Perfect! You’ve shared both **backend** and **frontend** code. Let’s break it down in a way that’s easy to present: I’ll explain **each module/file**, connect it to your **dashboard UI**, and highlight its role in the project.

□ BACKEND (Python + FastAPI)

Your backend is responsible for:

- Accepting claim PDF/image uploads.
- Extracting text using OCR or PDF parsing.
- Checking claim details against hospital/disease data.
- Returning fraud detection results to the frontend.

---

1. main.py – API Entry Point

This is the **core backend server**.

🔹 Key Features:

- **FastAPI setup**: Creates the API app and allows CORS so the frontend can call it.
- **Dataset loaders**: Loads hospitals and diseases datasets (`Hospital_Dataset.xlsx`, `disease_treatment_dataset.xlsx`). If datasets are missing, it uses mock data.
- **Text Extraction**:
    - `extract_text_from_pdf`: Reads text from PDFs using **pdfplumber**.
    - `extract_text_from_image`: Uses **pytesseract OCR** for images.
- **Fraud Check**:
  `check_fraud()` checks if:
    - The hospital exists in the dataset.
    - The disease exists and matches the treatment.
    - Claim amount is suspicious (>5 lakh).
- **API Endpoint**:
  `/api/claims/upload`:
    - Accepts a file upload.
    - Extracts hospital, disease, treatment, amount, patient name, and claim ID using regex.
    - Runs `check_fraud` and returns fraud status + reason.

🎯 **How it connects to UI**:
The **ClaimUploader.tsx** frontend component sends uploaded files here. The API returns fraud status that’s displayed in the UI card.

2. datasets_loader.py

Loads and cleans:

- Hospital data
- Disease-treatment mapping
- Historical claims dataset

🎯 Supports fraud detection checks by providing a clean dataset for validation.

3. fraud_checker.py

This is your **brain of fraud detection**.

It:

- Flags missing critical information.
- Detects placeholder text like “enter hospital name”.
- Marks unknown hospitals/diseases as **fraudulent**.
- Flags mismatched treatment/disease combos.
- Detects suspicious behavior like:
  - High claim amounts
  - Emergency or risky procedures
  - Odd claim IDs
  - Round-number amounts
  - Placeholder names

🎯 **Impact on Dashboard**:
The fraud reasons from this file populate:

- **Fraud Reason** section in the ClaimUploader cards.
- **Fraud Type stats** in AnalyticsDashboard.

4. pdf_checker.py

- Provides reusable functions for **text extraction**:
  - Faster PDF text reading with `pdfplumber`.
  - OCR for scanned images.

🎯 Supports all backend fraud detection logic.

🎨 FRONTEND (React + TypeScript + Tailwind)

Your frontend is a **dashboard-style web app** with four main views.

1. ClaimUploader.tsx

The first interaction point for users.

🔹 Features:

- Drag & drop or click-to-upload PDF/image.
- Calls `/api/claims/upload` backend.
- Shows fraud detection status:
    - **Clean, Suspicious, Fraudulent** badges.
- Displays extracted info: claim ID, hospital, treatment, amount, etc.
- Shows fraud reason if detected.

🎯 **Direct link to backend**:
This is the only component that communicates with the backend API.

2. ClaimsTable.tsx

A searchable, filterable table of all claims.

🔹 Features:

- **Filters**: by claim status, region, and search.
- **Badges**: Color-coded fraud status indicators.
- **Details**: Patient name, hospital, location, fraud reason.
- **Action Button**: "View" for detailed claim review.

🎯 **Dashboard Role**:
Acts like an **admin panel** to monitor claim patterns.

---

3. AnalyticsDashboard.tsx

A visually rich dashboard summarizing fraud patterns.

🔹 Sections:

1. **Regional Fraud Analysis** – Fraud distribution by city with badges (color-coded by risk).
2. **Fraud Types** – Bar showing top fraud categories.
3. **Monthly Trends** – Detection accuracy, false positives, processing speed.
4. (Detailed Mode) **Hospital-wise Analysis & Risk Indicators**.

🎯 **Where data comes from**:
Right now, numbers are static, but in production, they'd come from backend analytics.

---

4. App.tsx

Sets up routing, providers, and app structure:

- Home route (`/`) → Index page (likely has ClaimUploader, Dashboard).
- Fallback `NotFound` route.

🔗 Flow of the Whole App

1. **User uploads a PDF/image** → ClaimUploader → Backend `/api/claims/upload`.
2. **Backend extracts text** (pdfplumber/ocr) → parses hospital, disease, etc.
3. **Fraud Checker runs validations** → returns `fraudStatus` + `fraudReason`.
4. **Frontend updates UI**:
    - ClaimUploader shows extraction results.
    - ClaimsTable shows claim with fraud badges.
5. **AnalyticsDashboard** visualizes overall trends.

🗣 Speech-Friendly Summary

“Our project is a full-stack insurance fraud detection system. The backend, built with FastAPI, receives uploaded claim forms, extracts text using OCR and PDF parsing, and checks the data against verified hospitals and diseases. It runs multiple rule-based fraud checks, such as unknown hospitals, invalid treatments, or unusually high amounts, and assigns each claim a status: clean, suspicious, or fraudulent.

The frontend, built in React with Tailwind CSS, provides three main dashboards: ClaimUploader, where users upload claim forms and see fraud analysis in real time; ClaimsTable, a searchable and filterable claims overview; and AnalyticsDashboard, which shows fraud patterns by region, hospital, and type.

This combination makes it easy for insurance teams to upload documents, get instant fraud analysis, and visualize fraud trends at a glance.”

---

main.py
Perfect, let’s go through your code **line by line** in plain, understandable terms. I’ll split this into logical sections so it’s easier to follow.

□ Line-by-Line Explanation of `main.py`

---

1. Imports

```
import shutil
import os
import re
import pandas as pd
from fastapi import FastAPI, UploadFile, File, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import JSONResponse
import pdfplumber
from PIL import Image
import pytesseract
```

- `shutil, os`: Handle file operations (saving & deleting temporary files).
- `re`: Regular expressions for text pattern matching.
- `pandas`: Reads and processes Excel datasets (hospitals, diseases).
- `FastAPI, UploadFile, File, HTTPException`: Build the API, accept uploaded files, and handle errors.

- `CORSMiddleware`: Lets your frontend call this API from a browser.
- `JSONResponse`: Return data in JSON format.
- `pdfplumber`: Extract text directly from PDF documents.
- `PIL.Image` & `pytesseract`: Load images and extract text (OCR).

---

2. FastAPI App Setup

```
app = FastAPI(title="Insurance Fraud Detection API", version="1.0.0")
```

Creates a FastAPI instance with a title and version.
This is your **backend application entry point**.

---

3. Allowing Frontend to Connect (CORS Setup)

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

- This middleware allows your **frontend (React app)** to talk to the API without restrictions.
- `allow_origins=["*"]`: Any domain can call your API (for testing/development).

---

4. Dataset Loaders

```
def load_hospitals():
    ...
```

- Tries to load `Hospital_Dataset.xlsx`.
- If file missing, **prints a warning** and uses mock hospital names.
- Cleans column names (`str.strip()` removes extra spaces).

```
def load_diseases():
    ...
```

- Similar to `load_hospitals()`, but loads a disease-treatment mapping Excel file.
- Also has mock data as a fallback.

🎯 **Purpose**: Your fraud detection logic will use these datasets to validate claims.

---

5. Fraud Checking Logic

```
def check_fraud(claim_data, hospitals_df, diseases_df):
```

- Accepts extracted claim data and both datasets.
- Extracts `hospital_name, disease, treatment, amount` from `claim_data`.

```
fraud_status = "clean"
fraud_reason = ""
```

- Default assumption: claim is clean unless flagged.

Fraud Checks Inside `check_fraud`

1. **Check if hospital is known**:

```
if hospital_name and hospital_name not in
hospitals_df['HospitalName'].values:
    fraud_status = "fraudulent"
    fraud_reason = "Hospital not in dataset"
    return fraud_status, fraud_reason
```

2. **Check if disease exists**:

```
disease_row = diseases_df[diseases_df['Disease'].str.lower() ==
disease.lower()]
```

- Looks for a matching disease in the dataset.
- If not found → Fraudulent.

3. **Check treatment matches disease**:

```
valid_treatments_str = disease_row.iloc[0]['Treatment']
valid_treatments = [t.strip().lower() for t in
valid_treatments_str.split(',')]
if treatment.lower() not in valid_treatments:
    fraud_status = "fraudulent"
    fraud_reason = "Treatment mismatch for disease"
```

4. **Suspicious amount check**:

```
if amount > 500000:
    fraud_status = "suspicious"
    fraud_reason = "Unusually high claim amount"
```

5. **Return clean if all checks pass**:

```
return "clean", "All checks passed."
```

---

6. Text Extraction Functions

```
def extract_text_from_pdf(file_path: str) -> str:
    with pdfplumber.open(file_path) as pdf:
        for page in pdf.pages:
            page_text = page.extract_text()
            if page_text:
                text += page_text + "\n"
```

- Reads PDF content **page by page**.
- Returns combined text.

```
def extract_text_from_image(file_path: str) -> str:
    img = Image.open(file_path)
    text = pytesseract.image_to_string(img)
```

- Opens an image and extracts text using OCR.

---

7. Load Datasets at Startup

```
try:
    hospitals_df = load_hospitals()
    diseases_df = load_diseases()
    print("Datasets loaded successfully")
```

- Pre-loads datasets when the server starts.

---

8. API Endpoints

Root Endpoint

```
@app.get("/")
async def root():
    return {"message": "Insurance Fraud Detection API", "status": "running"}
```

- Simple test route. Shows API is live.

Health Check

```
@app.get("/health")
```

```
async def health_check():
    return {"status": "healthy", "hospitals_count": len(hospitals_df),
"diseases_count": len(diseases_df)}
```

- Verifies if datasets are loaded properly.

