Technical Manual

Student 1 – Lorena Gomez – 21734359

Student 2 – Darragh Manning - 21506373

Supervisor - Tomas Ward

Project Name – TimelineXtract

Last edited on: 02/05/2025

Abstract - TimelineXtract is a system designed to automate the extraction of questionnaire schedules from complex clinical trial protocols. Clinical trial PDFs are often lengthy, inconsistently structured, and difficult to navigate, making manual data extraction slow and error prone. Our solution combines Adobe PDF Extract API for precise document structure recognition, GPT-4 for semantic data extraction, Django and React.js for processing and presentation, and MongoDB for secure storage.

The project overcame major challenges such as inconsistent table formats, varying questionnaire names, and aligning data from different extraction sources. Through extensive prompt engineering, vector similarity matching, and real world testing with protocols from ClinicalTrials.gov, TimelineXtract delivers clean, structured outputs that streamline clinical research.

Table of Contents

Introduction

Clinical trial protocols are formal documents that describe how a clinical study will be conducted. They include essential details such as the study’s objectives, design, methodology, statistical considerations, and the schedule for data collection.

A core component of these protocols is the *Schedule of Assessments*, which outlines **what is measured** (e.g., lab tests, physical exams, or patient reported questionnaires), **when it is measured** (specific visits, weeks, or months), and **how it is measured** (the type or format of questionnaire used).

Questionnaires used in clinical trials often capture patient experience, symptoms, or quality of life. These are commonly classified as:

- **Patient Reported Outcome (PRO)**: which are answered directly by the patient.
- **Patient Reported Experience Measure (PREM)**: focused on patient satisfaction or experience.
- **Clinician Reported Outcome (ClinRO)**: filled in by a healthcare professional.

The schedule defines the timepoints at which each questionnaire should be completed. For example: a PRO like the EQ-5D might be administered at baseline, Month 3, and Month 12.

TimelineXtract extracts this structured schedule and links each questionnaire to its timepoints.

Motivation

Clinical trials are essential for developing new treatments and improving healthcare, but working with the protocols that support them can be a real challenge. These documents are often long, complex PDFs filled with dense and detailed information. And while every part of a protocol matters, one of the trickiest sections to find and work with is the schedule of questionnaires. This schedule of questionnaires tells the patient what gets asked, when, and how often.

What makes this so frustrating is that no two protocols are the same. Each one can be structured completely differently, with sections named in unique ways, tables placed in unexpected spots, or information spread across appendices and footnotes. Sometimes the questionnaire schedule is buried deep in a table under a vague heading like "Assessments" or "Visit Plan"; other times, it might be hidden in a wall of text. There's no standard, and that inconsistency makes things even harder for researchers, especially when they're dealing with dozens or hundreds of these documents.

Right now, if someone wants to know which questionnaires are used in a trial, when they’re given, and what type they are (like PROs, PREMs, or ClinROs), they often have to scroll through the entire document, locate complex tables, and manually pull out the information. It’s time consuming, repetitive, and error-prone, especially when dealing with large numbers of trials. For researchers, clinical coordinators, and even data scientists trying to compare studies, this creates a big time consuming puzzle.

We created *TimelineXtract* to solve this problem. The goal of this project is to automatically extract questionnaire data and their timelines from clinical trial protocols using AI. By combining tools like GPT-4 for understanding context, PDF processing for extracting tables, and MongoDB for storing structured outputs, the system turns these difficult to read documents into clean, searchable information.

This project is also designed with real users in mind. It includes Google login restricted to DCU emails, which helps keep data secure and ensures that only authorised users (like students or researchers) can access the platform.

What motivated us most was the idea of saving people time and helping them focus more on analysis rather than manual data entry. In a world where data is everything, tools like this can make clinical research faster, smarter, and more reliable.

Research

Understanding Clinical Trial Protocols

Before we could build a system to extract clinical trial information, we first needed to truly understand the complexity of clinical trial protocols themselves. These documents guide how studies are conducted, but they are notoriously dense, inconsistent, and formatted differently depending on the sponsor, therapeutic area, or regulatory agency.

To familiarise ourselves with real world protocols, we extensively studied examples from publicly available databases such as ClinicalTrials.gov. This allowed us to observe firsthand:

- How questionnaires are referenced at different points in the protocols.
- How schedules of assessments are sometimes hidden in tables, while other times mixed into paragraphs.
- How terminology can vary significantly between documents (e.g., "Baseline Visit" vs. "Visit 1").
- The lack of a standardised structure across different clinical trials, even when they target similar diseases.

