IITGnGPT: Revolutionizing Plagiarism Checkers

Tanish Yelgoe[1], Venkatakrishnan E[1], Vivek Raj[1], Praanshu Patel[1], Aeshaa Shah[1], Sharvari Mirge[1], and Harinarayan J[1]

[1]firstname.lastname@iitgn.ac.in, IIT Gandhinagar

Abstract

In this project, we aim to develop an AI detection system capable of classifying a given text into the following four categories:

- Human-Written
- Machine-Generated
- Machine Humanized
- Machine Polished

Additionally, our model will identify the source of the AI model used (Claude, ChatGPT, Gemini) and highlight the parts that contributed to the decision of classifying the text. We also plan to extend this framework to other languages. A demonstration of our work can be found at: Link. The code for the same can be found at GitHub. Also the dataset is uploaded on Hugging Face and can be found here.

Mentor Feedback

During our weekly meetings with our mentor, we had detailed discussions about both the model and the website we are building for the demo. After Assignment 2, the mentor suggested the following things:

1. Add a tag for the AI sources on the website, which we get from the second layer that detects sources.
2. Deploy the model on a server that has enough capacity to handle requests efficiently. This ensures that the system runs smoothly, even when processing complex queries or multiple users at the same time.
3. Collect data from two more AI sources and train the second layer on those sources too.
4. Changing the prompts that we used for data generation for grammatical coherence.
5. Uploading the dataset to hugging face.
6. Adding a plagiarism checking feature.

Baseline Implementations

Comparison of our work with the baseline implementations

For our work, baseline availability differs between the two tasks:

- **4-Class Classification:** Baseline implementations for this task are available. This is illustrated in Figure 1.

Detector	Prec	Recall	F1-macro	Acc
RoBERTa	94.79	94.63	94.65	94.62
DANN+RoBERTa	**96.30**	**95.54**	**96.06**	**95.24**

Figure 1: Baseline 4-class text classification task.

- **LLM Identification (LLaMA vs Mistral):** For the novelty of identifying which LLM generated a text, no prior baselines existed. Hence, we implemented our own models using:
  - Core ML algorithms: Logistic Regression, Random Forest, XGBoost, LightGBM
  - Transformer architectures: TinyBert, RoBERTa, DistilBERT

The **LLM DetectAIve** paper compares their models with ZeroGPT, GPTZero, and SaplingAI. However, these systems only support binary Human vs AI classification and not four-class detection or model-source prediction.

System	Classification Accuracy (%)
GPTZero	87.50
ZeroGPT	69.17
Sapling AI	88.33
LLM-DetectAIve	97.50

Table 1: Comparison of classification accuracies across popular systems.

Impact and Limitations

Our `IITGnGPT` system currently supports a wide variety of lightweight and large language models, including LLaMA, Mistral, Aya, and Falcon. The system also provides explicit attribution for each AI-generated response, clearly indicating which model produced the output. We are actively expanding support to additional frontier models such as ChatGPT and Claude. However, the baseline implementations we compare against do not offer such multi-model support. Our system, also supports plagiarism checks between the uploaded files of a particular project using Jaccard Similarity.

Some of our limitations:

- Second layer trained on limited model set(llama, mistral, aya and falcon).
- Models struggle with short inputs (<10 words)

Screenshots from the Website

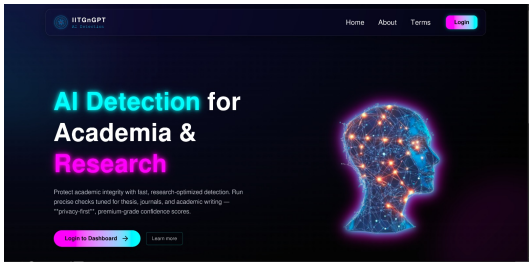

Figure 2: Landing Page

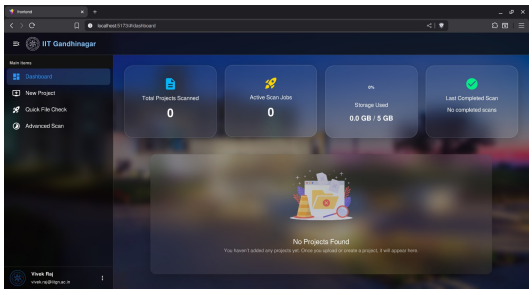

Figure 3: Home Page

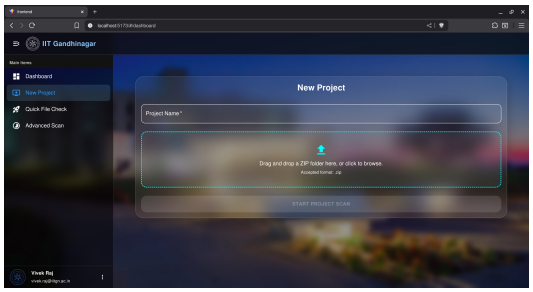

Figure 4: Upload Document

Figure 5: Result showing classes and model tags(mistral and llama)

