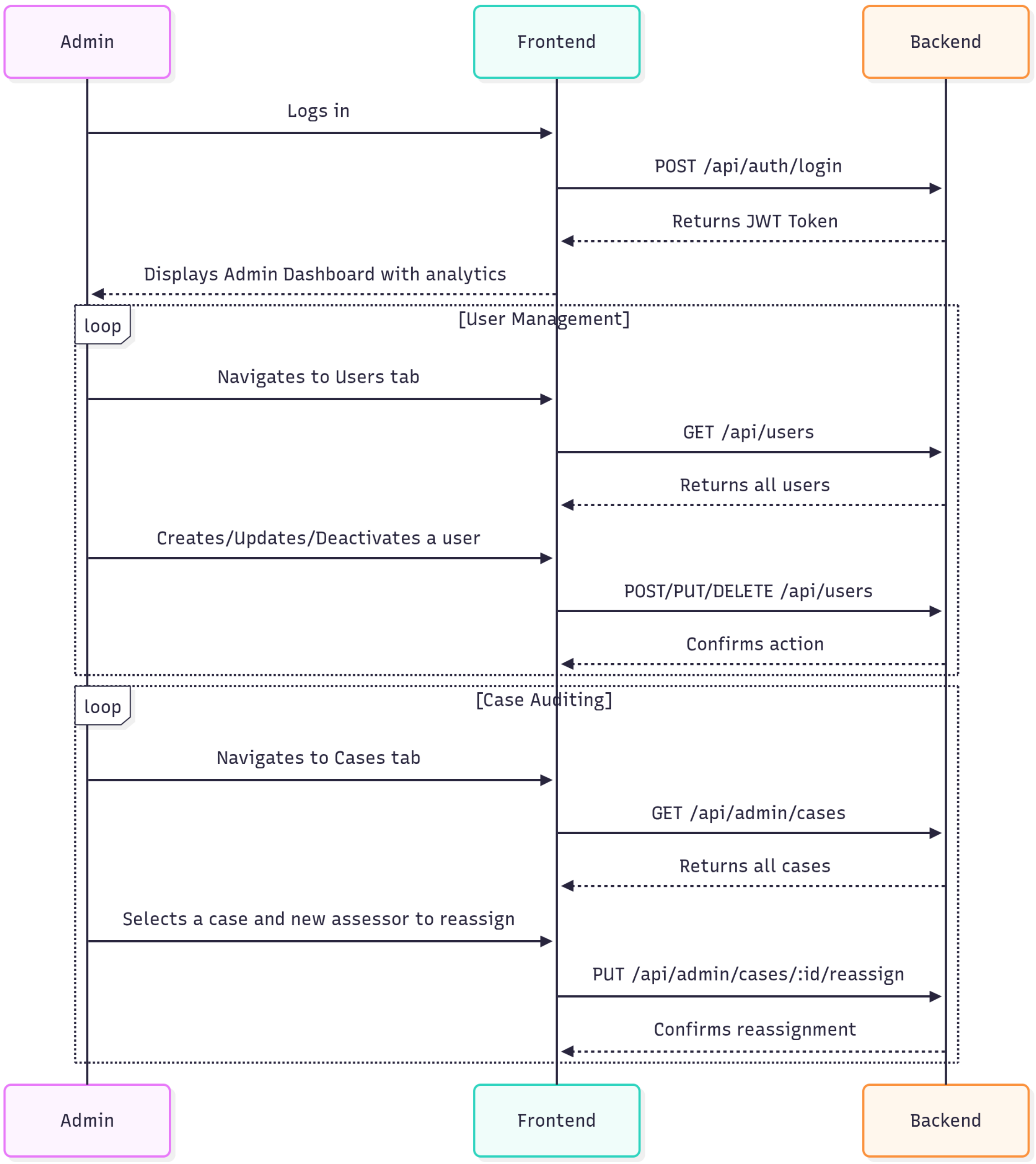


Figure 6: Admin Workflow

5 API Endpoints

The platform exposes a set of RESTful API endpoints, secured with JWT and role-based access control.