This web-based research was critical in shaping both our understanding of what data needed to be extracted and how messy the extraction process could become. It also guided the decisions behind the choice of tools and model tuning.

Why Adobe PDF extract API?

There are several tools available for processing PDFs, such as PyMuPDF, PDFMiner, and Tika. However, these often struggle with complex, heavily formatted clinical trial documents. These files can include multicolumn layouts, embedded tables, and inconsistent headers, all of which cause basic PDF parsers to break structure or misread data.

We chose the Adobe PDF Extract API because it consistently outperformed other tools in preserving the layout of the documents.

Adobe PDF Extract API was chosen because:

- It preserved the semantic structure of the PDF: headers, paragraphs, and tables were correctly detected and hierarchically organised.
- Its output format (CSV) made it easy to locate tables and perform downstream processing.
- It was cloud hosted and scalable, allowing faster and more consistent parsing.

This structural integrity was crucial for isolating sections like the schedule of assessments or questionnaire tables, which are often hidden deep within lengthy protocols.

Why GPT-4?

Once the content was reliably extracted, we needed a language model that could interpret domain specific text, understand context, and summarise complex information. We evaluated several large language models (LLMs), including open source ones like BERT, BioBERT, and T5, as well as commercial offerings.

We evaluated several alternatives:

- BERT, BioBERT, and T5 models: good at classification and question answering, but not great at structured generation or multihop reasoning (needed to infer schedule links across sections).
- Smaller LLMs (Alpaca, LLaMA): not powerful enough without major fine tuning.

GPT-4 outperformed all of these because:

- It could understand and generate structured JSON outputs from unstructured text.
- It handled long, messy paragraphs and could infer questionnaire details hidden in complicated text.
- It followed carefully engineered prompts with much higher consistency.

Research shows that prompt-based methods, especially with LLMs like GPT-4, are particularly valuable in healthcare NLP applications. In a recent survey, Tran et al. [4] noted that prompt engineering enables non-technical users to adapt models to new use cases without retraining. Similarly, Shin et al. [5] found that well crafted prompts allow LLMs to outperform fine-tuned smaller models on many downstream tasks in software and medical domains.

We also drew inspiration from prompt design frameworks like those introduced by White et al. [3], using a pattern based prompt strategy to extract questionnaire content, patient timelines, and treatment information.

Together, Adobe's semantic PDF parser and GPT-4's language capabilities allowed us to bridge the gap between structured document formats and natural language reasoning, enabling the reliable extraction of critical data from diverse and often convoluted clinical trial protocols.

Challenges in Clinical Trial Protocol Extraction

- Clinical trials protocols can exceed 100 pages with no standardised section names.
- Questionnaires are not always listed explicitly; sometimes they are described narratively within timelines.
- Tables can span multiple pages, or be embedded in unrelated sections.
- Terminology inconsistency (e.g., "FACT-P" vs. "Functional Assessment of Cancer Therapy-Prostate") added extraction complexity.

Prior work:

- Rosemblat et al. [1] used regular expressions and simple NLP to extract purpose statements, but this approach is too brittle for our target (tables, schedules, timelines).
- CT-BERT introduced by Liu et al. [2], showed that fine-tuned BERT models could extract eligibility criteria, but again, fine tuning large models is computationally expensive and inflexible compared to GPT prompt engineering.

Advancements in Prompt Engineering with GPT Models

The emergence of large language models (LLMs) like GPT-4 has revolutionised NLP tasks, enabling a more elegant understanding and generation of human-like text. When using large language models like GPT-4, developers can either:

- **Fine-tune** a model: train it on domain specific data. This requires substantial data and compute resources.
- **Prompt** a model: design smart instructions and context examples to guide the model's output.

We chose prompt engineering because it avoids retraining the model, is faster to iterate on, and works well for diverse protocols.

Prompting allowed us to adapt GPT to clinical protocols by giving it structure, examples, and domain context. A systematic survey by White et al. [1] presented a catalog of prompt patterns to enhance interactions with ChatGPT, emphasising the importance of prompt design in obtaining desired outputs.