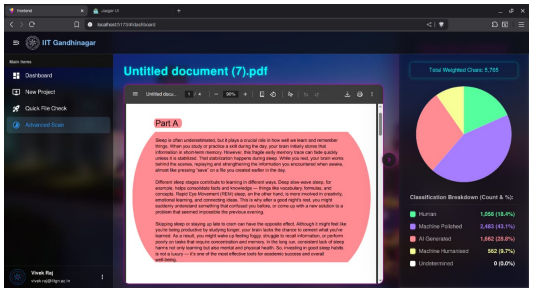

Figure 6: Detecting Machine Generated Text

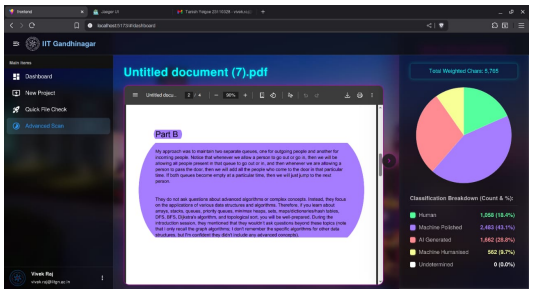

Figure 7: Detecting Machine Polished Text

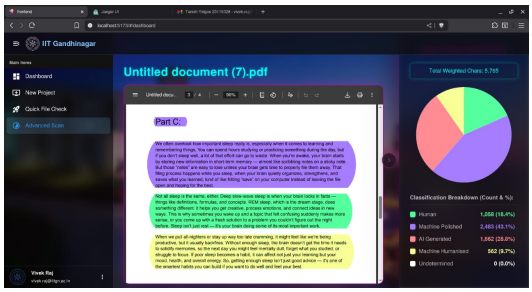

Figure 8: Detecting machine polished, human written and machine humanized text

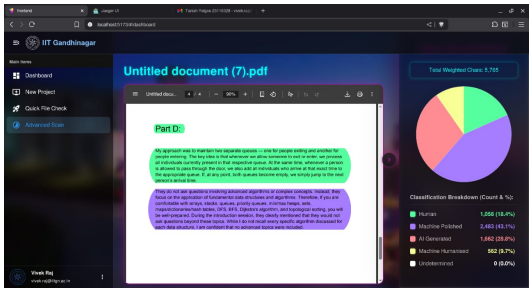

Figure 9: Detecting human written and machine polished text

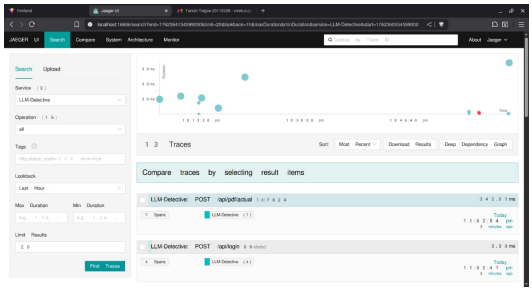

Figure 10: Jaeger traces

Dataset

We extended the dataset by using 2 new models *tiiuae/Falcon3-7B-Instruct* and *CohereLabs/aya-expanse-8b* for generation of the machine polished (hw_mp) and the machine written(wm) classes, see Table 2. The dataset is also uploaded on Hugging Face and can be found here.

Implementation Plan

Model and Website Pipeline

We applied for access to the Singularity server. We utilized Lightning AI accounts and Google Colab and ran some of the smaller models locally. We only had access to one account of the server, which was a significant challenge for us as it had limited resources.