Endpoint	Access	Description
Authentication Routes (‘/api/auth‘)		
POST /login	Public	Authenticates a user. **Request Body:** `{"username": "...", "password": "..."}` **Success (200):** `{"token": "...", "user": {...}}` **Error (401):** Invalid credentials.
GET /me	Authenticated	Fetches the current user’s profile. **Headers:** `Authorization: Bearer <token>` **Success (200):** User object. **Error (401):** Invalid token.
Drawing Routes (‘/api/drawings‘)		
POST /	Uploader	Submits a new drawing for analysis. **Request Body:** multipart/form-data Fields: `childId, childAge, teacherNotes, image`. **Success (202 Accepted):** Acknowledges asynchronous processing. **Error (400):** Missing image file.
GET /	Uploader	Retrieves submission history for logged-in uploader, including case statuses. **Success (200):** Array of drawing objects.
Case Routes (‘/api/cases‘)		
GET /	Assessor	Retrieves all cases assigned to the assessor. **Success (200):** Array of case objects.
GET /:id	Assessor, Admin	Fetches full details of a single case. **Success (200):** Case object with drawing metadata. **Error (404):** Case not found.
POST /:id/review	Assessor	Submits the final review for a case. **Request Body:** `{"finalStatus": "...", "assessorReport": "..."}` **Success (200):** Updated case object. **Error (403):** Not the assigned assessor.
User Management Routes (‘/api/users‘)		
GET /	Admin	Retrieves all users.
POST /	Admin	Creates a new user. **Request Body:** `{"username": "...", "password": "...", "role": "...", "status": "..." }`
PUT /:id	Admin	Updates a user’s details.
DELETE /:id	Admin	Soft deletes (deactivates) a user.
Admin Routes (‘/api/admin‘)		
GET /analytics	Admin	Retrieves aggregated system analytics.
GET /cases	Admin	Fetches all cases in the system for auditing.
PUT /cases/:id/reassign	Admin	Reassigns a case to another assessor.

Endpoint	Access	Description
		Request Body: `{"newAssessorId": "..." }`
GET /logs	Admin	Retrieves recent system logs.
ML Service (‘/ml/analyze-drawing‘)		
POST /	Backend	Runs ML analysis on a drawing. **Request Body:** `{"imageURL": "https://..." }` **Success (200):** Example: `{"flaggedForReview": true, "psychIndicators": [...], "modelVersion": "..." }`

6 Folder Structure

The project is organized into a monorepo structure with distinct directories for each service, promoting clean separation and independent development.

6.1 Top-Level Structure

```
dsi-platform/
|-- backend/          # Node.js/Express application
|-- frontend/         # React application
|-- ml-api/           # Python/Flask ML service
|-- docker-compose.yml # Main Docker Compose file
|-- .env.example      # Environment variable template
```

6.2 Backend Folder Structure

The backend follows a standard MVC-like pattern, separating concerns into routes, controllers, models, and services.

```
backend/
|-- config/
|   `-- db.js               # MongoDB connection
|-- controllers/
|   |-- auth.controller.js
|   `-- ... (other controllers)
|-- middlewares/
|   |-- auth.middleware.js    # JWT and RBAC
|   `-- upload.middleware.js  # Multer/Cloudinary setup
|-- models/
|   |-- User.model.js
|   `-- ... (other models)
|-- routes/
|   |-- index.js              # Main router file
|   `-- ... (route files)
|-- services/
|   |-- ml.service.js         # Logic to call the ML API
|   |-- assignment.service.js # Round-robin assessor assignment
|   |-- jobQueue.service.js   # Async job queue
|   `-- log.service.js        # System logging service
`-- server.js                 # Main server entry point
```

7 Setup Instructions

The entire application stack is containerized and managed by Docker Compose for a streamlined, one-command setup process. The following steps guide a developer in setting up the project for local development.

7.1 Prerequisites

1. **Docker & Docker Compose:** Latest versions installed on your host machine.
2. **Miniconda/Anaconda:** A local Conda installation is required to manage the Python environment for the ML service during development.
3. **Cloud Accounts & API Keys:**
   - A MongoDB Atlas connection string.
   - Cloudinary account credentials (‘`CLOUD_NAME`‘, ‘`API_KEY`‘, ‘`API_SECRET`‘).
   - A Google AI Studio API key for the Gemini model.

7.2 Configuration Steps

1. **Clone the Repository:**

   ```
   git clone <repository_url>
   cd Tree-psych-eval
   ```

2. **Configure Environment Variables:** Create a single ‘.env‘ file in the root of the project by copying the example file. This file will hold all necessary credentials for all services.

   ```
   cp .env.example .env
   ```

   Now, open the newly created ‘.env‘ file and replace the placeholder values with your actual credentials.

3. **Prepare Local Conda Environment Python version 3.13 (for Dev):** The ‘docker-compose.yml‘ file mounts your local Conda environment’s packages into the ML container. This requires a one-time setup on your host machine.

   (a) Create a Conda Environment (e.g., ‘htp‘).

   ```
   conda create -n htp python=3.13
   ```

   (b) Activate your local Conda environment (e.g., ‘htp‘).

```
conda activate htp
```

(b) Install the required Python libraries into this environment using the provided requirements file. From the project's root directory, run:

```
pip install -r ml-api/requirements.txt
```

(c) **Mount the host's Conda directory into the container.** To allow the container to use your local machine's Conda environment, you must find the base path of your Miniconda/Anaconda installation and add it as a volume.

i. First, find the absolute path to your Conda installation by running the following command in your terminal:

```
conda info --base
```

This command will output a single line, for example: `/home/rohan/miniconda3`.

ii. Next, open the `docker-compose.yml` file. Under the `ml-api` service's `volumes` section, add a new line to mount the path you just found. The format should be `- /path/from/command:/path/from/command:ro`. For example:

```
    volumes:
      - ./ml-api:/app
      - /home/rohan/miniconda3:/home/rohan/miniconda3:ro
```

(d) **Update the Gunicorn executable path in the Dockerfile.** The final `CMD` instruction in the `ml-api/Dockerfile` contains a hardcoded path. To find the correct path for your machine, activate your Conda environment (`conda activate htp`) and run `which gunicorn`. Replace the original hardcoded path with the full path output by this command.

