Low-Level Design (LLD) - XtractIQ

1. Introduction

1.1 Purpose

The aim of this Low-Level Design (LLD) report is to outline the internal detailed design and functional behavior of the XtractIQ-Local-Run system. The system allows users to upload scanned forms, obtain structured information via an AI-driven backend, and present results in a user-friendly interface.

This guide gives a precise picture of every component's reponsibilities, internal processes, data conversions, and integration points. It is an immediate technical reference for developers, testers, and maintainers participating in the implementation or integration of this system.

1.2 Scope

This LLD is concerned with the following significant components of the XtractIQ project:

1. Frontend (React-based):
   - Components in charge of file upload and result presentation
   - State management and API communication
   - UI rendering logic and layout
2. Backend (Node.js + Python Integration):
   - File upload processing and API routing (server.js, uploadroutes.js)
   - AI integration logic (extractor/aiApiCall.js)
   - Database insertion process (insert_to_pg.py)
3. Data Contracts and Workflow:
   - Request/response schemas between frontend ↔ server ↔ backend
   - JSON formats, file handling conventions, and database payloads
     Not included in this document:
     High-Level Architecture (already discussed separately)
     Non-functional requirements (e.g., scalability, security hardening)

2. System Overview

2.1 System Description

XtractIQ-Local-Run is a smart document processing platform for the automated extraction of structured data from scanned forms and documents (e.g., PDFs or images). The platform includes a full workflow—from user file upload, through AI-powered field extraction, to the presentation of results in a web interface and optional storage in a PostgreSQL database.

Key Features:

- File Upload UI: A React-based user interface for the upload of scanned documents.
- AI-Based Extraction: Utilizes an external AI API to scrape form fields like name, date of birth, address, etc.
- Data Presentation: The results are presented in a tabular frontend interface for simple viewing.
- Backend Integration: Node.js server handles routing and communication between frontend, AI module, and database.
- PostgreSQL Storage: Optionally saves the extracted data for permanent access or additional processing.

The app is standalone and executes locally, enabling rapid deployment for demos, development, or offline scenarios.

2.2 System Context

Business Context:

Manual data entry is labor-intensive and prone to error in businesses with high volumes of paper-based or scanned forms—e.g., banks, insurance, hospitals, and schools. XtractIQ makes this process efficient by automating the extraction, saving time and effort, and enhancing efficiency.

Technical Context:

This system is a self-contained, full-stack local application with the technical limitations and integration as follows:

- Frontend: React app delivered statically through the Node.js server.
- Middleware/API Layer: Node.js Express server performs file uploads, invocation of Python modules or external AI APIs, and database communications.
- AI Extraction: AI model called through an external API that maps fields in a smart manner based on raw OCR text.
- Database: PostgreSQL is employed to store structured data for analytics or retrieval.
- Runtime Environment: Built to execute on local machines only (no cloud reliance), with only external API calls as network interaction.

3. Detailed Design

3.1 Module Descriptions

3.1.1 backend/server.js

Purpose:
Serves as the main backend entry point. Sets up the Express server, configures middleware, and connects routing logic.

Responsibilities:

- Initialize and configure the Express app
- Enable CORS and JSON body parsing
- Forward upload-related requests to the routing layer
- Serve the static React build (in production)

Dependencies:

- express, cors, body-parser
- Internal route module: routes/uploadroutes.js
- React build (served from my-react-app/build/)

3.1.2 backend/routes/uploadroutes.js

Purpose: Handles document upload, triggers AI data extraction, and calls Python scripts to store results in the PostgreSQL database.

Responsibilities:

- Accept file uploads via multer
- Save uploaded files to disk
- Call AI extractor (aiApiCall.js) with file data
- Spawn a Python script (insert_to_pg.py) with extracted fields
- Return structured results or errors to the frontend

Dependencies:

- multer, fs, path, child_process.spawn
- extractor/aiApiCall.js for AI inference
- Python script insert_to_pg.py for DB insert

3.1.3 backend/extractor/aiApiCall.js

Purpose: Performs the core AI interaction by sending uploaded file content to an external AI model or API for field-level data extraction.