Claim Upload & Analysis

```
@app.post("/api/claims/upload")
async def upload_claim(file: UploadFile = File(...)):
```

This is the **heart of your backend**.
Steps:

1. Save uploaded file temporarily.
2. Detect file type (PDF or image) → Extract text.
3. Parse text with regex:

```
hospital_match = re.search(r"Hospital(?:\s+Name)?:?\s*(.+)", text,
re.IGNORECASE | re.MULTILINE)
```

- Extracts `Hospital`, `Disease`, `Treatment`, `Amount`, `Patient Name`, `Claim ID`.

4. Build a `claim_data` dictionary:

```
claim_data = {
    "hospital": hospital,
    "disease": disease,
    "treatment": treatment,
    "amount": amount,
    "patientName": patient_name,
    "claimId": claim_id,
}
```

5. Pass it to `check_fraud()`:

```
fraud_status, fraud_reason = check_fraud(claim_data, hospitals_df,
diseases_df)
```

6. Return JSON to frontend:

```
return JSONResponse(content={"status": "success", "extractedData":
claim_data}, status_code=200)
```

7. Finally, delete the temp file.

---

9. Run the App

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="127.0.0.1", port=8000)
```

- Runs the FastAPI app locally at `http://127.0.0.1:8000`.

🗣️ How to Present This in Speech

“Our backend starts by importing Python libraries like pdfplumber for PDFs, pytesseract for OCR, and FastAPI for building the API. We first load hospital and disease datasets, or use mock data if files are missing. The `/api/claims/upload` endpoint accepts claim PDFs or images, extracts details like hospital, disease, treatment, and amount using regex, and then runs fraud checks. It flags claims as clean, suspicious, or fraudulent depending on dataset validation and business rules. Finally, the API returns a JSON response to our frontend.”

Would you like me to now do the **frontend file-by-file explanation** in the same detailed style?

dataset_loader.py

Great! Let’s break down your `dataset_loader.py` **line by line** so you can explain it clearly in your presentation.

◆ Code Analysis: `dataset_loader.py`

```
import pandas as pd
import os
```

- `pandas as pd`: Loads the **pandas library** (used for working with tabular data like Excel or CSV files).
- `os`: Used for handling file paths in an OS-independent way.

```
def load_hospitals():
    """
    Loads the hospital dataset and cleans up the column names.
    """
```

- Defines a function `load_hospitals()` to load the **hospital dataset**.

- The docstring (`""" ... """`) explains what this function does.

```
    file_path = os.path.join(os.path.dirname(__file__),
"../datasets/Hospital_Dataset.xlsx")
```

- Builds the **file path** to the hospital dataset:
    - `os.path.dirname(__file__)` → Gets the directory where this file (`dataset_loader.py`) is located.
    - `"../datasets/Hospital_Dataset.xlsx"` → Points to the Excel file inside a `datasets` folder, one level up.
    - `os.path.join(...)` → Joins these paths safely for all OSes (Windows, Mac, Linux).

```
    hospitals_df = pd.read_excel(file_path)
```

- Reads the Excel file into a **DataFrame** (table-like structure) using pandas.
- `hospitals_df` now holds all hospital data.

```
    # Clean up column names by stripping whitespace
    hospitals_df.columns = hospitals_df.columns.str.strip()
```

- Cleans all column headers (removes any leading/trailing spaces).
- This ensures consistent column names like `HospitalName` instead of `" HospitalName "`.

```
    return hospitals_df
```

- Returns the DataFrame so other files (like `main.py`) can use it for fraud detection.

```
def load_diseases():
    """
    Loads the disease-treatment dataset and cleans up the column names.
    """
```

- Similar structure as `load_hospitals()`.
- This function is dedicated to loading **disease-treatment mapping**.

```
    file_path = os.path.join(os.path.dirname(__file__),
"../datasets/disease_treatment_dataset.xlsx")
```

- Builds the path to the **disease-treatment dataset** Excel file.

```
    diseases_df = pd.read_excel(file_path)
```

- Reads the dataset into a pandas DataFrame (`diseases_df`).

```
    diseases_df.columns = diseases_df.columns.str.strip()
```

- Cleans all column headers.

```
    return diseases_df
```

- Returns this dataset for use in fraud checks.

```
def load_claims_history():
    """
    Loads the claims history dataset and cleans up the column names.
    """
```

- A third function for loading **historical claims data**.
- Useful for analyzing trends, fraud history, and future ML improvements.

```
    file_path = os.path.join(os.path.dirname(__file__),
"../datasets/Claims_Fraud_Dataset_with_Reasons.xlsx")
```

- Path to an Excel file storing **past claims and fraud reasons**.

```
    claims_df = pd.read_excel(file_path)
```

- Reads claims history into a DataFrame.

```
    claims_df.columns = claims_df.columns.str.strip()
```

- Cleans column names.

```
    return claims_df
```

- Returns claims history dataset.

📝 Summary of What This File Does:

- Contains **3 helper functions**:
    1. `load_hospitals()` → Loads hospital names from Excel.
    2. `load_diseases()` → Loads diseases and treatments.
    3. `load_claims_history()` → Loads historical claim and fraud data.
- Uses **pandas** to read Excel sheets.
- Uses **os.path** to make sure file paths work on all systems.
- Cleans column names to avoid mismatches during fraud checks.

🗣️ How to Explain in Presentation:

"This file is a utility script to load datasets needed for fraud detection. It uses pandas to read Excel files for hospital names, disease-treatment mappings, and historical claim data. By stripping unnecessary spaces from column names, it ensures consistency. This makes our fraud detection checks accurate and reliable, as all other scripts can call these functions to get clean data."

Would you like me to **next explain `fraud_checker.py` line by line**? This is where all your fraud detection rules live.

fraud_checker.py
Perfect! Let's go through your `fraud_checker.py` code **line by line**. This file is the "brain" of your fraud detection logic, so explaining it well will make your presentation stronger.

◆ Code Walkthrough: `fraud_checker.py`

Imports

```
import re
```

- Imports **regular expressions** (`re`) to search for text patterns in extracted claim data. Example: Detecting placeholder text like `"Enter Hospital Name"`.

Function Definition

```
def check_fraud(claim_data, hospitals_df, diseases_df):
    """
    Checks for potential fraud based on hospital, disease, and treatment
data.
    Returns: clean, suspicious, or fraudulent
    """
```

- Defines `check_fraud`, which:
    - Takes `claim_data` (a dictionary with hospital, disease, treatment, amount, patient name, claim ID).
    - Uses `hospitals_df` and `diseases_df` datasets for validation.
    - Returns:
        - `"fraudulent"` (serious fraud),
        - `"suspicious"` (requires review), or
        - `"clean"` (valid claim).

Extracting Claim Details

```
    hospital_name = claim_data.get("hospital", "").strip()
    disease = claim_data.get("disease", "").strip()
    treatment = claim_data.get("treatment", "").strip()
    amount = claim_data.get("amount", 0)
    patient_name = claim_data.get("patientName", "").strip()
    claim_id = claim_data.get("claimId", "").strip()
```

- Safely extracts claim details using `.get()`:
    - If a field is missing, defaults to `""` (empty string) or `0` for amount.
- `.strip()` removes extra spaces.

Debugging Output

```
    print("\n--- Checking for Fraud ---")
    print(f"Hospital Name Extracted: '{hospital_name}'")
    print(f"Disease Extracted: '{disease}'")
    print(f"Treatment Extracted: '{treatment}'")
    print(f"Amount Extracted: {amount}")
    print(f"Patient Name: '{patient_name}'")
    print(f"Claim ID: '{claim_id}'")
```

- Prints extracted claim details to the console for debugging.

Initialize Fraud Status

```
fraud_status = "clean"
fraud_reason = ""
```

- Default assumption: Claim is valid until proven fraudulent or suspicious.

● FRAUDULENT CHECKS (Severe Issues)

1. Missing Critical Information

```
if not hospital_name or not disease or not treatment or not patient_name:
    fraud_status = "fraudulent"
    fraud_reason = "Missing critical claim information"
    print(f"Fraud Detected: {fraud_reason}")
    return fraud_status, fraud_reason
```

- If **any of these are missing**, the claim is marked fraudulent immediately.

2. Detect Placeholder/Template Text

```
    placeholder_patterns = [
        r"enter\s+\w+\s+name", r"enter\s+claimed\s+amount",
r"provide\s+any\s+extra",
        r"patient\s+name", r"hospital\s+name", r"claim\s+amount"
    ]
```

- Defines regex patterns for common placeholder text seen in fake claims.

```
    combined_text = f"{hospital_name} {disease} {treatment}
{patient_name}".lower()
    for pattern in placeholder_patterns:
        if re.search(pattern, combined_text):
            fraud_status = "fraudulent"
            fraud_reason = "Document contains template/placeholder text"
            print(f"Fraud Detected: {fraud_reason}")
            return fraud_status, fraud_reason
```

- Combines all key text fields, converts to lowercase, and searches for placeholders.
- If found, flags claim as fraudulent.

3. Unknown Hospital Check

```
    if hospital_name not in hospitals_df['HospitalName'].values:
        fraud_status = "fraudulent"
        fraud_reason = "Hospital not in dataset"
        print(f"Fraud Detected: {fraud_reason}")
        return fraud_status, fraud_reason
```

- Flags if the hospital is not in the known dataset.

---

4. Unknown Disease Check

```
    disease_row = diseases_df[diseases_df['Disease'].str.lower() ==
disease.lower()]
    if disease_row.empty:
        fraud_status = "fraudulent"
        fraud_reason = "Disease not in dataset"
        print(f"Fraud Detected: {fraud_reason}")
        return fraud_status, fraud_reason
```

- Checks if disease is listed in dataset. If not, marks fraudulent.

---

5. Treatment-Disease Mismatch

```
    valid_treatments_str = disease_row.iloc[0]['Treatment']
    valid_treatments = [t.strip().lower() for t in
valid_treatments_str.split(',')]

    if treatment.lower() not in valid_treatments:
        fraud_status = "fraudulent"
        fraud_reason = "Treatment mismatch for disease"
        print(f"Fraud Detected: {fraud_reason}")
        return fraud_status, fraud_reason
```

- Retrieves valid treatments for the disease.
- If claim's treatment isn't valid, marks fraudulent.

▯ SUSPICIOUS CHECKS (Needs Review)

---

6. High Amount Check

