Chargeback Processing System: Full Migration & Modern Architecture Documentation

1. Introduction

This document outlines the full migration of a Chargeback Processing System originally built using Streamlit into a modern, decoupled architecture using **FastAPI (Python)** for backend logic and **React (TypeScript)** for the frontend UI.**Goals of Migration:**

- **Improve scalability and maintainability**
- **Decouple business logic from presentation layer**
- **Enable integration with external systems**
- **Provide a modern, responsive user interface**

This documentation serves as both a technical reference and a step-by-step migration playbook.

2. Original Streamlit Architecture

2.1 How Streamlit Worked

The original **Chargeback Processing System** was implemented using **Streamlit**, a Python-based web framework primarily used for building quick data apps and dashboards with minimal frontend effort.

The entire system was encapsulated in a **single Python script** named `gveiv_ui.py`. This file handled all the following responsibilities:

- **User interface rendering**
- **Business logic execution**
- **Session management**
- **Data processing pipeline orchestration**
- **Display of results and metrics**

Key Functional Areas in Streamlit:

- **File Upload**:
  The application used `st.file_uploader()` to allow users to upload transaction CSV files from their local systems.
- **Progress and Status Display**:
  Streamlit's `st.progress` and `st.info` components were used to show real-time progress updates to users during pipeline execution.
- **Output Rendering**:
  Final processed results, including tables and KPIs, were shown using `st.dataframe()` and `st.metric()`.
- **Session State Management**:
  Streamlit's `st.session_state` was used to retain uploaded files, intermediate results, and user interactions between reruns of the app.

Advantages of Streamlidt in Initial Prototype:

- Extremely fast evelopment
- Simple UI building for data scientists
- No need to write HTML/JS
- Easy debugging and deployment

Limitations of Streamlit Architecture:

Area	Limitation
Scalability	Designed for single-user sessions. Concurrent users face conflicts or redundant reruns.
Separation of Concerns	All logic (data, UI, session state) lived in one file, making it difficult to test or refactor.
UI Customization	Limited control over styling and animations. Not suitable for modern, responsive UIs.
Extensibility	Streamlit is not API-first. Integrating with third-party services, APIs, or external UIs was cumbersome.

2.2 Example Code Snippets

Below are key sections from the original `gveiv_ui.py` Streamlit script that illustrate how each core functionality was implemented.

A. File Upload with Session Storage

```
uploaded_file = st.file_uploader("Upload CSV", type=["csv"])
if uploaded_file:
    df = pd.read_csv(uploaded_file)
    st.session_state["transactions"] = df
```

Explanation:

- The user was prompted to upload a `.csv` file.
- Once uploaded, the file was read into a pandas DataFrame.
- The data was stored in `st.session_state["transactions"]` so it could be reused in other parts of the app (e.g., processing pipeline).

Why it's problematic long-term:

- File uploads only exist in memory during a session.
- No external or cloud-based storage.
- Can't track files across users or restarts.

B. Progress Indicators and Processing Status

```
progress_bar = st.progress(0)
status_text = st.empty()

for i in range(100):
    progress_bar.progress(i + 1)
    status_text.text(f"Processing... {i+1}%")
    time.sleep(0.01)
```

Explanation:

- Simulated progress bar using a loop with `time.sleep`.
- Displayed dynamic status text for better user feedback.
- Enhanced UX during longer processing tasks.

Issues:

- Artificial progress — not tied to real agent completion.
- No actual pipeline tracking.
- Not suitable for long-running background jobs or async logic.

C. Running the Processing Pipeline

```
if st.button("Start Processing"):
    result_df = process_pipeline(st.session_state["transactions"])
    st.session_state["results"] = result_df
```

Explanation:

- When the user clicked "Start Processing," the function `process_pipeline()` was called using the uploaded transaction data.
- The output (a processed DataFrame) was saved into `st.session_state["results"]`.

Drawbacks:

- All logic was synchronous and ran on the main thread.
- No separation between agent stages.
- No error handling or job tracking (e.g., pipeline status).

D. Displaying Results and Downloading CSV

```
if "results" in st.session_state:
    st.dataframe(st.session_state["results"])
    st.download_button("Download Results", data=to_csv(),
file_name="results.csv")
```

Explanation:

- Displayed the processed results in a scrollable interactive table using `st.dataframe()`.
- Users could download the final result CSV using `st.download_button`.

Limitations:

- Static rendering — no pagination, filtering, or real interactivity.
- Downloaded file generated client-side, without audit or record.