In the healthcare domain, prompt engineering has shown promise in improving the efficiency and accuracy of LLMs. A study published in the Journal of Medical Internet Research [4] discussed the significance of prompt engineering for medical professionals, suggesting that well designed prompts can enhance the utility of LLMs in clinical settings.

Moreover, research by Shin et al. [5] compared prompt engineering techniques with fine tuning approaches for automated software engineering tasks, finding that task specific prompting with GPT-4 could outperform fine-tuned models in certain scenarios.

Design

The system is designed to provide a seamless experience for clinical researchers to upload complex clinical trial protocols and receive structured outputs, such as questionnaire schedules and metadata, extracted using AI and robust PDF parsing. The architecture consists of several modern technologies working together in a modular fashion:

System Architecture Overview

Frontend:

- **React.js** handles the user interface, providing pages for login, file upload, result viewing, and error handling.
- **Google OAuth** is integrated in the frontend, allowing only DCU users to authenticate.

Backend:

- **Django** serves as the backend framework, managing API routing, request validation, and integration with third party services.
- Secure endpoints are exposed to accept authenticated POST requests from the frontend.
- A strict check is enforced to allow only @mail.dcu.ie addresses for system access.

AI Services:

- **Adobe PDF Extract API** is used to extract structured content (tables) from PDF files.

- **GPT-4** is used for semantic understanding and question answering, specifically trained with prompt engineering to identify questionnaire sections and timelines.

Database:

- **MongoDB** stores users, uploaded PDF metadata, extracted data, tables, and results. BSON's flexible schema supports the complexity of extracted data.

Authentication:

- **Google OAuth** is used to restrict access to verified DCU accounts only.
- **JWT Tokens** manage secure sessions and requests between the client and server.