```
    if amount > 100000:
        fraud_status = "suspicious"
        fraud_reason = "High claim amount requires additional verification"
        print(f"Suspicious Activity: {fraud_reason}")
        return fraud_status, fraud_reason
```

- Claims over ₹1,00,000 flagged as suspicious.

---

7. Emergency Treatment Check

```
    emergency_keywords = ['emergency', 'urgent', 'critical', 'immediate',
'trauma']
    if any(keyword in treatment.lower() for keyword in emergency_keywords):
        fraud_status = "suspicious"
        fraud_reason = "Emergency treatment requires additional review"
        print(f"Suspicious Activity: {fraud_reason}")
        return fraud_status, fraud_reason
```

- Flags if treatment mentions urgent/emergency words.

---

8. Expensive Treatment for Common Illness

```
    expensive_treatments = ['surgery', 'operation', 'transplant', 'bypass',
'angioplasty']
    common_diseases = ['fever', 'cold', 'headache', 'cough']

    if (any(exp_treatment in treatment.lower() for exp_treatment in
expensive_treatments) and
        any(common_disease in disease.lower() for common_disease in
common_diseases)):
        fraud_status = "suspicious"
        fraud_reason = "Expensive treatment for common condition requires
review"
        print(f"Suspicious Activity: {fraud_reason}")
        return fraud_status, fraud_reason
```

- Example: Surgery claimed for a simple cold → suspicious.

---

9. Irregular Claim ID

```
    if len(claim_id) < 5:
        fraud_status = "suspicious"
        fraud_reason = "Claim ID format appears irregular"
        print(f"Suspicious Activity: {fraud_reason}")
        return fraud_status, fraud_reason
```

- Flags very short or fake claim IDs.

---

10. Round-Number Claim Amounts

```
    if amount > 0 and amount % 10000 == 0 and amount > 50000:
        fraud_status = "suspicious"
        fraud_reason = "Suspicious round-number claim amount"
        print(f"Suspicious Activity: {fraud_reason}")
        return fraud_status, fraud_reason
```

- Claims with perfectly round numbers (₹50,000, ₹1,00,000) are flagged.

---

11. Fake Names Check

```
    common_fake_names = ['john doe', 'jane doe', 'test user', 'sample
patient', 'demo patient']
    if patient_name.lower() in common_fake_names:
        fraud_status = "suspicious"
        fraud_reason = "Patient name appears to be placeholder or generic"
        print(f"Suspicious Activity: {fraud_reason}")
        return fraud_status, fraud_reason
```

- Flags fake or test names.

✓ CLEAN CLAIM

```
    print("No fraud detected. Claim is clean.")
    return "clean", "All checks passed - claim appears legitimate"
```

- If none of the above checks trigger, claim is **clean**.

🗣 □ How to Present This

“This file defines the fraud detection engine. It applies over a dozen checks to decide if a claim is fraudulent, suspicious, or clean.

We first validate critical details: if any key fields like hospital or patient name are missing, or contain placeholder text, we flag it as fraudulent. Next, we verify if the hospital and disease exist in our datasets, and whether the treatment matches the disease.

We then run suspicious checks, like unusually high amounts, emergency procedures, expensive treatments for minor illnesses, fake names, or round-number amounts.

Finally, if all checks pass, the claim is marked clean. This rule-based engine ensures fast, explainable fraud detection."

Would you like me to **do the same line-by-line breakdown for `pdf_checker.py` next**?

pdf _reader.py
Perfect! Let's break down your `pdf_reader.py` file **line by line**. This file is a **utility module** for extracting text from PDFs and images — it's a core step before running fraud detection.

◆ Code Walkthrough: `pdf_reader.py`

Imports

```
import pdfplumber
from PIL import Image
import pytesseract
```

- `pdfplumber`: Library for **extracting text directly from PDFs** (works well for PDFs that already contain text, not just scanned images).
- `PIL.Image`: Part of the **Pillow** library; used to open and process images.
- `pytesseract`: Python wrapper for **Tesseract OCR**, used to read text from images.

Function: extract_text_from_pdf()

```
def extract_text_from_pdf(file_path: str) -> str:
    """
    Extracts text from a PDF file using pdfplumber.
    This is much faster and more accurate than OCR for native PDFs.
    """
```

- Defines a function that:
    - Takes a `file_path` (string) of a PDF file.
    - Returns extracted text as a **string**.
- The docstring explains that **pdfplumber is preferred for PDFs that contain text layers** because it's faster than running OCR.

```
    try:
        text = ""
```

- Starts a **try block** for error handling.
- Initializes `text` as an empty string to store extracted text.

```
        with pdfplumber.open(file_path) as pdf:
```

- Opens the PDF file using `pdfplumber`.
- `with` ensures the file is properly closed after processing.

```
            for page in pdf.pages:
                page_text = page.extract_text()
                if page_text:
                    text += page_text + "\n"
```

- Loops through **each page** in the PDF.
- Uses `extract_text()` to get text from the current page.
- If text exists, appends it to the `text` variable, adding a line break (`\n`) after each page.

```
        return text.strip()
```

- Returns all extracted text, removing unnecessary leading/trailing whitespace.

```
    except Exception as e:
        print(f"Error extracting text from PDF: {e}")
        return ""
```

- If there's an error (e.g., PDF is encrypted or corrupted):
  - Prints the error message.
  - Returns an empty string instead of crashing the app.

Function: extract_text_from_image()

```
def extract_text_from_image(file_path: str) -> str:
    """
    Extracts text from an image using pytesseract OCR.
    """
```

- This function extracts text from an image file instead of a PDF.
- Uses OCR for scanned or image-only documents.

```
    try:
        img = Image.open(file_path)
```

- Opens the image file using Pillow (`Image.open()`).

```
        text = pytesseract.image_to_string(img)
```

- Runs **Tesseract OCR** on the opened image.
- `image_to_string()` converts detected text into a Python string.

```
        return text.strip()
```

- Returns extracted text with leading/trailing spaces removed.

```
    except Exception as e:
        print(f"Error extracting text from image: {e}")
        return ""
```

- If OCR fails (e.g., image unreadable), prints error and returns an empty string.

📝 Summary of `pdf_reader.py`

1. This file provides **two helper functions**:
    - `extract_text_from_pdf()` → Uses `pdfplumber` to quickly extract text from PDFs that contain digital text layers.
    - `extract_text_from_image()` → Uses `pytesseract` OCR to detect text from image files or scanned documents.
2. Both functions handle errors gracefully, ensuring the system doesn't crash on bad input.
3. These functions are used in `main.py` to process uploaded claim documents before fraud detection.

🗣 How to Present This

"Our `pdf_reader.py` module handles text extraction. For PDFs that already contain text, we use the `pdfplumber` library, which is much faster and more accurate than OCR. For image-based claim forms or scanned PDFs, we use Tesseract OCR via Pillow. Both functions are wrapped in error handling, ensuring our application doesn't fail if a file is unreadable. This modular design makes document parsing reliable and efficient."

Would you like me to now **start explaining the frontend code** (like `ClaimUploader.tsx`) in the same detailed style so you can cover both backend and UI?

Frontend

`analyticsDashboard.tsx`

Great! Let's break down `analyticsDashboard.tsx` **line by line**. This file is a **React + TypeScript component** that creates your entire analytics dashboard with fraud statistics, risk indicators, and hospital/region breakdowns.

◆ File: `analyticsDashboard.tsx`

Imports

```
import { Card, CardContent, CardDescription, CardHeader, CardTitle } from
"@/components/ui/card";
import { Progress } from "@/components/ui/progress";
import { Badge } from "@/components/ui/badge";
import { BarChart3, PieChart, MapPin, TrendingDown, AlertTriangle } from
"lucide-react";
```

- Imports **UI components**:
    - `Card`, `CardHeader`, `CardContent`, etc. → Reusable styled containers for dashboard cards.
    - `Progress` → Progress bar component for showing percentages visually.
    - `Badge` → Small labels (color-coded status indicators).
- `lucide-react` → Icon library (BarChart3, PieChart, MapPin, etc.).

Props Interface

```
interface AnalyticsDashboardProps {
  detailed?: boolean;
}
```

- Defines the props for this component.
- `detailed?: boolean;` → Optional prop to toggle detailed analytics (hospital analysis and risk indicators).

Component Definition

```
export const AnalyticsDashboard = ({ detailed = false }:
AnalyticsDashboardProps) => {
```

- Exports a functional React component.
- Uses **destructuring** to set a default value: `detailed = false`.

Static Data

```
  const regionData = [
    { region: 'Delhi', total: 847, fraudulent: 38, percentage: 4.5 },
    { region: 'Mumbai', total: 923, fraudulent: 52, percentage: 5.6 },
    { region: 'Bangalore', total: 654, fraudulent: 21, percentage: 3.2 },
    { region: 'Chennai', total: 423, fraudulent: 27, percentage: 6.4 },
  ];
```

- **regionData**: Array of regions with:
  - `total` claims, `fraudulent` claims, and fraud percentage.

```
  const hospitalData = [
    { hospital: 'Apollo Hospital', claims: 234, fraudulent: 12, amount:
'₹2.4M' },
    { hospital: 'Max Healthcare', claims: 189, fraudulent: 18, amount:
'₹3.1M' },
    { hospital: 'Fortis Hospital', claims: 167, fraudulent: 8, amount:
'₹1.8M' },
    { hospital: 'AIIMS', claims: 145, fraudulent: 3, amount: '₹0.8M' },
    { hospital: 'Unknown Clinics', claims: 89, fraudulent: 67, amount:
'₹8.9M' },
  ];
```

- **hospitalData**: Array showing fraud stats for each hospital.