7.3 Running the Application

1. **Build and Start Containers:** From the root directory, run:

```
docker compose up --build
```

This command will build the Docker images for all services and start them.

2. **Seed the Database:** Open a **new terminal window** and run the seeder script to create the initial Admin, Uploader, and Assessor users.

```
docker compose exec backend node seeder.js
```

3. **Access Services:**

   - **Frontend Application:** http://localhost:3000

- **Backend API:** http://localhost:5001
- **ML Service API:** http://localhost:5002

7.4 Stopping the Application

To stop and remove all containers, press 'Ctrl+C' in the terminal where the services are running, and then execute:

```
docker compose down
```

8 Individual Contributions

The project's success was the result of a collaborative and dedicated team effort, with responsibilities distributed across all facets of the development lifecycle. Each team member took ownership of a core component, ensuring high-quality execution from concept to deployment.

Akmal: Backend Architecture & DevOps strategy.

- Architected the foundational Node.js/Express backend, establishing the service-oriented structure and core business logic for the platform.
- Implemented the asynchronous job queue system, a critical architectural component for ensuring a responsive user experience by offloading long-running ML tasks.
- Led the overall DevOps strategy, architecting the Docker Compose configuration, defining the inter-service network, and setting up the volume-mounting strategy for an efficient development workflow with hot-reloading.

Arihant: ML Research UI/UX Prototyping.

- Led the research and development of the core "Seer" (EfficientNet) model, including dataset sourcing, preprocessing, and setting up the PyTorch training pipeline.
- Conducted extensive experimentation to fine-tune the model, analyzing performance metrics like F1-score and validation loss to identify and address challenges such as overfitting.
- Co-led the initial UI/UX design phase, creating the foundational wireframes and high-fidelity mockups in Figma that guided the frontend development.

Rohan: Backend & Services Development.

- Enhanced the ML service by engineering and integrating a novel geometric analysis module to detect drawing size and placement, adding a critical new dimension to the automated analysis.
- Architected and implemented the core backend services, including the platform's entire security layer (JWT, RBAC), the complete set of Mongoose data models, and the controller logic for key features like the Admin dashboard and case management.
- Played a vital role in the design-to-development pipeline by co-managing the Figma design system and authoring significant portions of the final technical documentation.

Aktar: Frontend Development & DevOps strategy.

- Served as a key developer on the React application, tasked with translating the high-fidelity Figma designs into pixel-perfect, reusable, and functional components.
- Implemented the styling and CSS architecture for the entire platform, ensuring visual consistency and responsiveness across a range of devices.
- Co-managed the project's containerization, specifically authoring and maintaining the Dockerfiles for the 'frontend' and 'ml-api' services, including configuring the multi-stage build for the production frontend to ensure a lightweight final image.

Deepraj: RAG Pipelines Development.

- Architected and implemented the advanced "Interpreter" module, engineering the Retrieval Augmented Generation (RAG) pipeline to ground the LLM's output in verifiable clinical knowledge.
- Meticulously curated the clinical knowledge base that serves as the foundation for the RAG system, a critical step for ensuring the tool's accuracy and reliability.
- Designed the prompt engineering strategy for the Gemini model, which was crucial for transforming the raw outputs of the RAG system into coherent, traceable, and clinically relevant psychological reports.

Devansh: Frontend Architecture.

- Engineered the architectural foundation of the client-side application using React, establishing a scalable and maintainable component structure.
- Implemented the core frontend systems, including global state management with the Context API for authentication and the robust protected routing system to enforce role-based access controls.
- Directed the overall frontend development, ensuring seamless API integration with the backend and a cohesive user experience across all role-based dashboards.

9 Conclusion

The ML-Assisted HTP Interpretation Platform successfully demonstrates the potential of a hybrid AI architecture to augment psychological assessment of Tree Diagrams. The system's service-oriented design provides a scalable and maintainable foundation for future development. While the current "Seer" model faces challenges with overfitting due to dataset limitations—a common issue in specialized medical AI domains—the foundational architecture, particularly the RAG-enhanced "Interpreter," is conceptually sound and effectively mitigates the risks of ungrounded AI-generated text.

By focusing on critical areas for improvement—such as data acquisition, advanced vision model refinement, and iterative LLM prompt engineering—the platform's clinical utility can be significantly enhanced. The microservice design ensures that the system can evolve effectively as these crucial improvements are implemented, ultimately leading to a more objective, efficient, and reliable HTP interpretation process.