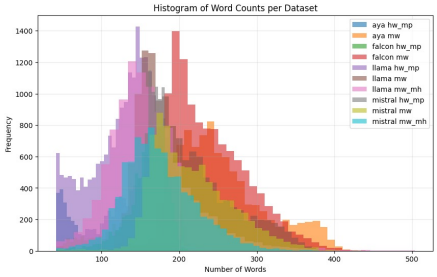

Figure 11: Histogram of Word Counts per class and model

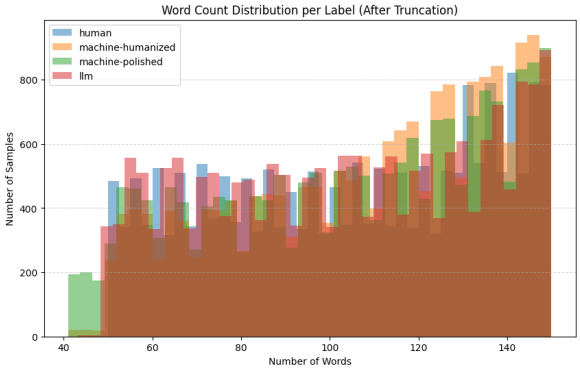

Figure 12: Histogram of Word Counts per class After Truncation

To ensure robustness, we balanced the dataset. We manually reviewed some of the samples to verify the accuracy of the collected data. We checkpointed the model training at various epochs and compared the metrics. We utilized proven models as our pre-trained weights, including BERT, TinyBERT, and Distil-BERT.

We tested the website's functionality with various samples collected from different sources to verify the initial layer. We also verified the second layer using data generated from specific sources (LLAMA and Mistral).

Models and Approaches Used:

You can find the model architecture in Figure 13 and the System architecture in 14.

For the first layer:

We fine-tuned multiple transformer-based language models for the 4-way classification task involving the following classes: *(i)* human-written (HW), *(ii)* machine-written (MW), *(iii)* machine-written machine-humanized (MW_MH), and *(iv)* human-written machine-polished (HW_MP). All models were trained using the training framework provided in the `LLM-DetectAIve` repository, with identical hyperparameters wherever possible to al-

Dataset	Number of Samples
human hw	22,000
aya hw_mp	9,821
aya mw	10,493
falcon hw_mp	9,504
falcon mw	11,881
llama hw_mp	21,385
llama mw	13,235
llama mw_mh	10,515
mistral hw_mp	11,044
mistral mw	8,820
mistral mw_mh	8,728

Table 2: Sample counts for each dataset category.

low for controlled comparison.

For the second layer:

- **Core ML Algorithms:**
  - **Logistic Regression:** Linear model for classification; uses feature weights instead of neural parameters.
  - **Random Forest:** Ensemble of decision trees with parameters such as number of trees and max depth.
  - **XGBoost / LightGBM:** Gradient-boosted decision trees with tunable depth, learning rate, and leaf count.
- **Transformer Models:**
  - **TinyBERT:** 4 encoder layers, hidden size 312, 12 attention heads, 14.5 M parameters.
  - **DistilBERT:** 6 encoder layers (vs 12 in BERT-base), same hidden size and vocab, 66 M parameters.
  - **RoBERTa-base:** 12 encoder layers , hidden size 768, 12 attention heads, around 125 M parameters.

Ensuring Differences Between Approaches:

- Core ML models rely on statistical features (e.g., TF–IDF), while transformers capture contextual semantics using self-attention.
- Thus, both groups differ fundamentally in architecture, learning paradigm, and parameter scale.

Parameters

Hyperparameter Optimization

We used **Randomized Search CV** instead of exhaustive Grid Search, as it is more computationally efficient while yielding comparable results. The Logistic Regression performance with and without the *type* feature is summarized in Table 3:

Metrics Used

The primary metric we used to evaluate performance was accuracy. The dataset was balanced across all classes, hence accuracy provided a reliable measure of overall model performance.

F1 score, precision and recall were the additional metrics we used. These metrics gave a more detailed view of the model's performance across different classes and helped confirm that the model performed uniformly across the classes.

System Effort

Hardware Requirements

- **CPU:** The backend performs tasks like PDF reading, editing, ZIP extraction, OCR processing, and managing socket communication. A capable CPU ensures these tasks run without delays.
- **GPU:** The AI model uses PyTorch and other libraries to process data and make predictions. A GPU allows the model to handle computations efficiently.
- **RAM:** Large PDFs and multiple user requests are processed in memory. Adequate RAM prevents slowdowns or crashes when handling several tasks at once.
- **Storage:** The system stores model weights, intermediate data, and uploaded files. Enough storage ensures that large files can be handled without errors. However we have limitted the Storage Bandwith of each user to be 5GB/Person
- **Network bandwidth:** Users will upload PDFs and the system communicates results in real-time. Reliable bandwidth keeps uploads and data transfers smooth.