```
const fraudTypes = [
  { type: 'Overbilling', count: 45, percentage: 35.4 },
  { type: 'Fake Hospital', count: 32, percentage: 25.2 },
  { type: 'Duplicate Claims', count: 28, percentage: 22.0 },
  { type: 'Document Forgery', count: 15, percentage: 11.8 },
  { type: 'Other', count: 7, percentage: 5.5 },
];
```

- **fraudTypes**: Array categorizing fraud types and their percentages.

Return JSX

```
return (
  <div className="space-y-6">
```

- Component returns JSX.
- `space-y-6` (TailwindCSS): Adds vertical spacing between child elements.

🔹 QUICK ANALYTICS SECTION

```
<div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-6">
```

- Sets up a **responsive grid**:
  - 1 column on small screens
  - 2 on medium
  - 3 on large screens.

1. Regional Fraud Analysis Card

```
<Card className="bg-gradient-to-br from-card to-card/50 shadow-lg">
```

- A `Card` component with gradient background and shadow.

```
<CardHeader>
  <CardTitle className="flex items-center text-lg">
    <MapPin className="w-5 h-5 mr-2 text-danger" />
    Regional Fraud Analysis
  </CardTitle>
  <CardDescription>Fraud distribution across regions</CardDescription>
</CardHeader>
```

- Title with a **MapPin** icon and subtitle.

```
<CardContent className="space-y-4">
  {regionData.map((region) => (
```

- Loops through `regionData` to display region stats dynamically.

```
  <div key={region.region} className="space-y-2">
    <div className="flex justify-between items-center">
      <span className="font-medium">{region.region}</span>
      <Badge
        className={
          region.percentage > 5
            ? "bg-danger text-danger-foreground"
            : region.percentage > 3
            ? "bg-warning text-warning-foreground"
            : "bg-success text-success-foreground"
        }
      >
        {region.percentage}%
      </Badge>
    </div>
```

- Displays:
  - Region name.
  - Percentage badge with conditional color coding:
    - Red if >5%, yellow if >3%, green otherwise.

```
    <Progress value={region.percentage} className="h-2" />
    <div className="text-xs text-muted-foreground">
      {region.fraudulent} of {region.total} claims
    </div>
  </div>
```

- Shows a **progress bar** for fraud percentage and claim counts.

---

2. Fraud Types Card

```
<Card className="bg-gradient-to-br from-card to-card/50 shadow-lg">
...
  {fraudTypes.map((type) => (
    <div key={type.type} className="flex justify-between items-center">
```

- Loops through `fraudTypes`.
- Displays fraud categories with a small colored dot and stats.

```
<div
  className="w-3 h-3 rounded-full"
  style={{
    backgroundColor: type.type === 'Overbilling' ? 'hsl(var(--danger))' :
                     type.type === 'Fake Hospital' ? 'hsl(var(--warning))' :
                     type.type === 'Duplicate Claims' ? 'hsl(var(--info))' :
                     'hsl(var(--muted-foreground))'
  }}
/>
```

- Uses inline styles to dynamically color fraud type indicators.

---

3. Monthly Trends Card

```
<Card className="bg-gradient-to-br from-card to-card/50 shadow-lg">
...
<Progress value={94.2} className="h-2" />
```

- Shows detection accuracy, false positives, and processing speed.
- Progress bars visualize performance.

```
<div className="pt-2 border-t border-border">
  <div className="text-xs text-muted-foreground">
    ↗️ 12% improvement this month
  </div>
</div>
```

- A bottom note showing month-over-month improvement.

DETAILED ANALYTICS SECTION (Optional)

```
{detailed && (
  <div className="grid grid-cols-1 lg:grid-cols-2 gap-6">
```

- This section **only renders** if `detailed={true}` is passed as a prop.
- Displays **Hospital Analysis** and **Risk Indicators**.

Hospital-wise Analysis Card

```
{hospitalData.map((hospital) => (
  <div key={hospital.hospital} className="p-3 bg-muted/30 rounded-lg">
```

- Loops through hospital data to show:
  - Name, total claims, fraud count, and amount at risk.
  - Risk level (High/Medium/Low) based on fraud ratio.

Risk Indicators Card

```
<div className="p-3 bg-danger/10 rounded-lg border border-danger/20">
  <div className="text-2xl font-bold text-danger">127</div>
  <div className="text-sm text-danger">Active Alerts</div>
</div>
```

- Shows alert counts and risk notifications with **color-coded severity levels**:
  - Critical, Medium, Low alerts.

End of Component

```
    </div>
  );
};
```

- Closes the component.

📝 Summary of `analyticsDashboard.tsx`

- This file creates a **fraud analytics dashboard UI** with:
  1. **Quick stats**: Region-wise fraud heatmap, fraud type distribution, monthly trends.
  2. **Detailed stats (optional)**: Hospital-level fraud breakdown and risk alerts.
- Uses:
  - TailwindCSS for responsive styling.
  - Icons from `lucide-react`.
  - `Card`, `Badge`, `Progress` components for reusable UI patterns.
- Data is currently **static** but can be connected to backend APIs later.

🗣 How to Present This

"Our `AnalyticsDashboard.tsx` file is a React component that visually summarizes fraud detection data. It has two sections: a quick overview showing region-wise fraud, fraud types, and monthly improvements; and a detailed section, enabled with a `detailed` prop, that breaks down fraud trends by hospital and displays key risk alerts. We use reusable components like cards, badges, and progress bars, along with TailwindCSS for responsive styling. This makes the dashboard dynamic and scalable for real-time data integration."

Would you like me to now **explain ClaimUploader.tsx** (the upload page) next? That's where the frontend talks directly to your backend API.

ClaimsTable.tsx

Great! Let's carefully break down `ClaimsTable.tsx` **line by line** so you can clearly present how this table works in your UI.

This file is your **Claims Overview Dashboard**, showing a searchable and filterable table of all claims, with fraud statuses, reasons, and quick actions.

◆ File: `ClaimsTable.tsx`

Imports

```
import { useState } from 'react';
import { Button } from "@/components/ui/button";
import { Input } from "@/components/ui/input";
import { Select, SelectContent, SelectItem, SelectTrigger, SelectValue } from
"@/components/ui/select";
import { Badge } from "@/components/ui/badge";
import { Table, TableBody, TableCell, TableHead, TableHeader, TableRow } from
"@/components/ui/table";
import { Card, CardContent, CardHeader, CardTitle } from
"@/components/ui/card";
import { Search, Filter, Eye, AlertTriangle, CheckCircle, Clock } from
"lucide-react";
```

- `useState`: React hook for managing local state (filters, search, etc.).
- `Button`, `Input`, `Select`, `Badge`, `Table`, `Card`: UI components from your design system.
- `lucide-react`: Icon library (Search, Filter, Eye, AlertTriangle, etc.) for visual indicators.

Claim Interface

```
interface Claim {
  id: string;
  patientName: string;
  hospital: string;
  amount: string;
  date: string;
  fraudStatus: 'clean' | 'suspicious' | 'fraudulent' | 'pending';
  fraudReason?: string;
  location: string;
  claimType: string;
}
```

- Defines the structure of a claim object.
- `fraudStatus`: limited to 4 values.
- `fraudReason?`: Optional (only appears if there’s an explanation).
- This helps with TypeScript validation.

Mock Data

```
const mockClaims: Claim[] = [
  { id: 'CLM001', patientName: 'Rajesh Kumar', hospital: 'Apollo Hospital,
Delhi', amount: '₹45,000', date: '2024-01-15', fraudStatus: 'clean',
location: 'Delhi', claimType: 'Surgery' },
  ...
];
```

- Hardcoded sample claim records.
- Each object represents a claim row (with fraud status and reasons).
- This simulates real API data for demo purposes.

Component Definition

```
export const ClaimsTable = () => {
```

- Defines a React functional component named `ClaimsTable`.

State Variables

```
const [claims, setClaims] = useState<Claim[]>(mockClaims);
const [searchTerm, setSearchTerm] = useState('');
const [statusFilter, setStatusFilter] = useState<string>('all');
const [locationFilter, setLocationFilter] = useState<string>('all');
```

- `claims`: Stores all claims (initially mock data).
- `searchTerm`: Stores search input text.
- `statusFilter`: Stores the selected fraud status filter.
- `locationFilter`: Stores the selected location filter.

Filtering Logic

```
const filteredClaims = claims.filter(claim => {
  const matchesSearch =
    claim.patientName.toLowerCase().includes(searchTerm.toLowerCase()) ||
    claim.hospital.toLowerCase().includes(searchTerm.toLowerCase()) ||
    claim.id.toLowerCase().includes(searchTerm.toLowerCase());

  const matchesStatus = statusFilter === 'all' || claim.fraudStatus ===
statusFilter;
  const matchesLocation = locationFilter === 'all' || claim.location ===
locationFilter;

  return matchesSearch && matchesStatus && matchesLocation;
});
```

- Filters claims dynamically:

- **Search**: Matches `patientName`, `hospital`, or `id`.
    - **Status Filter**: Matches fraud status or shows all.
    - **Location Filter**: Matches selected location or shows all.
- Returns only matching claims for rendering.

Status Badge Helper

```
const getStatusBadge = (status: string) => {
  switch (status) {
    case 'clean':
      return <Badge className="bg-success text-success-
foreground"><CheckCircle className="w-3 h-3 mr-1" />Clean</Badge>;
    case 'suspicious':
      return <Badge className="bg-warning text-warning-
foreground"><AlertTriangle className="w-3 h-3 mr-1" />Suspicious</Badge>;
    case 'fraudulent':
      return <Badge className="bg-danger text-danger-
foreground"><AlertTriangle className="w-3 h-3 mr-1" />Fraudulent</Badge>;
    case 'pending':
      return <Badge variant="secondary"><Clock className="w-3 h-3 mr-1"
/>Pending</Badge>;
    default:
      return <Badge variant="secondary">{status}</Badge>;
  }
};
```

- Returns a **color-coded badge** based on claim status:
    - Green for clean, yellow for suspicious, red for fraudulent, gray for pending.
- Uses icons to visually reinforce meaning.

Amount Color Helper

```
const getAmountColor = (status: string) => {
  switch (status) {
    case 'fraudulent':
      return 'text-danger font-bold';
    case 'suspicious':
      return 'text-warning font-semibold';
    default:
      return 'text-foreground';
  }
};
```

- Dynamically styles claim amounts:
    - Fraudulent → Bold red.

  - Suspicious → Yellow.
  - Clean → Default.

Component Return JSX

```
return (
  <div className="space-y-6">
```

- Returns the full table UI.
- `space-y-6`: Adds vertical spacing between cards.

Filters Section

```
<Card className="bg-muted/20">
  <CardHeader>
    <CardTitle className="text-lg flex items-center">
      <Filter className="w-5 h-5 mr-2" />
      Filters & Search
    </CardTitle>
  </CardHeader>
```

- First card contains search bar and filter dropdowns.