Responsibilities:

- Format and send HTTP requests to AI service
- Parse and return the JSON response

Dependencies:

- fetch or axios (depending on implementation)
- AI API endpoint (external or internal service)

3.1.4 backend/insert_to_pg.py

Purpose: Handles structured data persistence by inserting extracted fields into a PostgreSQL database.

Responsibilities:

- Read JSON input via CLI
- Connect to PostgreSQL using credentials
- Build and execute SQL insert statements
- Handle DB exceptions and print logs/errors

Dependencies:

- psycopg2, json, sys, os
- PostgreSQL connection parameters via env/config

3.1.5 my-react-app/src/App.jsx

Purpose: Acts as the main React component, initializing the layout and managing the overall app state.

Responsibilities:

- Handle extracted document data via state
- Pass data to AllDocumentsTable.jsx for display
- Serve as the entry point for rendering the UI

Dependencies:

- AllDocumentsTable.jsx
- CSS styles and main.jsx

3.1.6 my-react-app/src/AllDocumentsTable.jsx

Purpose: UI component responsible for rendering a table of extracted document fields.

Responsibilities:

- Display field labels and extracted values
- Handle empty or loading states
- Present structured results in a user-readable format

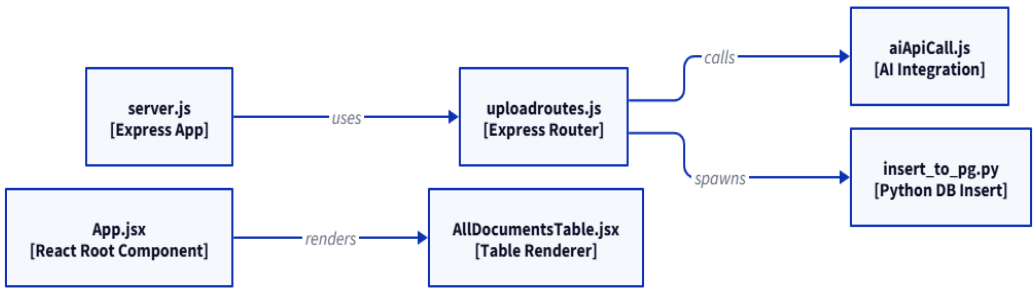
Dependencies:

- Props passed from App.jsx

3.2 Class Diagrams

The following class-level diagram illustrates the key modules and their relationships within the XtractIQ system. It is organized into frontend and backend layers, showing how components interact across the system boundary.