Summary of Streamlit Shortcomings

Function	Streamlit Code	Shortcoming
File Upload	`st.file_uploader()`	No persistent storage or tracking
State Management	`st.session_state`	Bound to browser session; volatile
Long Tasks	Inline `for` loop with sleep	Blocking, UI-freezing behavior
Agent Processing	`process_pipeline()`	No modular agent handling
Results UI	`st.dataframe`, `st.metric`	Basic widgets, no customization
Download	`st.download_button()`	No access logging or API

This tightly coupled architecture provided rapid development in early stages, but posed significant challenges in terms of:

- Extending functionality
- Collaborating in teams
- Integrating with external systems
- Scaling for multiple users
- Achieving enterprise-grade UX

3. Why Migrate?

The initial version of the Chargeback Processing System was built rapidly using Streamlit. While this helped in early prototyping and stakeholder demos, several structural, performance, and maintainability issues made it unsuitable for production use.

🔍 Detailed Reasons:

3.1 Scalability Issues

Streamlit Limitation:
Streamlit apps run in a single-threaded model where the entire app is rerun from top to bottom on each interaction (like a button click). This architecture is not suitable for concurrent users, as it lacks native support for multi-user sessions, load balancing, or persistent storage.

Why FastAPI + React Solves It:

- FastAPI handles multiple simultaneous requests using asynchronous I/O.
- React runs independently on the browser, handling UI state locally.
- Backend logic (like long-running pipelines) runs in the background, freeing up API threads for other users.

3.2 Separation of Concerns

Problem in Streamlit:
UI, backend logic, session management, and file handling were all implemented in a single Python script (`gveiv_ui.py`). This tight coupling made it:

- Hard to write unit tests
- Difficult to maintain or extend logic
- Impossible to separate developer roles (frontend vs backend)

Modern Approach:

- FastAPI handles backend logic, validation, and data pipelines
- React manages frontend UI and user interaction
- Clean REST APIs separate communication clearly between the two

3.3 UI Customization Limitations

Streamlit's Weakness:
You're limited to built-in widgets like sliders, buttons, and checkboxes, with minimal styling control. Advanced components (drag-drop, real-time animations, interactive dashboards) are nearly impossible.

React's Strength:

- Tailwind CSS allows precise utility-based styling
- Framer Motion enables smooth transitions and micro-animations
- Recharts and Chart.js allow for interactive and animated charts

3.4 API Integration Needs

Streamlit Issues:
It does not offer routes or REST APIs for third-party tools to access data, making it hard to integrate with:

- CRMs (like HubSpot)
- External dashboards
- Automation pipelines
- Microservices

FastAPI:
Built with API-first in mind. Each route is:

- Exposed using OpenAPI
- Automatically documented (Swagger UI)
- Easily testable using Postman or curl

3.5 Future-Proofing

Modern enterprise systems demand:

- Modular design
- Language agnostic frontends
- Scalable microservices
- DevOps-friendly deployment

The new architecture aligns with these principles by decoupling frontend and backend and making the system cloud-native, testable, and scalable.

How React Fetches and Displays Data

React plays a crucial role as the frontend interface of the Chargeback Processing System. Its responsibility includes accepting data from FastAPI, triggering pipeline operations, managing state, and rendering results such as tables, charts, KPIs, and downloadable files. Below is a breakdown of how React handles the full data lifecycle.

✨ How React Accepts Data

1. API Calls Using Axios

- Axios is a promise-based HTTP client that allows React to communicate with the FastAPI backend.
- It supports `GET`, `POST`, `PUT`, and `DELETE` methods.

- Common Axios setup includes:
- ```
  import axios from 'axios';
  const api = axios.create({ baseURL: '/api' });
  ```

- Error handling and request interceptors can also be added globally.

2. Data Format Handling

- **JSON Data**: Used for structured responses such as tabular data, progress status, and metadata.
- **Blob/File**: Used for downloadable content like CSV files. Axios must set `responseType: 'blob'` when downloading.

3. State Management

- React stores data in local state using `useState()`:

  ```
  const [results, setResults] = useState<Transaction[]>([]);
  ```

- For global or shared state, `useContext()` is used:

  ```
  const { user } = useContext(UserContext);
  ```

- Persistent state like theme or auth tokens may go into `localStorage` or `sessionStorage`.

📊 How Data is Fetched

1. On File Upload

- A file is selected via:

  ```
  <input type="file" onChange={handleFileSelect} />
  ```

- Then wrapped in `FormData` and posted:
- ```
  const handleUpload = async () => {
  ```
- ```
    const formData = new FormData();
  ```
- ```
    formData.append("file", selectedFile);
  ```
- ```
    await axios.post("/upload", formData);
  ```
  ```
  };
  ```

2. On Pipeline Start

- Triggered by a button click.
- Sends the uploaded file path to backend:
- ```
  const startPipeline = async (filePath) => {
  ```
- ```
    const res = await axios.post("/start-pipeline", { filePath });
  ```
- ```
    return res.data.pipeline_id;
  ```
  ```
  };
  ```

3. Polling Status

- Uses `useEffect()` + `setInterval()` to regularly hit `/status/{pipeline_id}`.
- Once status is "completed", stops polling and calls `fetchResults()`.