```
<Input
  placeholder="Search claims..."
  value={searchTerm}
  onChange={(e) => setSearchTerm(e.target.value)}
  className="pl-10"
/>
```

- Text input for searching claims.
- Updates `searchTerm` dynamically.

```
<Select value={statusFilter} onValueChange={setStatusFilter}>
```

- Dropdown filter for fraud status.
- Changes `statusFilter` state on selection.

```
<Select value={locationFilter} onValueChange={setLocationFilter}>
```

- Dropdown filter for locations.

```
<Button variant="outline" onClick={() => {
  setSearchTerm('');
  setStatusFilter('all');
  setLocationFilter('all');
}}>
  Clear Filters
</Button>
```

- Resets all filters when clicked.

Claims Table

```
<Card>
  <CardHeader>
    <CardTitle className="text-lg">
      Claims Overview ({filteredClaims.length} results)
    </CardTitle>
  </CardHeader>
```

- Displays total count of filtered claims dynamically.

```
<Table>
  <TableHeader>
    <TableRow>
      <TableHead>Claim ID</TableHead>
      <TableHead>Patient Name</TableHead>
      <TableHead>Hospital</TableHead>
      <TableHead>Amount</TableHead>
      <TableHead>Date</TableHead>
      <TableHead>Status</TableHead>
      <TableHead>Actions</TableHead>
    </TableRow>
  </TableHeader>
```

- Creates **table headers** for claim data columns.

```
<TableBody>
  {filteredClaims.map((claim) => (
    <TableRow key={claim.id} className="hover:bg-muted/30">
```

- Loops through filtered claims and renders each as a table row.

```
<TableCell className="font-mono font-medium">{claim.id}</TableCell>
<TableCell className="font-medium">{claim.patientName}</TableCell>
<TableCell>
  <div>
    <div className="font-medium">{claim.hospital.split(',')[0]}</div>
    <div className="text-sm text-muted-foreground">{claim.location}</div>
  </div>
</TableCell>
```

- Displays ID, patient name, and hospital info.

```
<TableCell className={getAmountColor(claim.fraudStatus)}>
  {claim.amount}
</TableCell>
```

- Dynamically colors claim amount based on fraud status.

```
<TableCell>
  <div className="space-y-1">
    {getStatusBadge(claim.fraudStatus)}
    {claim.fraudReason && (
      <div className="text-xs text-muted-foreground max-w-xs">
        {claim.fraudReason}
      </div>
    )}
  </div>
</TableCell>
```

- Shows fraud status badge and fraud reason (if available).

```
<TableCell>
  <Button variant="outline" size="sm">
    <Eye className="w-4 h-4 mr-1" />
    View
  </Button>
</TableCell>
```

- View button (placeholder for future claim details view).

📝 Summary of `ClaimsTable.tsx`

- A **searchable and filterable table** for insurance claims.
- Features:
    - Search bar (patient, hospital, ID).

    - Filters for fraud status and location.
    - Color-coded badges for fraud levels.
    - Fraud reasons displayed inline.
    - Action button for viewing details.
- Built with **TypeScript** for strong typing and **TailwindCSS** for styling.

🗣️ How to Present This

“The `ClaimsTable.tsx` file renders our claims dashboard. It uses React’s `useState` for dynamic filters and a search bar, allowing users to quickly find claims by ID, patient, or hospital. Each claim’s fraud status is highlighted with color-coded badges, and suspicious or fraudulent claims are emphasized with warnings. The design is responsive and visually clear, with easy-to-use dropdown filters for fraud status and location. This makes claim review faster and more intuitive for insurance teams.”

Would you like me to now explain `ClaimUploader.tsx` (the **upload feature that talks to the backend**) in the same detailed way? That’s the key piece connecting frontend and fraud detection.

ClaimUploader.tsx

Perfect! Let’s break down `ClaimUploader.tsx` **line by line**.
This file builds your **upload and fraud analysis system** for insurance claim documents.
It lets users upload **PDFs or images**, sends them to your **FastAPI backend**, tracks progress, and shows fraud analysis results.

◆ File: `ClaimUploader.tsx` — Detailed Explanation

Imports

```
import { useState, useCallback } from 'react';
import { useDropzone } from 'react-dropzone';
import { Button } from "@/components/ui/button";
import { Progress } from "@/components/ui/progress";
import { Card, CardContent, CardHeader, CardTitle } from
"@/components/ui/card";
import { Badge } from "@/components/ui/badge";
import { Upload, FileText, Image, CheckCircle, AlertTriangle, Loader2 } from
"lucide-react";
import { useToast } from "@/components/ui/use-toast";
```

- `useState`, `useCallback`: React hooks for managing **component state** and performance optimizations.
- `useDropzone`: From **react-dropzone**, creates drag-and-drop file upload zones.
- `Button`, `Progress`, `Card`, `Badge`: Prebuilt UI components (styled with Tailwind).
- Icons (`Upload`, `FileText`, `AlertTriangle`, etc.) from **lucide-react** for better UX.
- `useToast`: Notification system for upload progress and errors.

UploadedFile Interface

```
interface UploadedFile {
  id: string;
  name: string;
  size: string;
  status: 'processing' | 'completed' | 'failed';
  fraudStatus?: 'clean' | 'suspicious' | 'fraudulent';
  extractedData?: any;
  error?: string;
}
```

Defines a TypeScript type for each uploaded file:

- `id`: Unique identifier for each file.
- `name`: File name.
- `size`: File size (human-readable, e.g., MB).
- `status`: Tracks processing state.
- `fraudStatus`: Result of fraud detection.
- `extractedData`: Contains OCR results or claim details from backend.
- `error`: Error message if upload fails.

Component State Setup

```
export const ClaimUploader = () => {
  const [files, setFiles] = useState<UploadedFile[]>([]);
  const [uploadProgress, setUploadProgress] = useState(0);
  const { toast } = useToast();
```

- `files`: Stores a list of uploaded files and their statuses.
- `uploadProgress`: Upload/processing progress (0–100%).
- `toast`: Used for showing notifications like success or failure.

onDrop Function (File Handling)

```
const onDrop = useCallback(async (acceptedFiles: File[]) => {
```

- `onDrop` is triggered when users **drop files** or select them.
- Wrapped in `useCallback` to prevent re-creating the function unnecessarily (performance boost).

```
const newFiles: UploadedFile[] = acceptedFiles.map(file => ({
  id: Math.random().toString(36).substring(7),
  name: file.name,
  size: `${(file.size / (1024 * 1024)).toFixed(2)} MB`,
  status: 'processing'
}));
```

- Converts uploaded files into `UploadedFile` objects.
- Generates a random `id`.
- Converts file size from bytes → MB.
- Sets initial status to **processing**.

```
setFiles(prev => [...prev, ...newFiles]);
setUploadProgress(0);
```

- Adds new files to `files` state.
- Resets upload progress.

```
toast({
  title: "Upload Started",
  description: `Processing ${acceptedFiles.length} file(s) for fraud
detection`,
});
```

- Shows a notification that files are being processed.

🔥 *Backend API Call*

```
for (let i = 0; i < acceptedFiles.length; i++) {
  const file = acceptedFiles[i];
  const fileId = newFiles[i].id;
  try {
    const formData = new FormData();
    formData.append('file', file);

    const response = await fetch('http://127.0.0.1:8000/api/claims/upload', {
      method: 'POST',
      body: formData,
    });
```

- Loops through files **one by one**.
- Prepares `FormData` with each file.
- Sends POST request to **FastAPI endpoint** `/api/claims/upload`.

```
if (!response.ok) {
  throw new Error(`HTTP error! status: ${response.status}`);
}
```

- Checks if API request was successful; otherwise throws error.

```
const result = await response.json();
```

- Parses JSON response from backend.
- Expected to contain `{ status: 'success', extractedData: {...} }`.

🔹 *Success Handling*

```
if (result.status === 'success') {
  setFiles(prev => prev.map(f =>
    f.id === fileId
      ? {
          ...f,
          status: 'completed',
          fraudStatus: result.extractedData.fraudStatus,
          extractedData: result.extractedData
        }
      : f
  ));

  toast({
    title: "Processing Complete",
    description: `${file.name} analyzed - Status:
${result.extractedData.fraudStatus}`,
  });
```

- Updates file status to **completed**.
- Stores fraud detection results in `extractedData`.
- Shows a success toast with fraud status.

Error Handling

```
} else {
  throw new Error('Processing failed');
}

} catch (error) {
  console.error('Upload failed:', error);
  setFiles(prev => prev.map(f =>
    f.id === fileId
      ? {
          ...f,
          status: 'failed',
          error: error instanceof Error ? error.message : 'Unknown error'
        }
      : f
  ));

  toast({
    title: "Upload Failed",
    description: `Failed to process ${file.name}`,
    variant: "destructive",
  });
}
```

- Catches errors and updates file status to **failed**.
- Logs error, displays toast notification.

Update Progress

```
setUploadProgress(((i + 1) / acceptedFiles.length) * 100);
```

- Updates upload progress bar as files are processed.

Dropzone Config

```
const { getRootProps, getInputProps, isDragActive } = useDropzone({
  onDrop,
  accept: {
    'application/pdf': ['.pdf'],
    'image/*': ['.png', '.jpg', '.jpeg']
  },
  multiple: true
});
```

- Configures drag-and-drop:
    - Accepts PDFs and images only.

  - Supports multiple uploads.
  - `getRootProps`, `getInputProps`: Provide props for drop area.

Status Helpers

```
const getStatusIcon = (status: string, fraudStatus?: string) => {
  if (status === 'processing') return <Loader2 className="h-4 w-4 animate-spin" />;
  if (status === 'failed') return <AlertTriangle className="h-4 w-4 text-danger" />;
  if (fraudStatus === 'fraudulent') return <AlertTriangle className="h-4 w-4 text-danger" />;
  if (fraudStatus === 'suspicious') return <AlertTriangle className="h-4 w-4 text-warning" />;
  return <CheckCircle className="h-4 w-4 text-success" />;
};
```

- Displays **icon based on status**:
  - Processing → Spinner.
  - Fraudulent → Red warning.
  - Suspicious → Yellow warning.
  - Clean → Green check.