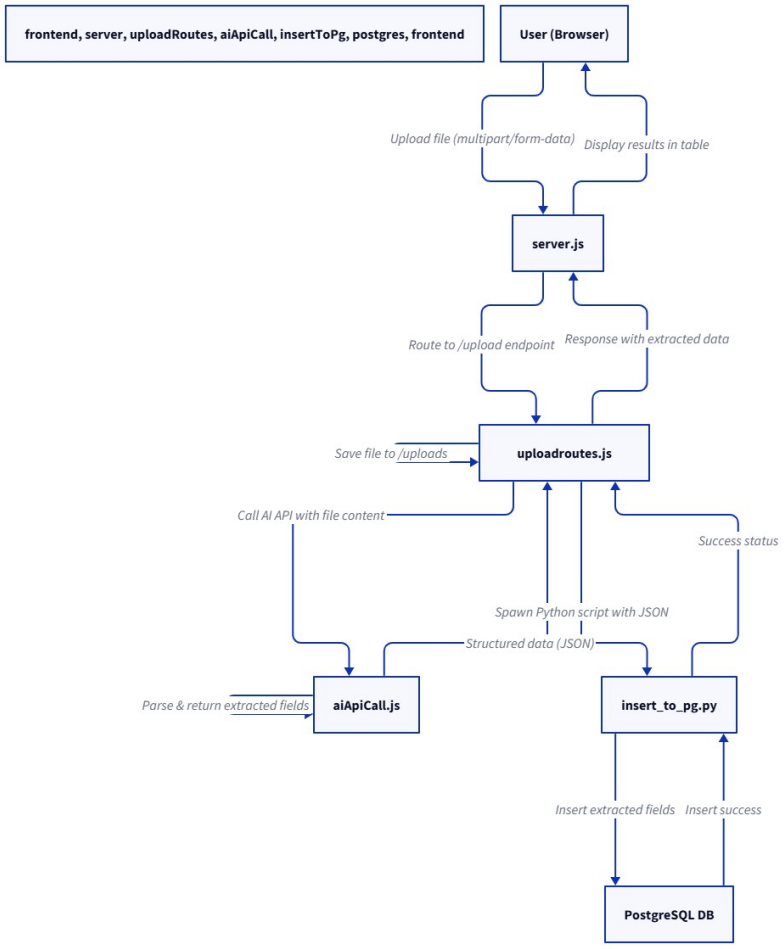
- The backend is initiated by server.js, which routes API requests to uploadroutes.js.
- uploadroutes.js coordinates the core logic: calling the AI module (aiApiCall.js) and triggering a Python script (insert_to_pg.py) for database insertion.
- On the frontend, the main React component App.jsx renders the AllDocumentsTable.jsx component to display extracted results.



3.3 Sequence Diagrams

The diagram below illustrates the sequence of interactions between system components during the core document upload and processing flow. It begins with a user uploading a scanned document and ends with the structured data being displayed on the frontend.

- The Node.js backend handles file upload via Express and triggers AI field extraction logic.
- Structured data is passed to a Python script, which persists it into PostgreSQL.
- The extracted fields are returned to the client and rendered on the interface.



4. Data Design

4.1 Data Structures

The system deals primarily with semi-structured data extracted from uploaded documents. The following key data structures are used:

1. Document Upload Object (Frontend → Backend)

{ "file": "binary_file",
"filename": "application_form_1.jpg" }

- Used In: React form and Express route /upload
- Data Type: multipart/form-data
- Purpose: Sends scanned document to backend for processing

2. Extracted Fields Object (AI Output)
{ "full_name": "Ravi Kumar",
"father_name": "Shiv Kumar",
"dob": "1995-08-19",
"phone": "9876543210",
"email": ravi@example.com,
"account_type": "Savings",
"nominee": "Father",
"address": "12, Gandhi Street, Chennai" }
   - Used In: AI API response and passed to insert_to_pg.py
   - Data Type: JSON (Python dictionary / JS object)
   - Purpose: Represents the extracted structured data

4.2 Database Design

The system does not adhere to a rigid relational schema. Rather, database design is dynamic and changes according to the extracted key-value pairs from every document.

- One table called `documents` stores field data in structured form.
- Column names are dynamically generated through keys available in the AI-extracted output.
- New columns are introduced at runtime as required with `ALTER TABLE` statements.
- All values are represented as `TEXT` to ensure consistency and prevent type mismatches.

This allows the platform to have numerous form templates and arbitrary field structures without necessarily having to do schema migrations or predefining columns in tables.

4.3 Data Flow

Data flow within the system is sequential pipeline from user input to database insertion and UI rendering:

- The user uploads a scanned document (image or PDF) through the frontend.
- The file is passed to the backend via a POST request to the /upload endpoint.
- The backend stores the file locally within the /uploads directory.
- The file saved is sent to the AI module, where structured data fields are extracted.
- The key-value pairs extracted are sent to a Python script to insert data into the PostgreSQL database.
- The database schema is dynamically updated according to the extracted keys, and data is inserted into the documents table.
- The data extracted is sent back to the frontend and displayed in tabular form.

This sequential data flow allows for modular processing and separates each step for maintainability and future extendibility.