4. On Completion

- React fetches processed results from `/results/{pipeline_id}` and stores in local state.
- These results are then used across multiple UI components.

🌎 How Data is Displayed in React

1. Tables

- Tables are built using `map()` or libraries like `react-table`.
- Example:
- `<table>`
- `  <thead><tr><th>ID</th><th>Amount</th><th>Status</th></tr></thead>`
- `  <tbody>`
- `    {results.map(r => (`
- `      <tr key={r.id}><td>{r.id}</td><td>{r.amount}</td><td>{r.status}</td></tr>`
- `    ))}`
- `  </tbody>`
  `</table>`

2. Charts

- Libraries like `Recharts`, `Chart.js`, or `Victory` are used.
- Metrics are aggregated and passed as props to components like `<BarChart />`, `<PieChart />`, etc.

3. KPIs / Metrics

- Simple number cards showing total records, disputed count, recovered amount.
- Rendered via components like:

```
<MetricCard label="Disputes" value={totalDisputes} />
```

4. Downloads

- A download button sends a request to `/download/{pipeline_id}`.
- Receives file as Blob and auto-downloads:
- `const res = await axios.get('/download/123', { responseType: 'blob' });`
- `const url = URL.createObjectURL(res.data);`
- `const a = document.createElement('a');`
- `a.href = url;`
- `a.download = 'results.csv';`

```
a.click();
```

How FastAPI Sends Data

FastAPI acts as a bridge between React and the backend logic, often fetching from S3 and sending the result back.

🧾 JSON Responses

- FastAPI returns Python lists/dictionaries which are automatically converted to JSON.
- For DataFrames:

```
return df.to_dict(orient="records")
```

🗂️ File Responses

- CSVs or PDF invoices can be served via `FileResponse`:

```
return FileResponse("/tmp/report.csv", media_type="text/csv")
```

⏳ Status Updates

- To keep frontend informed, backend returns status objects:

```
return {"progress": 50, "status": "in_progress"}
```

How Components Get Data

🔍 Props

- Most data is passed down from parent to child component:

```
<ResultsTable data={results} />
```

🌐 Context API

- For shared state like pipeline ID or auth user:

```
const { pipelineId } = useContext(AppContext);
```

🕸️ Custom Hooks

- Hooks simplify reuse of fetch logic:
- `const useResults = (pipelineId) => {`
- `  const [data, setData] = useState([]);`
- `  useEffect(() => {`

```
    axios.get(`/results/${pipelineId}`).then(res =>
setData(res.data));
  }, [pipelineId]);
  return data;
}
```

How DataFrames are Handled

🧰 Backend: Pandas Usage

- Raw transaction data is loaded into `pandas.DataFrame`.
- Each agent manipulates the DataFrame and saves intermediate results to S3.
- At the end:
  - `df.to_dict()` is used for API JSON
  - `df.to_csv()` is used for download

🌐 Frontend: Rendering & Downloading

- JSON: Rendered using JSX `<table>` or chart libraries.
- CSV: Triggered download using `Blob` and `URL.createObjectURL()`.
- Advanced feature: `XLSX` export can be supported with SheetJS in frontend.

4. New Architecture Overview

The migrated system consists of two loosely coupled components:

- A **backend** in FastAPI for business logic and data processing
- A **frontend** in React for a rich user interface experience

4.1 Backend (FastAPI) — Detailed

📦 Tech Stack:

- Python 3.11+
- FastAPI
- Uvicorn (ASGI server)
- Pydantic (data validation)
- boto3 (for AWS S3)
- Redis (for pipeline state)
- PostgreSQL (for user/session data)

📂 Upload Endpoint

```
from fastapi import FastAPI, UploadFile, File
import boto3

app = FastAPI()
s3 = boto3.client("s3")
```

```
BUCKET_NAME = "chargeback-system-bucket"

@app.post("/upload/")
async def upload_file(file: UploadFile = File(...)):
    file_path = f"uploads/{file.filename}"
    s3.upload_fileobj(file.file, BUCKET_NAME, file_path)
    return {"filename": file.filename, "path": file_path}
```

What happens here?

- Accepts a file through multipart/form-data from the frontend
- Saves the file to a secure S3 bucket (`uploads/` folder)
- Returns metadata back to the frontend for future reference

Start Pipeline Endpoint

```
from uuid import uuid4
import asyncio

@app.post("/start-pipeline/")
async def start_pipeline(data: PipelineRequest):
    pipeline_id = str(uuid4())
    asyncio.create_task(run_pipeline(data.file_path, pipeline_id))
    return {"pipeline_id": pipeline_id}
```

- This endpoint triggers an asynchronous pipeline run.
- Each pipeline run is uniquely identified by a UUID.
- Actual agent execution is handled by `run_pipeline(...)`, which calls Agent 1 through Agent 4.