```
const getStatusBadge = (status: string, fraudStatus?: string) => {
  if (status === 'processing') return <Badge variant="secondary">Processing</Badge>;
  if (status === 'failed') return <Badge className="bg-danger text-danger-foreground">Failed</Badge>;
  if (fraudStatus === 'fraudulent') return <Badge className="bg-danger text-danger-foreground">Fraudulent</Badge>;
  if (fraudStatus === 'suspicious') return <Badge className="bg-warning text-warning-foreground">Suspicious</Badge>;
  return <Badge className="bg-success text-success-foreground">Clean</Badge>;
};
```

- Shows **badge label** with colors.

Upload Area

```
<div {...getRootProps()} className={`border-2 border-dashed rounded-lg p-8 text-center cursor-pointer transition-all duration-200
  ${isDragActive
```

```
    ? 'border-primary bg-primary/5 shadow-lg shadow-primary/20'
    : 'border-border hover:border-primary/50 hover:bg-muted/30'}
`}>
```

- Dropzone with dynamic styles:
    - Changes appearance when files are dragged over.
    - Makes it clickable for file selection.

Upload Progress

```
{files.some(f => f.status === 'processing') && (
  <Card className="bg-muted/50">
    <CardHeader>
      <CardTitle className="text-sm">Processing Files...</CardTitle>
    </CardHeader>
    <CardContent>
      <Progress value={uploadProgress} className="w-full" />
      <p className="text-xs text-muted-foreground mt-2">
        Extracting data and analyzing for fraud indicators
      </p>
    </CardContent>
  </Card>
)}
```

- Shows progress bar **only while files are being processed**.

File List with Details

```
{files.length > 0 && (
  <div className="space-y-3">
    <h3 className="text-lg font-semibold">Uploaded Files</h3>
    {files.map((file) => (
      <Card key={file.id} className="p-4">
```

- Displays uploaded files after processing.

```
{file.extractedData && (
  <div className="mt-4 p-3 bg-muted/30 rounded-lg">
    <h4 className="text-sm font-semibold mb-2">Extracted Information</h4>
    <div className="grid grid-cols-2 gap-2 text-sm">
      <div><span className="font-medium">Claim ID:</span>
{file.extractedData.claimId || 'N/A'}</div>
```

```
    ...
  </div>
```

- If backend sent extracted data, display claim details like:
  - Claim ID
  - Patient
  - Hospital
  - Amount
  - Disease
  - Treatment
  - Fraud reason (highlighted if exists)

📝 Summary of `ClaimUploader.tsx`

Feature	Purpose
Drag-and-drop & file select	Easy document upload UI (PDF/Images).
Upload & API integration	Sends each file to FastAPI endpoint `/api/claims/upload`.
Real-time progress tracking	Shows progress bar as files are analyzed.
Fraud detection visualization	Displays fraud status badges/icons and detailed fraud reasons.
Error handling & notifications	Uses `toast` for feedback, shows failed uploads clearly.
Responsive & interactive UI	Built with Tailwind CSS, dynamic states, and icons for better UX.

🗣 How to Present This in Speech:

“The `ClaimUploader.tsx` file powers the document upload and fraud analysis feature. It lets users upload PDFs or images using drag-and-drop. Each file is sent to our FastAPI backend for processing. The component displays a live progress bar, fraud detection results, and extracted details like claim ID, hospital, and amount. We’ve added clear color-coded badges and icons for fraudulent or suspicious claims, along with error handling and notifications for a smooth user experience. This feature is a crucial bridge between the frontend and backend fraud detection system.”

Would you like me to also **trace exactly how this connects to your FastAPI backend route (`/api/claims/upload`)** step by step? That way, you'll see the full **frontend-backend pipeline**.

App.tsx

Perfect! Let's go through your **`App.tsx`** file **line by line**.
This is the **main entry point of your frontend React app**. It sets up routing, notifications, tooltips, and React Query for API data fetching.

◆ File: `App.tsx` — Detailed Explanation

Imports

```
import { Toaster } from "@/components/ui/toaster";
import { Toaster as Sonner } from "@/components/ui/sonner";
import { TooltipProvider } from "@/components/ui/tooltip";
import { QueryClient, QueryClientProvider } from "@tanstack/react-query";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Index from "./pages/Index";
import NotFound from "./pages/NotFound";
```

Import	Purpose
`Toaster` & `Sonner`	UI components for **toast notifications**. `Sonner` is just renamed because there are two toasters.
`TooltipProvider`	Wraps your app to enable tooltips everywhere.
`QueryClient` & `QueryClientProvider`	From **React Query**. `QueryClient` manages API calls, caching, and data fetching.
`BrowserRouter`, `Routes`, `Route`	From **React Router**. Handles client-side navigation without reloading pages.
`Index`	The homepage component.
`NotFound`	The 404 page for undefined routes.

Query Client Setup

```
const queryClient = new QueryClient();
```

- Creates a **new instance of QueryClient**.
- React Query uses this to:
    - Fetch data from APIs
    - Cache responses
    - Auto-refresh stale data

Main App Component

```
const App = () => (
  <QueryClientProvider client={queryClient}>
    <TooltipProvider>
      <Toaster />
      <Sonner />
      <BrowserRouter>
        <Routes>
          <Route path="/" element={<Index />} />
          {/* ADD ALL CUSTOM ROUTES ABOVE THE CATCH-ALL "*" ROUTE */}
          <Route path="*" element={<NotFound />} />
        </Routes>
      </BrowserRouter>
    </TooltipProvider>
  </QueryClientProvider>
);
```

Let's break this down:

🔹 *`<QueryClientProvider client={queryClient}>`*

- Wraps the entire app so that **React Query** can be used anywhere.
- This means components can easily fetch API data using hooks like `useQuery`.

🔹 *`<TooltipProvider>`*

- Wraps your app with a tooltip manager so any element can show helpful hints when hovered over.

🔹 *`<Toaster />` & `<Sonner />`*

- Adds **toast notifications** for user feedback (e.g., "File uploaded successfully" or "Error processing file").

🔹 `<BrowserRouter>`

- Enables **React Router**.
- This lets you navigate between pages like `/`, `/dashboard`, `/upload`, etc., **without reloading**.

🔹 `<Routes>` & `<Route>`

```
<Routes>
  <Route path="/" element={<Index />} />
  <Route path="*" element={<NotFound />} />
</Routes>
```

- Defines your app's routes:
  - `/` → Loads `Index` (homepage).
  - `*` → Catches **all undefined routes** and shows `NotFound` (404 page).

Exporting the App