Software Requirements

- **Linux System** Linux is recommended for reliability and stability. It's also the cheapest option available. Its also secure, uploading

Model	Best Params	Accuracy	Precision	Recall	F1-Score
Without Type Feature	{'C': 1.8443, 'solver': 'saga'}	0.8810	0.8843	0.8810	0.8821
With Type Feature	{'C': 1.8443, 'solver': 'saga'}	0.8881	0.8911	0.8881	0.8891

Table 3: Performance of Logistic Regression using Randomized Search CV with and without the *type* feature.

viruses to the system won't affect the system much.

- **Python 3.10 or above** Python is required to run the backend services, AI model inference, and scripts for PDF processing and data handling. Most our packages require Python version 3.10 or above
- **Docker** We are using Jeager UI for tracking the system's faults and errors. Installation of Jeager has recommened to use docker as its best option.
- **MongoDB** MongoDB stores metadata, user information, processed results, and configuration settings.
- **Nginx with SSL** Nginx acts as a reverse proxy to manage incoming requests and serve the frontend. SSL ensures secure communication between users and the server, protecting sensitive data like uploaded PDFs and model outputs.
- **Tesseract OCR** Tesseract is used to extract text from uploaded PDFs. This step is essential so the AI model can process the document content accurately.
- **Bash / any POSIX-compliant Shell** A standard shell is required to run start-up and maintenance scripts for backend, database, and model services. POSIX compliance ensures that scripts behave consistently across different Linux distributions.

Development Effort

Building this system required effort across several areas, and we approached it step by step. The software was developed and maintain.

- **Frontend Integration:** We built a glassamorpic themed UI using MUI and Toolpad integration, then we spent time making sure it connected smoothly with the backend. This included handling file uploads, displaying results, and making sure updates happen in real time.
- **Backend Development:** The backend is responsible for reading and editing PDFs, extracting text using Tesseract, handling ZIP files, and storing results in MongoDB. We focused on making these processes fast and reliable.
- **Plagiarism Checker:** We added a plagiarism checker based on the mentor's feedback. This plagiarism detection system analyzes PDF documents by extracting text (including OCR from embedded images) and comparing them using shingling and Jaccard similarity. It features persistent caching to avoid reprocessing documents.
- **AI Model Integration:** We integrated the AI model with the backend so it could classify documents. This involved sending data in manageable chunks, managing CPU and GPU resources, and making sure the model's predictions are delivered correctly to the frontend.
- **Server Setup and Deployment:** Setting up Docker containers for the Jeager UI, setting up Backend and Model, configuring Nginx with SSL, and preparing the environment for large PDF uploads was an important part of development. We also wrote startup scripts using a POSIX-compliant shell to make deployment easy. This is step is yet to be done.
- **Testing and Optimization:** We tested the system extensively to ensure everything works as expected. This included checking large file uploads, validating AI predictions, and making sure the results update correctly in the frontend. For structured testing, we built test suites using $UnitTests$ for Python backend components and $vitest$ for the React frontend.
- **Documentation and Maintenance:** Writing clear instructions for setup, usage, and deployment was an important part of the effort. This ensures anyone using or maintaining the system can understand how it works and keep

it running smoothly. This can be seen in the README file of the Github Repository.

Data Curation and Model Preparation:

We began by exploring existing datasets to gather initial training and testing data. Next, we used the Groq API to collect more data, but since the quota was quickly exhausted, we rotated multiple API keys to continue data collection. Later, we set up a Singularity server to run open-source large language models like LLama and Mistral for model inference. After running initial predictions, we cleaned the output by removing unnecessary text, such as filler phrases like "Ok, so here is...", to make the data more suitable for training and evaluation.

System Workflow

The system workflow describes how a user interacts with the platform and how the backend processes their requests:

1. **User Login:** The user starts by logging in using Google OAuth2. The system verifies their credentials to ensure secure access.

2. **Dashboard:** After logging in, the user lands on the dashboard. Here, they can see past uploads, check the status of ongoing tasks, and start new ones. The dashboard is the main hub for interacting with the system.

3. **Uploading PDF(s):** The user uploads either a single PDF or multiple PDFs bundled as a ZIP file. The files are securely sent to the backend along with authentication information.

4. **Preprocessing and OCR:** If the files are zipped, the system first extracts them. Then, Tesseract OCR processes each PDF to extract the text. This text is prepared for the AI models to analyze.

5. **AI Model Processing:** The extracted text is split into chunks suitable for model inference. Each chunk is processed by AI models like LLama or Mistral, which classify the content. The backend collects and organizes all the predictions.