4.2 Frontend (React + Vite) — Detailed

🛠 Tech Stack

- React 19 + Vite (blazing-fast development)
- Tailwind CSS (for styling)
- Axios (for API communication)
- Recharts + Chart.js (for analytics)
- Framer Motion (for animation)
- Context API (for global state)
- TypeScript (for safety)

File Upload Component

```
const FileUploadCard = () => {
  const [file, setFile] = useState<File | null>(null);

  const handleUpload = async () => {
    const formData = new FormData();
    formData.append("file", file!);
    const response = await axios.post("/api/upload", formData);
    console.log(response.data);
  };
```

```
  return (
    <div className="p-4 border rounded">
      <input type="file" onChange={(e) => setFile(e.target.files?.[0] ||
null)} />
      <button onClick={handleUpload} className="bg-blue-500 text-white px-4
py-2 mt-2 rounded">
        Upload File
      </button>
    </div>
  );
};
```

How It Works:

- Captures the selected file using a React state variable
- Wraps it in a `FormData` object
- Sends the file to FastAPI via Axios POST
- Displays or logs the response (e.g., uploaded file path)

Polling Pipeline Status

```
useEffect(() => {
  const interval = setInterval(async () => {
    const res = await axios.get(`/api/status/${pipelineId}`);
    if (res.data.status === "completed") {
      clearInterval(interval);
      setResults(res.data.results);
    }
  }, 2000);

  return () => clearInterval(interval);
}, [pipelineId]);
```

- Continuously checks the backend for pipeline status
- Stops polling when processing is complete
- Fetches results and updates the UI accordingly

5. Detailed Migration Steps

5.1 File Upload

Layer	Streamlit	FastAPI	React
Upload Method	`st.file_uploader()`	`/upload/` endpoint	`<input type="file">` + Axios

Old Flow (Streamlit):

```
uploaded_file = st.file_uploader("Upload CSV", type=["csv"])
if uploaded_file:
    df = pd.read_csv(uploaded_file)
    st.session_state["transactions"] = df
```

New Flow:

1. **React**:
    - User selects a file from browser.
    - `handleUpload()` sends the file to `/upload/` using Axios.
2. **FastAPI**:
    - Receives file using `UploadFile`
    - Saves it to `uploads/filename.csv` in S3
    - Returns file path in response
3. **React**:
    - Stores the path in a state variable
    - Enables further buttons like "Start Pipeline"

5.2 Pipeline Processing

Layer	Old	New
Trigger	`st.button(...)`	`axios.post("/start-pipeline")`
Execution	Inline function call	Background async task
Polling	Not supported	React polling `/status/{id}`

New Flow:

```
const startPipeline = async () => {
  const res = await axios.post("/api/start-pipeline", { filePath });
  setPipelineId(res.data.pipeline_id);
};
```

- React sends a POST to start the pipeline.
- Backend launches a background process.
- React polls every 2 seconds to get status updates.

5.3 Agent Logic (Python Classes)

Each agent is structured as a class with a `run()` or task-specific method. Example:

```
class DisputeClassifier:
    def classify(self, transactions: pd.DataFrame) -> pd.DataFrame:
        # Rule-based or ML-based classification
        transactions["dispute_type"] =
transactions["reason_code"].map(MY_RULES)
        return transactions
```

- **Agent 1**: Validates and ingests the CSV data
- **Agent 2**: Classifies dispute categories
- **Agent 3**: Enriches data with external or user info
- **Agent 4**: Makes final chargeback decision

Each agent's inputs and outputs are stored in **S3**, not `session_state`, which improves testability and scalability.

5.4 Results Display

Platform	Method
Streamlit	`st.dataframe()`, `st.metric()`
FastAPI	`/results/{pipeline_id}` (JSON or CSV)
React	`<ResultsTable />`, `<MetricCard />`, `<ChartComponent />`

```
const fetchResults = async () => {
  const res = await axios.get(`/api/results/${pipelineId}`);
  setResults(res.data);
};
```

- Fetches tabular data from backend
- Renders in rich React components with sorting, charts, KPIs