```
export default App;
```

- Exports this component so `main.tsx` or `index.tsx` can render it in the browser.

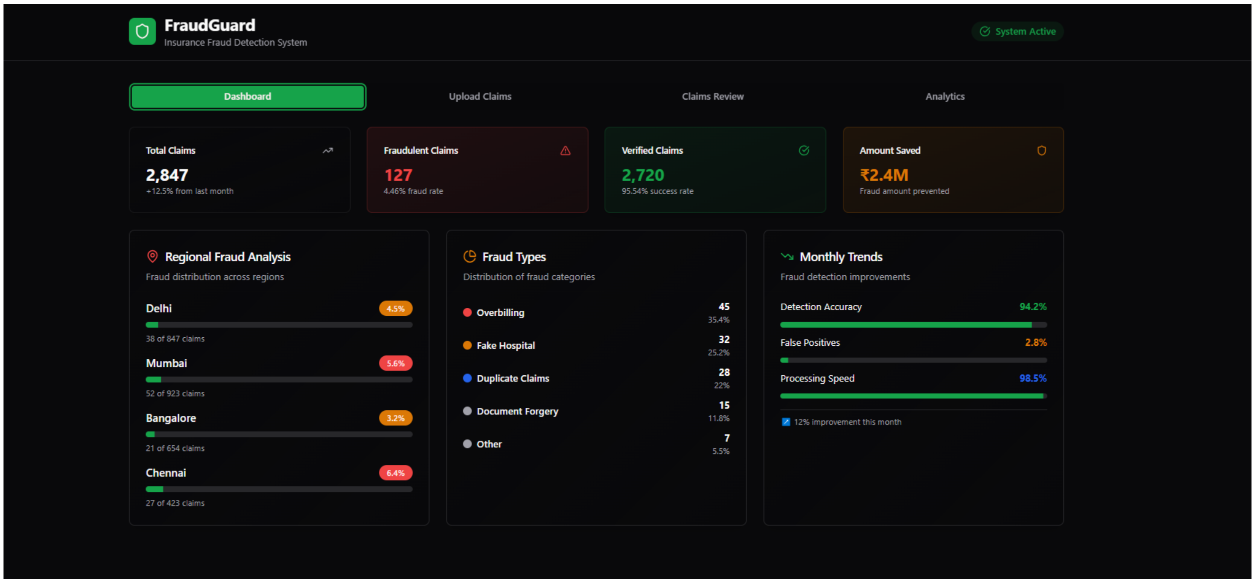
📝 Summary of `App.tsx`

Feature	Purpose
React Query Setup	Centralized API data fetching and caching.
Tooltip Provider	Easy tooltips across the app.
Toast Notifications	For success/error messages.
React Router	Enables smooth navigation without reloading pages.
404 Handling	Any wrong URL shows a `NotFound` page.

🗣️ How to Explain in a Presentation:

“Our `App.tsx` is the main entry point of the frontend. It sets up global tools like React Query for API data fetching, tooltips for better UX, and a notification system to give users instant feedback. We use React Router for smooth navigation between pages like the home page and a fallback 404 page for undefined routes. This structure ensures that our application is scalable, easy to maintain, and user-friendly.”

Would you also like me to explain **`Index.tsx` and `NotFound.tsx`** next so you can present the **full routing flow**?



Sure, here is an explanation of the FraudGuard dashboard, broken down by each section and its purpose.

Dashboard Overview

The dashboard provides a quick, high-level summary of the insurance fraud detection system's performance and key metrics. It's designed to give users a clear picture of the system's effectiveness and an overview of fraud trends.

Key Metrics Summary

This section at the top provides a quick snapshot of the most critical numbers:

- **Total Claims (2,847)**: The total number of insurance claims processed by the system. The **+12.5%** metric indicates the processing volume has increased from the previous month, suggesting the system is actively being used.
- **Fraudulent Claims (127)**: The number of claims flagged as fraudulent. The accompanying **4.46% fraud rate** shows the percentage of total claims that are fraudulent, which is a key performance indicator.
- **Verified Claims (2,720)**: The number of claims that have been verified as legitimate. The **95.54% success rate** is the inverse of the fraud rate, highlighting the system's accuracy in correctly processing valid claims.
- **Amount Saved (₹2.4M)**: This metric quantifies the financial impact of the system, showing the total amount of money prevented from being paid out on fraudulent claims. This is a crucial metric for demonstrating the system's value and return on investment (ROI).

Regional Fraud Analysis

This chart visualizes the distribution of fraud across different geographic regions. It helps identify which areas are most affected by fraudulent claims. The bar chart shows the total number of fraudulent claims for each city, along with the specific fraud rate for that region. For example:

- **Delhi (4.5%)**: Shows that out of the 847 claims from this region, 4.5% were fraudulent.
- **Mumbai (5.6%)**: Indicates a higher fraud rate of 5.6%, which is a red flag for the region.
- **Bangalore (3.2%)**: Has a lower fraud rate compared to others.
- **Chennai (6.4%)**: Has the highest fraud rate, suggesting a need for increased scrutiny in this region. This information is valuable for resource allocation and targeted fraud investigations.

Fraud Types Distribution

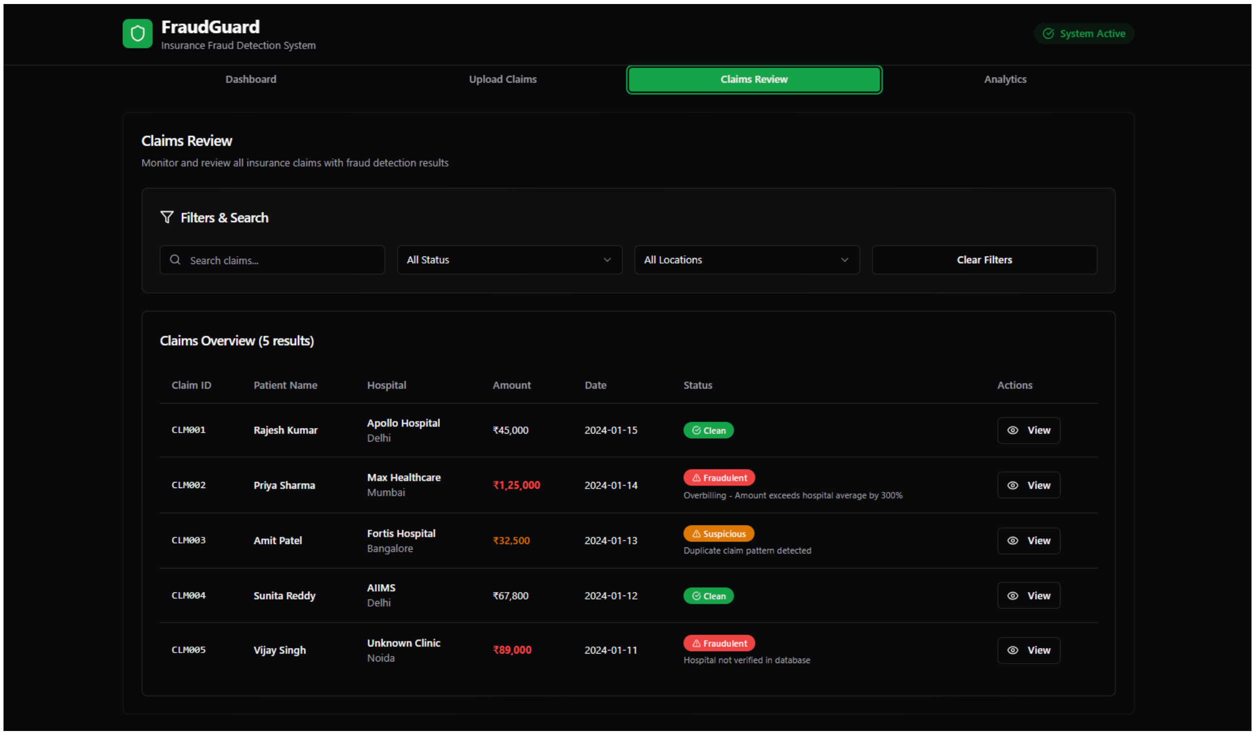
This section breaks down the fraudulent claims by category, providing insight into the most common types of fraud detected by the system.

- **Overbilling (35.4%)**: This is the most common fraud type, where medical services or products are billed at an inflated price.
- **Fake Hospital (25.2%)**: Refers to claims submitted from non-existent or unapproved medical facilities.
- **Duplicate Claims (22%)**: When the same claim is submitted multiple times to get multiple payouts.
- **Document Forgery (11.8%)**: Claims that involve falsified documents like prescriptions or medical reports.
- **Other (5.5%)**: A catch-all category for less common types of fraud. This breakdown helps in developing more specific fraud detection rules and training for staff.

Monthly Trends

This section tracks the system's performance over time, showing its effectiveness in detecting fraud.

- **Detection Accuracy (94.2%)**: Measures how well the system correctly identifies fraudulent claims. The high percentage shows the model is performing well.
- **False Positives (2.8%)**: Represents the percentage of legitimate claims incorrectly flagged as fraudulent. A low false positive rate is crucial to avoid inconveniencing genuine customers and over-burdening manual review teams.
- **Processing Speed (98.5%)**: Indicates the efficiency of the system in processing claims quickly.
- The **"12% improvement this month"** note is a positive reinforcement, showing that the system is constantly being refined and is getting better at its job.



Claims Review

Monitor and review all insurance claims with fraud detection results

Claims Overview (5 results)

Claim ID	Patient Name	Hospital	Amount	Date	Status	Actions
CLM001	Rajesh Kumar	Apollo Hospital Delhi	₹45,000	2024-01-15	Clean	View
CLM002	Priya Sharma	Max Healthcare Mumbai	₹1,25,000	2024-01-14	Fraudulent Overbilling - Amount exceeds hospital average by 300%	View
CLM003	Amit Patel	Fortis Hospital Bangalore	₹32,500	2024-01-13	Suspicious Duplicate claim pattern detected	View
CLM004	Sunita Reddy	AIIMS Delhi	₹67,800	2024-01-12	Clean	View
CLM005	Vijay Singh	Unknown Clinic Noida	₹89,000	2024-01-11	Fraudulent Hospital not verified in database	View

Based on the image provided, here is a detailed explanation of the "Claims Review" dashboard.

Claims Review Dashboard Overview

This page is designed for users to actively monitor, search, and review individual insurance claims and their corresponding fraud detection results. It acts as a central hub where the fraud detection system's output is presented in a clear, actionable format.

Filters & Search

This section at the top allows users to efficiently find specific claims within the system. This is crucial for managing a large volume of claims.

- **Search claims...**: A search bar where users can type keywords to find claims, likely by Claim ID, Patient Name, or Hospital.
- **All Status**: A dropdown menu to filter claims by their fraud status. The options would likely include "Clean", "Suspicious", "Fraudulent", or "Pending".
- **All Locations**: A dropdown to filter claims by their geographical location (e.g., city, region), which helps in analyzing regional fraud patterns.
- **Clear Filters**: A button to reset all search and filter options, restoring the default view.

Claims Overview (5 results)

This is the core of the dashboard, a table that lists the claims that match the current filters and search criteria. Each row represents a single claim and provides key information at a glance.

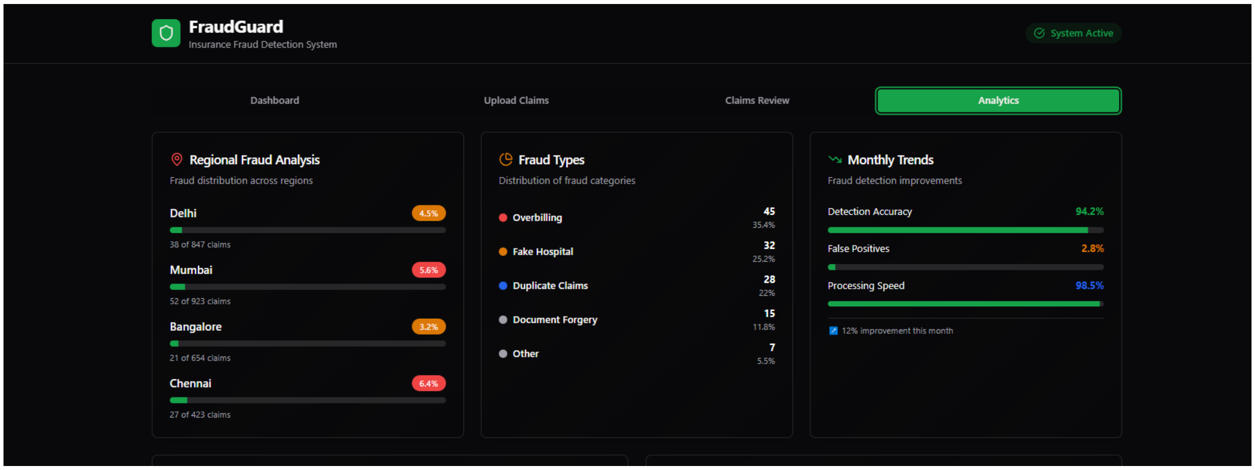
Table Columns:

- **Claim ID**: A unique identifier for each claim (e.g., `CLM001`). This is used for easy reference and tracking.
- **Patient Name**: The name of the person who submitted the claim (e.g., `Rajesh Kumar`).
- **Hospital**: The name and location of the hospital where the service was provided (e.g., `Apollo Hospital, Delhi`).
- **Amount**: The total amount requested for the claim. Amounts that are flagged as fraudulent or suspicious might be highlighted (e.g., `₹1,25,000` and `₹39,000` are in red, `₹32,500` in orange, while the others are black).
- **Date**: The date the claim was submitted or the date of service.
- **Status**: This column is the most important as it shows the result of the fraud detection. The system uses visual indicators (badges) to make the status immediately clear:
  - **Clean (Green badge)**: The claim is considered legitimate with no red flags.
  - **Suspicious (Orange badge)**: The claim has some inconsistencies but is not definitively fraudulent. It may require a manual review.
  - **Fraudulent (Red badge)**: The claim has been flagged as highly likely to be fraudulent. A brief reason is often provided below the status for quick context (e.g., "Overbilling - Amount exceeds hospital average by 300%" or "Hospital not verified in database").
- **Actions**: This column provides a `View` button. Clicking this button would likely take the user to a detailed page for that specific claim. This page would show more information, such as the uploaded documents, the specific rules that were triggered, and a full breakdown of the fraud analysis.

Example Breakdown of a Single Claim:

Let's look at CLM002 for Priya Sharma.

- **Status**: It's a "Fraudulent" claim.
- **Reason**: The system has detected "Overbilling - Amount exceeds hospital average by 300%". This gives the reviewer a specific, data-backed reason for the fraudulent tag, allowing them to quickly understand the issue and decide on the next steps, such as denying the claim or initiating a deeper investigation.



Regional Fraud Analysis
Fraud distribution across regions

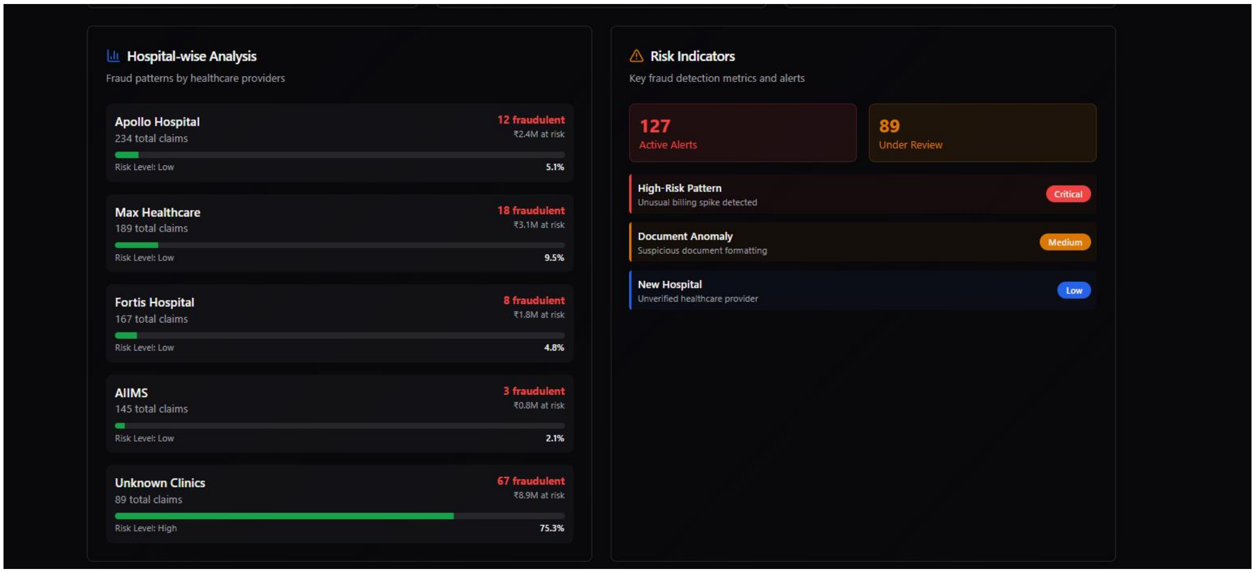
Delhi 4.5%
38 of 847 claims
Mumbai 5.6%
52 of 923 claims
Bangalore 3.2%
21 of 654 claims
Chennai 6.4%
27 of 423 claims

Fraud Types
Distribution of fraud categories

Overbilling	45	35.4%
Fake Hospital	32	25.2%
Duplicate Claims	28	22%
Document Forgery	15	11.8%
Other	7	5.5%

Monthly Trends
Fraud detection improvements

Detection Accuracy 94.2%
False Positives 2.8%
Processing Speed 98.5%
12% improvement this month


Hospital-wise Analysis
Fraud patterns by healthcare providers

Apollo Hospital
234 total claims
12 fraudulent
₹2.4M at risk
Risk Level: Low
5.1%

Max Healthcare
189 total claims
18 fraudulent
₹3.1M at risk
Risk Level: Low
9.5%

Fortis Hospital
167 total claims
8 fraudulent
₹1.8M at risk
Risk Level: Low
4.8%

AIIMS
145 total claims
3 fraudulent
₹0.8M at risk
Risk Level: Low
2.1%

Unknown Clinics
89 total claims
67 fraudulent
₹8.9M at risk
Risk Level: High
75.3%

Risk Indicators
Key fraud detection metrics and alerts

127
Active Alerts

89
Under Review

High-Risk Pattern
Unusual billing spike detected
Critical

Document Anomaly
Suspicious document formatting
Medium

New Hospital
Unverified healthcare provider
Low

Based on the images provided, here is a detailed explanation of the additional charts and sections on the analytics dashboard.

Hospital-wise Analysis

This section of the dashboard is designed to identify fraud patterns associated with specific healthcare providers. It lists hospitals and clinics, showing their individual fraud statistics to help users quickly spot high-risk providers.

- **Apollo Hospital**: Out of **234** total claims, **12** were flagged as fraudulent, representing a fraud rate of **5.1%**. The "Risk Level: Low" indicates that while there is some fraudulent activity, it is not a major concern relative to other providers.
- **Max Healthcare**: With **189** total claims, **18** were found to be fraudulent. The higher fraud rate of **9.5%** and the "Risk Level: Low" tag suggests this hospital warrants closer monitoring.
- **Fortis Hospital**: Shows a fraud rate of **4.8%** from **167** claims, with **8** being fraudulent. The risk is considered low.
- **AIIMS**: Has a very low fraud rate of **2.1%**, with only **3** fraudulent claims out of **145**.
- **Unknown Clinics**: This entry is a significant red flag. With **67** fraudulent claims out of **89** total, it has an exceptionally high fraud rate of **71.3%**. The "Risk Level: High" tag indicates that claims from these unverified providers are a major source of fraud, making this a critical area for investigation.

Risk Indicators

This section acts as a live feed of key fraud detection metrics and alerts, providing a summary of ongoing issues that require attention.

- **Active Alerts (127)**: This number indicates the total count of claims that have been flagged by the system and require review. This could include claims marked as "Suspicious" or "Fraudulent".
- **Under Review (89)**: This metric shows how many of the active alerts are currently being manually reviewed by the fraud analysis team. A high number here might suggest a backlog in the review process.

The section also provides a summary of specific types of alerts, categorized by severity:

- **High-Risk Pattern (Critical)**: This alert is for an "unusual billing spike detected." This is a critical issue as it suggests a sudden, significant increase in fraudulent activity, possibly indicating a new fraud scheme or a large-scale attempt.
- **Document Anomaly (Medium)**: This alert, for "suspicious document formatting," indicates a medium-priority issue. It could point to falsified or tampered documents that don't conform to the expected format, which requires a closer look.
- **New Hospital (Low)**: This low-priority alert for a "unverified healthcare provider" simply means a claim has been received from a hospital that is not yet in the system's database. While this is not necessarily fraudulent, it requires verification to prevent future fraud attempts from that provider.

SPEECH

Presentation Speech

"Hello everyone, I'd like to present our project on insurance fraud detection. The primary challenge we tackled was identifying fraudulent claims, specifically by pinpointing problematic hospitals and regions. We weren't asked to build a full dashboard, so we focused on the core logic: a tool that processes a claim and immediately tells us if it's fraudulent and why.

Our solution is a two-part system. First, the **backend**, built with **FastAPI and Python**. This is the brain of our application. A user simply uploads a claim document, like a PDF. The backend then uses **OCR and PDF parsing** to read and extract critical information like the patient's name, the hospital, the treatment, and the claimed amount.

The extracted data is then fed into our fraud detection logic. We have a set of rules and datasets that allow us to run checks. For example:

- **Hospital Check**: Is this hospital in our database of verified providers? If not, it's flagged as potentially fraudulent.
- **Treatment Check**: Does the treatment on the claim match the diagnosis? We can check if a claim for a "fever" includes an unusually expensive "surgical procedure."
- **Amount Check**: Is the claimed amount suspiciously high for this type of treatment?

This process allows us to not only detect fraud but also provide a reason for the flag. For example, a claim from a high-risk hospital or a claim with an inflated cost.

The **frontend**, built with **React**, is a simple, user-friendly interface. It allows anyone to easily **upload a claim document** with a drag-and-drop feature. Once the claim is uploaded, it's sent to our backend for analysis. The result is then displayed on the screen: a clear verdict of **'clean,' 'suspicious,' or 'fraudulent,'** along with the specific reason for the result. This makes the entire process transparent and easy to use.

In summary, our project automates a critical part of the insurance business. It takes a claim, analyzes it instantly, and provides a clear, actionable result based on a robust logic that considers fraud patterns by hospital and region. This simple yet powerful tool can significantly reduce fraud and save insurance companies millions."