5. Interface Design

5.1 User Interface

The frontend for XtractIQ is developed with React.js and has the following primary components:

Screen Layouts:

- Upload Screen (App.jsx)
- Results Screen (AllDocumentsTable.jsx)
- Table layout with dynamically determined columns based on extracted fields
- Scrollable view to accommodate large field sets
- Real-time visualizing of extracted values upon upload

Behavior:

- After file upload, the system extracts automatically without an extract button.
- Shows loading indicators while API processing.
- Errors (e.g., AI call failure or DB insertion) are indicated as toast messages or alert boxes.
- The table is refreshed on a successful upload to display the data of the new document.

5.2 External Interfaces

The system exposes a single backend interface for document processing.

Endpoint: POST /upload

- URL: http://localhost:5000/upload
- Method: POST
- Content-Type: multipart/form-data
- Payload:
- file: uploaded document (image or PDF)

Successful Response:

json
CopyEdit

```
{
  "message": "Extraction complete",
  "data": {
    "full_name": "Arun Raj",
    "dob": "1990-03-15",
    "email": "arun@example.com"
  }
}
```

6. Security Design

6.1 Security Measures

The following safety controls are in place to safeguard the system against typical threats and safe handling of documents:

- File Validation
  Uploaded documents are validated by MIME type and file extension to avert execution of hostile files.
- Input Sanitization
  User input and extracted values are processed with safe string handling and escaping functions to avert injection attacks.
- Dynamic SQL Safety
  Column names and values sent to the database from extracted data are sanitized. While dynamic column creation is implemented, attention is paid not to inject SQL by escaping identifiers appropriately.
- Error Handling and Logging
  Server-side errors are logged without the client seeing stack traces or sensitive data. User-facing errors are formatted consistently.
- Directory Access Control
  Uploaded files are held in a controlled directory (/uploads) and the directory is not accessible from the public frontend.

7.Error Handling and Logging

7.1 Error Handling

The system provides minimal error handling throughout the backend and AI extraction pipeline to provide graceful degradation and user feedback.

Backend (Node.js):

- Every principal route (e.g., /upload) has try-catch blocks for handling file processing, AI calls, and DB execution errors.
- Custom error messages are returned to the frontend in the following format:
  { "error": "AI extraction failed"}
- Python Script (insert_to_pg.py): Wrapped in try-except to catch: JSON decode errors SQL syntax errors DB connection errors
- Errors are output to standard output and make the process return an error status.

Frontend (React):

- Backends failed responses are caught in .catch() handlers and presented to the user through alerts or UI messages.

Recovery:

- Failed uploads are not stored to the database.
- The table is just updated on successful extract + insert.
- There is no current retry or rollback mechanism.

7.2 Logging

The backend employs simple console.log() calls to track significant events like file uploads, AI answers, and database insertions.

Logs are output solely to standard output. There is no permanent storage or log level system with structure installed.

Log messages serve mainly for debugging and are not categorized by severity (e.g., INFO, ERROR).

8. Performance Considerations

8.1 Performance Optimization

- File uploads are processed through multer in streaming mode to prevent memory overflow.
- AI extraction is invoked asynchronously to prevent blocking the main thread.
- No client-side caching or prefetching is done in this version.

9. Assumptions and Dependencies

9.1 Assumptions

- Uploaded files are either proper image files (.jpg, .png) or PDFs.
- The AI model returns always well-formed JSON with field-value pairs.
- There will be only one document processed per request.
- One document at a time will be uploaded by the users and they will wait for the outcome before re-submission.
- The local PostgreSQL server is present and available to use during script execution.

9.2 Dependencies

Frontend:

- React.js using Axios for HTTP calls.

Backend:

- Node.js using Express, Multer, and child_process for calling Python scripts.
- Python: Python using psycopg2 and normal JSON handling.

Database:

- PostgreSQL (locally hosted).

AI Extraction:

- API (Computer Vision from azure and Groq API).

Local File Storage:

- Files temporarily stored under /uploads.