5.5 Session State & User Profile

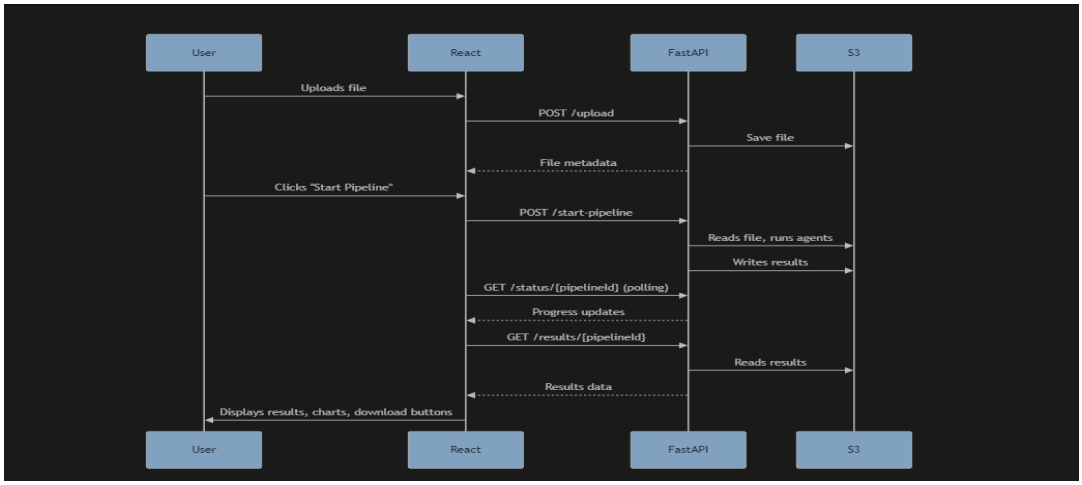
Concern	Streamlit	FastAPI	React
Temporary State	`st.session_state`	Redis	Context API
Persistent User Data	Not supported	PostgreSQL	localStorage

New Behavior:

- **Temporary UI state** (e.g., loading flags, selected file) is managed via `useContext()` or `useState()` in React.
- **Backend state** (e.g., pipeline progress, user ID, API keys) is stored in **Redis** or **PostgreSQL** for durability and concurrent access.

6. Data Flow: End-to-End

Architecture Flow



Steps

1. **User uploads file in React**
2. **FastAPI uploads to S3**
3. **React calls /start-pipeline**
4. **FastAPI runs agents**
5. **React polls /status**
6. **On complete, fetches from /results**
7. **Renders table + metrics**

🔄 Step-by-Step Process Flow

1. **User Uploads File in React**
   The user selects a `.csv` file using the `FileUploadCard` React component. The file is submitted via Axios to the `/upload` endpoint of the FastAPI backend
2. **FastAPI Uploads File to S3**
   FastAPI receives the file as a `UploadFile` object and stores it into a predefined S3 bucket (`uploads/` directory). Metadata like filename and path are returned.
3. **React Calls `/start-pipeline`**
   After the file is uploaded, React makes a POST request to `/start-pipeline` passing the uploaded file path. FastAPI triggers a background task using `asyncio.create_task`.
4. **FastAPI Executes Pipeline Agents**
   Agents run in sequence:
   - Agent 1: File validation & ingestion
   - Agent 2: Dispute classification
   - Agent 3: Enrichment

- Agent 4: Final decision
  Each agent writes intermediate output to S3 and returns a result for the next.

5. **React Polls `/status/{pipeline_id}`**
   While the pipeline is executing, React polls FastAPI every 2 seconds to check if the pipeline has completed.
6. **Pipeline Completion, React Calls `/results/{pipeline_id}`**
   Once the pipeline status is `completed`, React fetches results from FastAPI using a GET request.
7. **Results Rendered: Table, KPIs, Charts**
   React parses the JSON response and renders it using components:
   - `ResultsTable` for tabular data
   - `MetricCard` for KPIs
   - `ChartComponent` for graphical insights
   - `DownloadButton` for CSV downloads
   -

7. Frontend Component Mapping

This comparison illustrates how each Streamlit widget was mapped to a custom or reusable React component in the new architecture. React not only replaces each function but significantly enhances customization, reusability, and responsiveness.

Streamlit Widget	React Equivalent	Description
`st.file_uploader()`	`FileUploadCard`	Drag-drop file input with Axios integration
`st.progress()`	`ProgressBar`, `AgentCards`	Animated pipeline status using Framer Motion
`st.dataframe()`	`ResultsTable`	Sortable, filterable table using React Table
`st.metric()`	`MetricCard`, `Dashboard`	KPIs rendered with animation
`st.download_button()`	`DownloadButton`	Button to download CSV using Blob API
`st.session_state`	`Context API`, Redis	Session state split between frontend and backend

⚠ React frontend uses **Context API** to maintain UI state, and FastAPI optionally uses **Redis** to track backend session state during pipeline processing.

8. API Endpoint Usage in React

This section details **how React communicates with FastAPI** to perform the four major operations of the chargeback pipeline:

1. Upload File
2. Start Pipeline
3. Poll Status
4. Fetch Results

Each code snippet is followed by a clear explanation of what it does.