6. **Highlighting and Annotation:** Based on the model predictions, the system highlights relevant portions directly in the PDF. This makes it easy for users to see AI-detected content at a glance.

7. **Viewing Results:** Once processing is complete, the results and highlighted PDFs are sent back to the frontend. Users can view, download, or interact further with the processed files from the dashboard.

8. **Monitoring:** Throughout the workflow, tools like OpenTelemetry and Jaeger keep track of performance and errors. This helps ensure smooth operation and quick identification of any issues.

Project Management

Novel Solutions

Based on literature survey, we emulated the first layer from the paper (LLM-DetectAIve) we focused on. We collected our own datasets and trained it from scratch.

For the second layer, we discussed with our mentor and based on the suggestions, we decided to classify the AI source by training on specific AI sources and integrate this into our flow.

We decided to make a community website at the suggestion of our mentor and host it for the campus. We have proposed novel solutions through literature survey, discussion with mentor and, trial and error.

Computational resources

We used the following resources from the singularity server that we got access to.

Resource	Specification
RAM Used	48 GB
GPU Used	NVIDIA L40S
Storage	500 GB

Table 4: Computational Resources from Server

We also used our personal laptops and cloud services for data collection and training. We would like to have access to more compute so that we can collect more data and improve the robustness of the second layer to identify even more AI sources.

Work division

We split the project as follows:
Tanish Yelgoe: data collection and layer-1 model training
Venkatakrishnan E: layer-2 model training
Vivek Raj: Website Development
Praanshu Patel: layer-2 model training
Aeshaa Shah1: layer-2 model training
Sharvari Mirge: layer-2 model training
Harinarayan J: layer-1 model training

Results

Total datasets Collected

We created datasets for all three categories other than human-written using the singularity server and running the LLMs on the server as shown in Table 2.

We used a balanced version of this dataset for our training.

Model Results and Ablation Tables

Model	Accuracy	F1 Score	Precision	Recall
Bert-Tiny	0.906	0.907	0.908	0.906

Table 5: 4-class classification

Model	Accuracy	Precision	Recall	F1-Score
Logistic Regression	0.8872	0.8903	0.8872	0.8882
Random Forest	0.8276	0.8335	0.8276	0.8295
XGBoost	0.8825	0.8865	0.8825	0.8838
LightGBM	0.8724	0.8764	0.8724	0.8738

Table 6: Model Performance Comparison

Model	Accuracy	Precision	Recall	F1-Score
Logistic Regression	0.8827	0.8827	0.8827	0.8827
Random Forest	0.8204	0.8206	0.8204	0.8204
XGBoost	0.8578	0.8578	0.8578	0.8578
LightGBM	0.8587	0.8587	0.8587	0.8587

Table 7: Model Performance Comparison (Without "type")

Model	Accuracy	Precision	Recall	F1-Score
DistilBERT	0.9265	0.9312	0.9315	0.9308
MobileBERT	0.9056	0.9129	0.9085	0.9105
MiniBERT	0.8886	0.8995	0.9015	0.8980
SmallBERT	0.9026	0.9101	0.9173	0.9106
TinyBERT	0.8459	0.8613	0.8494	0.8536

Table 8: Performance comparison of DistilBERT, MobileBERT, MiniBERT, SmallBERT, and TinyBERT models.

Metric	Value
Eval Loss	0.1622
Accuracy	0.9596
Macro F1-Score	0.9599
Runtime (s)	236.34

Table 9: DeBERTa Evaluation Summary

Acknowledgments

We acknowledge the support of Prof. Mayank Singh, our mentor Mr. Himanshu Beniwal and the CSE department of IIT Gandhinagar for giving us access to the Singularity Server.