🗀 8.1 Upload File Function

☑ React Code

```
const uploadFile = async (file: File): Promise<any> => {
  const formData = new FormData();

// 1. Create FormData for file

  formData.append("file", file);

// 2. Append file to FormData

  const res = await api.post("/upload", formData, {

// 3. Send POST request to /upload

    headers: { "Content-Type": "multipart/form-data" },

// 4. Required header for file uploads

  });

  return res.data; // { filename: "xyz.csv", path: "uploads/xyz.csv" }
};
```

🔍 *What it does:*

- Accepts a file (CSV) selected by the user.
- Wraps it in a `FormData` object to send as `multipart/form-data`.
- Sends the file to the FastAPI backend using Axios `POST /upload`.
- Returns the uploaded file's metadata — like name and S3 path — to the frontend.

☑ FastAPI Code

```
@app.post("/upload/")
async def upload_file(file: UploadFile = File(...)):
    file_path = f"uploads/{file.filename}"

# 1. Define S3 file path

    s3.upload_fileobj(file.file, BUCKET_NAME, file_path)

# 2. Upload file to S3 bucket

    return {"filename": file.filename, "path": file_path}
# 3. Return file metadata
```

🔍 *What it does:*

- Receives the uploaded file using `UploadFile` (a FastAPI utility).
- Stores it to S3 under the path `uploads/<filename>`.
- Returns JSON with the filename and the full S3 path.

🚀 8.2 Start Pipeline Function

☑ React Code

```
const startPipeline = async (filePath: string): Promise<string> => {
  const res = await api.post("/start-pipeline", { filePath });

//Send pipeline start request

  return res.data.pipeline_id;

// 2. Return pipeline ID
};
```

🔍 *What it does:*

- Sends a POST request to start processing on the uploaded file.
- Passes the file path (from previous `/upload`) as JSON.
- Receives and returns a unique pipeline ID to track processing.

☑ FastAPI Code

```
@app.post("/start-pipeline/")
async def start_pipeline(data: PipelineRequest):
    pipeline_id = str(uuid.uuid4())

# 1. Generate a unique pipeline ID
    asyncio.create_task(run_pipeline(data.file_path, pipeline_id))

# 2. Run pipeline asynchronously
    return {"pipeline_id": pipeline_id}

# 3. Return pipeline ID to client
```

🔍 *What it does:*

- Accepts the file path from React.
- Creates a unique ID using `uuid.uuid4()` to track the job.
- Launches the full pipeline (all 4 agents) asynchronously using `asyncio.create_task` so the API doesn't block.
- Returns the pipeline ID to React to allow polling for its status.

🔄 8.3 Poll Pipeline Status

☑ React Code

```
useEffect(() => {
  const interval = setInterval(async () => {
    const res = await api.get(`/status/${pipelineId}`);

// 1. Poll /status endpoint
    if (res.data.status === "completed") {

// 2. Check if done
      clearInterval(interval);

// 3. Stop polling
      fetchResults();

// 4. Trigger results fetch
    }
  }, 2000);

// 5. Repeat every 2 seconds

  return () => clearInterval(interval);
//Cleanup on unmount

}, [pipelineId]);
```

🧠 *What it does:*

- Starts polling `/status/{pipeline_id}` every 2 seconds.
- Waits until the pipeline status becomes `"completed"`.
- Once complete, it stops polling and triggers the `fetchResults()` function.

☑ FastAPI Code

```
@app.get("/status/{pipeline_id}")
async def get_status(pipeline_id: str):
    return {"status": pipeline_state[pipeline_id]}

# e.g., "processing", "completed"
```

🧠 *What it does:*

- Looks up the current status of the pipeline (stored in an in-memory `pipeline_state` dictionary or Redis).
- Returns the current state — such as `"processing"` or `"completed"`.

📥 8.4 Fetch Results

☑ React Code

```
const fetchResults = async () => {
  const res = await api.get(`/results/${pipelineId}`);

// 1. Fetch processed results
  setResults(res.data);

// 2. Store in frontend state
};
```

What it does:

- Makes a GET request to `/results/{pipeline_id}`.
- Receives the processed result (usually a list of JSON objects).
- Stores the results in local state for rendering in `ResultsTable`, `MetricCard`, and `ChartComponent`.

☑ FastAPI Code

```
@app.get("/results/{pipeline_id}")
async def get_results(pipeline_id: str):
    results_df = read_csv_from_s3(f"results/{pipeline_id}.csv")
# 1. Load final result from S3
    return results_df.to_dict(orient="records")
# 2. Convert DataFrame to JSON list
```

What it does:

- Reads the final `.csv` file generated by the last pipeline agent from S3.
- Converts the CSV into a Pandas DataFrame.
- Converts the DataFrame to a list of dictionaries (`records`) and returns it as JSON.