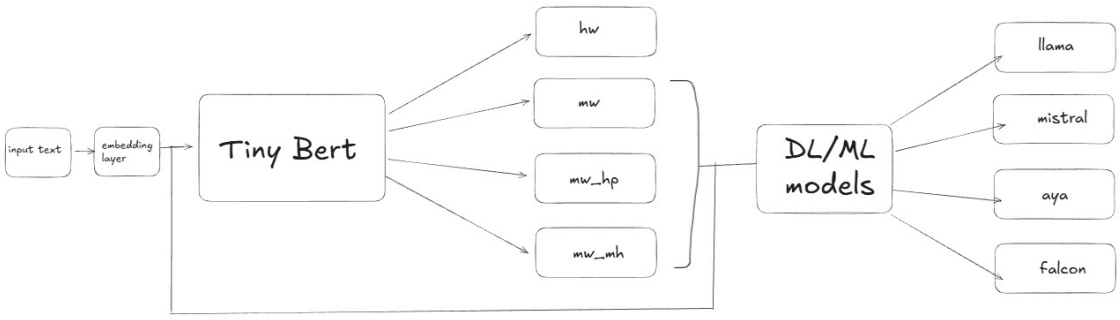

Figure 13: Overall Model Architecture

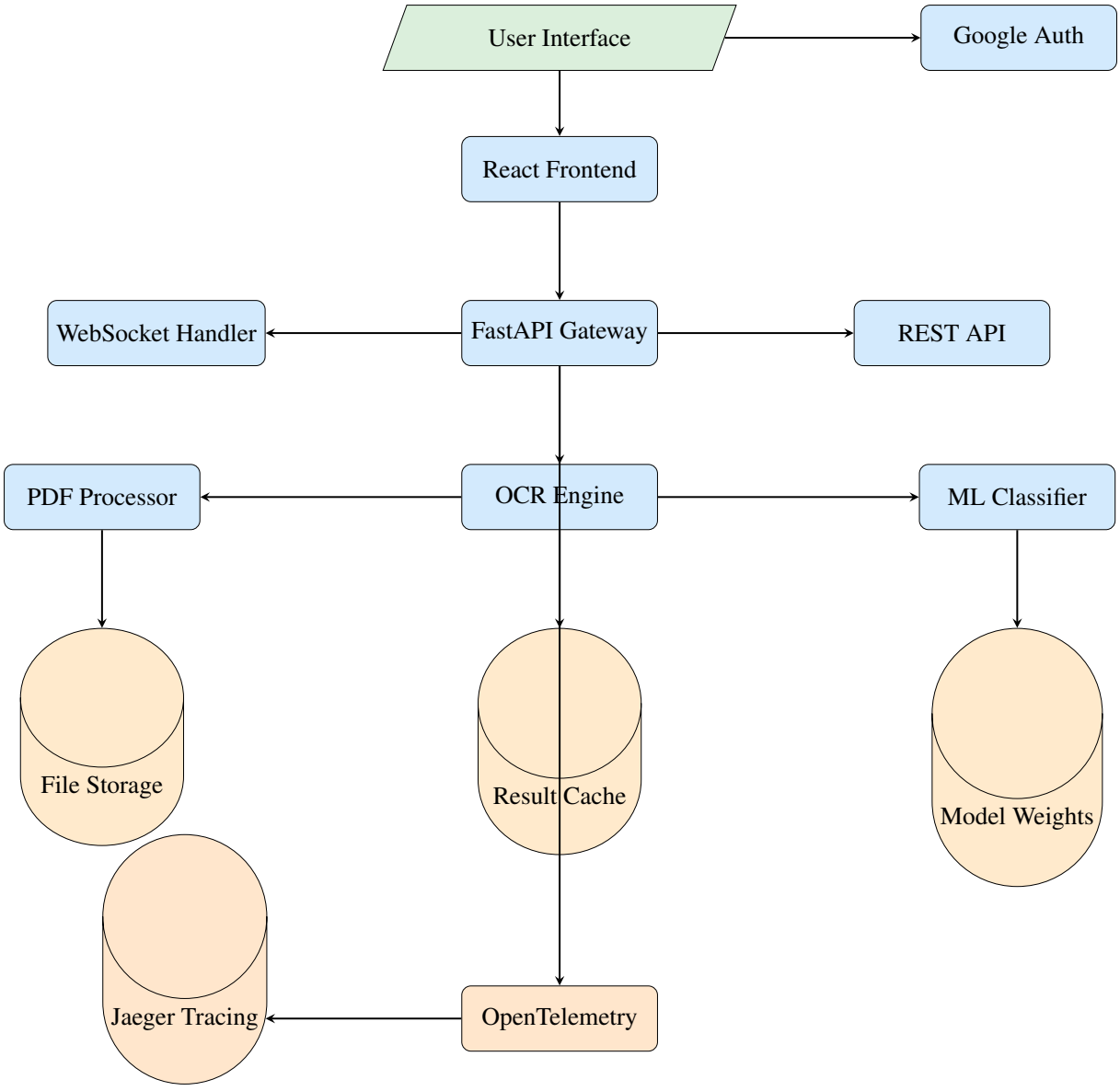

Figure 14: Architecture diagram