🧠 Summary of Endpoint Roles

Endpoint	Method	Used In React For	Backend Task
`/upload`	POST	File selection & upload	Upload file to S3
`/start-pipeline`	POST	Initiate pipeline run	Run async agents
`/status/{pipelineId}`	GET	Polling pipeline progress	Track current status
`/results/{pipelineId}`	GET	Fetch final results	Read from S3 and return JSON

9. Summary Table: Streamlit vs FastAPI + React

The table below compares the **original Streamlit-based architecture** with the newly migrated **FastAPI + React architecture**, across all major system features. Each point is further elaborated to provide technical insight and business value.

Feature	Streamlit	FastAPI + React
UI Layer	Built-in Streamlit widgets only	Fully custom UI with **React**, styled using **Tailwind CSS** and animated with **Framer Motion**
Backend Logic	Python logic embedded in UI scripts	REST API built using **FastAPI**, clean separation of concerns
File Upload	`st.file_uploader()` - uploads in memory	Handled using **Axios** in React and stored in **S3** via FastAPI endpoint
Progress Tracking	`st.progress()` and `st.info()` - static and simple	**Real-time polling** and **animated agent status** via React components
Agents Pipeline	Defined as inline Python functions	Implemented as **modular, testable classes**, orchestrated by a pipeline manager
Session State	`st.session_state` - in-browser state only	**Redis** (backend) + **React Context API/localStorage** (frontend) for persistence
Data Storage	File stored locally or manually in S3	All file storage and results handled via **FastAPI + S3** abstraction
Results Display	`st.dataframe`, `st.metric`, `st.line_chart` (limited)	**Highly customizable charts** using Recharts, tables with sorting and filters
Downloads	`st.download_button()`	React-based **custom download components** using Blob API and Axios responses
Integration Support	Very limited (no API layer, no third-party connectors)	Easy integration via **RESTful APIs** — connects with CRMs, notification tools, etc
Scalability	Single-threaded, not production-ready	Fully **asynchronous**, **multi-user**, and **container-ready** architecture

---

🔍 In-Depth Explanations for Each Feature

◇ 1. UI Layer

- **Streamlit:** Provides a set of built-in widgets like `st.text_input`, `st.button`, `st.dataframe`, but they are limited in design, interactivity, and mobile responsiveness.
- **React:** Offers full UI freedom with JSX, Tailwind CSS, and advanced interactivity. Animations are powered by **Framer Motion**, giving the interface a polished and modern look.

◇ 2. Backend Logic

- **Streamlit:** Backend and frontend are tightly coupled in a single Python script, making logic hard to test, scale, or reuse.
- **FastAPI:** Introduces a **clean RESTful separation**, allowing APIs to be independently tested, documented (via Swagger), and used by multiple frontends (web, mobile, etc.).

◇ 3. File Upload

- **Streamlit:** `st.file_uploader()` reads the file in memory and makes it available within session state.
- **FastAPI + React:** Files are uploaded from React using `Axios` as `multipart/form-data`, and stored in an **S3 bucket** via FastAPI's `/upload` endpoint — enabling persistent, secure, cloud-based storage.

◇ 4. Progress Tracking

- **Streamlit:** Uses basic linear `st.progress()` bars without backend awareness.
- **FastAPI + React:** Progress is tracked in real-time by polling `/status/{pipeline_id}`. React updates UI components dynamically with agent statuses and animation effects using `Framer Motion`.

◇ 5. Agents Pipeline

- **Streamlit:** All agents are simple Python functions in the same file, making them hard to test or maintain.
- **FastAPI:** Each agent (e.g., `DisputeClassifier`, `FinalDecisionMaker`) is a **separate Python class** in its own module, enabling:
    - Reusability
    - Unit testing
    - Easier debugging
    - Dependency injection

◇ 6. Session State Management

- **Streamlit:** Uses `st.session_state`, which is browser-based and not shared across users.
- **FastAPI + React:** Uses:

    - **Redis** (optional) for temporary session state on the backend
    - **React Context API** for managing state across components
    - **localStorage** for preserving state between refreshes

◇ 7. Data Storage

- **Streamlit:** Supports local file operations but lacks production-ready data storage integration.
- **FastAPI:** All file I/O is abstracted via **AWS S3**, which ensures:
    - High availability
    - Scalability
    - Data durability

◇ 8. Results Display

- **Streamlit:** Can show dataframes and charts using basic `st.dataframe()` and `st.line_chart()`.
- **React:** Offers rich visualization through:
    - **ResultsTable** – sortable, paginated tables
    - **MetricCard** – animated key metrics
    - **ChartComponent** – Recharts-based dynamic graphs

◇ 9. Download Options

- **Streamlit:** Uses `st.download_button()` to allow file export.
- **React:** Implements `DownloadButton` using:
    - Axios to fetch files
    - Blob API to create downloadable content
    - Optional CSV/JSON format selection

◇ 10. Integration Capability

- **Streamlit:** Not API-exposable. Difficult to integrate with external platforms like HubSpot, Slack, or data APIs.
- **FastAPI:** All features are built as **RESTful endpoints**, making it easy to:
    - Trigger pipelines via external services
    - Automate notifications
    - Build webhook-based automation

◇ 11. Scalability

- **Streamlit:** Runs on a single thread per session; limited concurrency; not suitable for production with many users.
- **FastAPI + React:**
    - **Asynchronous pipeline execution**
    - **Stateless frontend**
    - Supports deployment on scalable platforms (Docker, Kubernetes, AWS Lambda)

10. Appendix: Example Data Flows

This section provides real code examples and step-by-step explanations of how each part of the chargeback processing system worked in **Streamlit** vs how it now works in the **FastAPI + React architecture**.

Each flow is divided into 3 segments:

- **A. File Upload**
- **B. Pipeline Trigger**
- **C. Fetch Results**

◇ A. File Upload

□ Streamlit

```
uploaded_file = st.file_uploader("Upload CSV", type=["csv"])
if uploaded_file:
    df = pd.read_csv(uploaded_file)
    st.session_state["transactions"] = df
```

What it does:

- `st.file_uploader` opens a file dialog in the browser.
- When the file is uploaded, it is read into a pandas DataFrame and stored in session state for future processing.
- The file stays in memory only.

❁ FastAPI

```
@app.post("/upload/")
async def upload_file(file: UploadFile = File(...)):
    file_path = f"uploads/{file.filename}"
    s3.upload_fileobj(file.file, BUCKET_NAME, file_path)
    return {"filename": file.filename, "path": file_path}
```

What it does:

- Accepts the file as `multipart/form-data` from the frontend.
- Saves the file to an **S3 bucket** using `boto3`.
- Returns the file metadata (e.g., filename and S3 path) to the frontend.

💻 React

```
const uploadFile = async (file: File) => {
  const formData = new FormData();
  formData.append("file", file);
```

```
  const res = await api.post("/upload", formData);
  return res.data; // e.g., { filename: 'xyz.csv', path: 'uploads/xyz.csv'
}
};
```

What it does:

- User selects a file via a file input.
- File is wrapped in `FormData` and uploaded to FastAPI via Axios.
- Upload success triggers file-path storage and enables the next step (start pipeline).

◇ B. Pipeline Trigger

□ Streamlit

```
if st.button("Start Processing"):
    result_df = process_pipeline(st.session_state["transactions"])
    st.session_state["results"] = result_df
```

What it does:

- Button click directly triggers the pipeline (`process_pipeline`) on data in session state.
- Entire pipeline runs synchronously.
- Final results are stored back in session for display.

❁ FastAPI

```
@app.post("/start-pipeline/")
async def start_pipeline(data: PipelineRequest):
    pipeline_id = str(uuid.uuid4())
    asyncio.create_task(run_pipeline(data.file_path, pipeline_id))
    return {"pipeline_id": pipeline_id}
```

What it does:

- Accepts file path (S3 location) as JSON input.
- Generates a unique ID to track the pipeline.
- Launches the full pipeline asynchronously in the background.
- Returns the pipeline ID for polling.

💻 React

```
const startPipeline = async (filePath: string) => {
  const res = await api.post("/start-pipeline", { filePath });
  return res.data.pipeline_id;
};
```

What it does:

- React sends the file path to `/start-pipeline` using Axios.
- Stores the returned `pipelineId` in state for polling.
- UI transitions to a progress view or dashboard screen.

◇ C. Fetch Results

□ Streamlit

```
if "results" in st.session_state:
    st.dataframe(st.session_state["results"])
    st.download_button("Download", data=to_csv(), file_name="results.csv")
```

What it does:

- After processing, the results are stored in session state.
- `st.dataframe` shows them in a table.
- Download button allows export.

☸ FastAPI

```
@app.get("/results/{pipeline_id}")
async def get_results(pipeline_id: str):
    df = read_csv_from_s3(f"results/{pipeline_id}.csv")
    return df.to_dict(orient="records")
```

What it does:

- Reads the results CSV from S3 using the pipeline ID.
- Converts it to a dictionary list for JSON serialization.
- Sends it to the frontend for rendering.

💻 React

```
const fetchResults = async (pipelineId: string) => {
  const res = await api.get(`/results/${pipelineId}`);
  setResults(res.data); // stores for ResultsTable, Metrics, etc.
};
```

What it does:

- Called after polling detects pipeline is complete.
- Fetches final output from `/results/{pipeline_id}`.
- Saves the output in state and renders the table, KPIs, and charts.