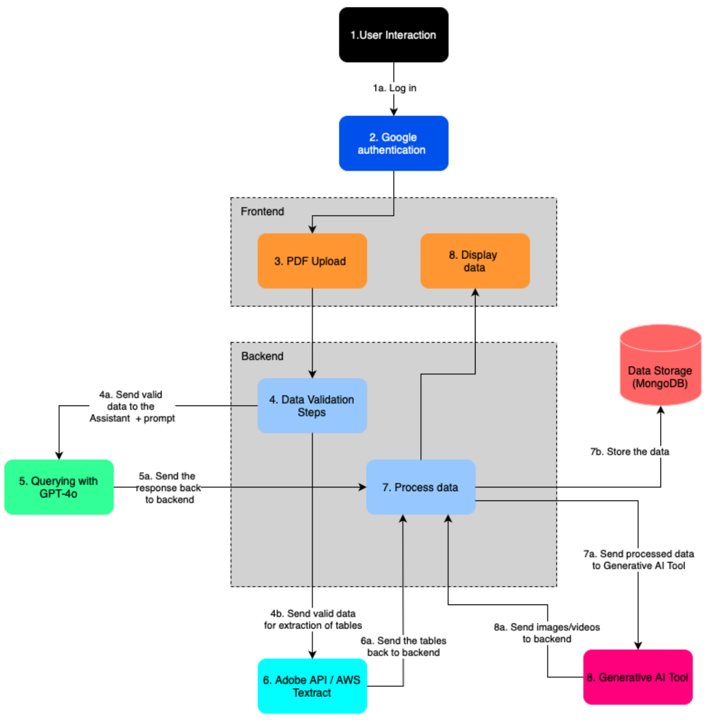
Data Flow

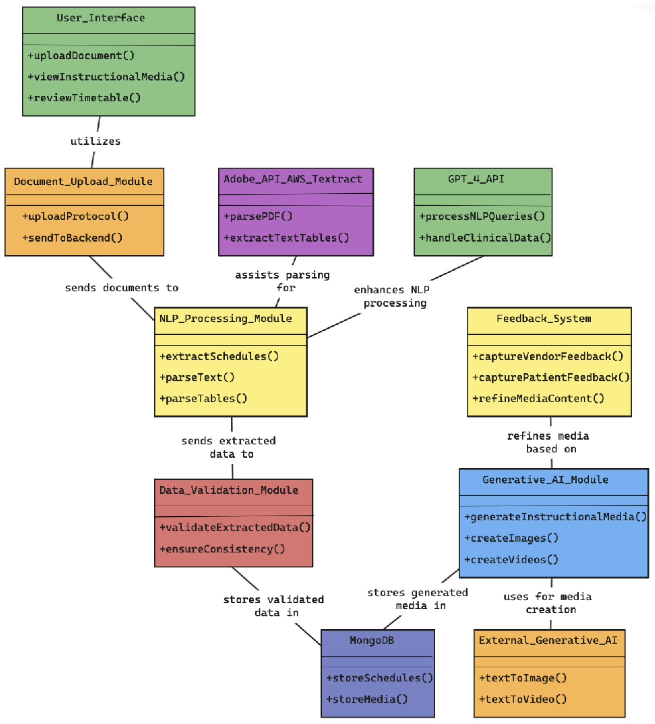
1. **User Authentication:**
   - A user logs in via Google.
   - Only users with an @mail.dcu.ie address are granted access.
2. **Upload PDF:**
   - The authenticated user uploads a PDF via the React UI.
   - The PDF is sent as a multipart/form-data POST request to the Django backend.
3. **PDF Processing:**
   - The file is stored temporarily.
   - Adobe API processes the file and extracts structured content.
   - GPT-4 is prompted with specific queries to extract questionnaire data.
4. **Data Matching:**
   - Extracted tables are matched with generated schedules to improve precision.
   - Matching results are merged into a unified JSON format.
5. **Database Storage:**
   - The file metadata, extracted data, and query results are saved in MongoDB.
6. **Display Result:**
   - The JSON output is returned to the frontend.

- The user is redirected to the display page to view the final structured result.

Security and Access Control

- **Email domain restriction:** Only users with a verified @mail.dcu.ie email can use the app.
- **Token-based access:** OAuth access tokens are validated before allowing uploads.
- **File validation:** Only PDF files are accepted. Malicious input is filtered server side.
- **CSRF protection:** All routes are CSRF exempt and use HTTPS only in production.


1.User Interaction
1a. Log in
2. Google authentication
Frontend
3. PDF Upload
8. Display data
Backend
4. Data Validation Steps
Data Storage (MongoDB)
4a. Send valid data to the Assistant + prompt
5. Querying with GPT-4o
5a. Send the response back to backend
7. Process data
7b. Store the data
7a. Send processed data to Generative AI Tool
4b. Send valid data for extraction of tables
6a. Send the tables back to backend
8a. Send images/videos to backend
6. Adobe API / AWS Textract
8. Generative AI Tool



Implementation

The implementation of the *TimelineXtract* system brings together several technologies to deliver a smooth, secure, and intelligent way to extract clinical trial data from complex PDF protocols. The application is built using a modular design with Python on the backend and React.js on the frontend.

Key Files and Their Roles

Backend (Python/Django):

- views/upload_pdf.py: Handles the upload pipeline from receiving the PDF to returning a JSON with extracted data.
- utils/pdf_processing.py: Coordinates calls to the GPT model and Adobe PDF Services for text and table extraction.
- utils/match_questionnaires.py: Matches extracted questionnaire tables with timelines based on similarity thresholds and conditions.
- services/mongodb/: Contains modules for inserting and retrieving data from MongoDB (user.py, pdf.py, query.py, output.py, etc.).

Frontend (React.js):

- App.js: Manages routing and layout for all pages (upload, login, display, error).
- pages/Upload.js: Lets users upload a PDF and manages the request/response cycle.
- pages/DisplayResponse.js: Renders the final JSON data in a structured, user friendly format.
- components/Navbar.js: Displays navigation and manages user state.
- pages/Login.js: Handles Google Sign In and stores the user token and info.

RESTful API Endpoints

- POST /srcExtractor/upload/: Accepts the uploaded PDF, processes it, and stores the result.
- Error handling: Each major step in the backend returns an informative JSON error message when something goes wrong (e.g., file missing, user not authenticated, GPT failure).

MongoDB Collections

Data is stored across five key collections to maintain structure and flexibility:

- **Users**: {_id, username, email}
- **PDFs**: {_id, user_id, file_name, file_object (compressed)}
- **Queries**: {_id, user_id, pdf_id, response (JSON)}
- **Tables**: {_id, pdf_id, table_data, classification}
- **Output**: {_id, query_id, summary, metadata}

We used bson.ObjectId to uniquely identify and link documents across collections.

File Compression

To avoid storing large raw PDF files uncompressed, we used Gzip via the gzip Python module. A helper function compresses the PDF binary before inserting it into MongoDB's PDFs collection under the file_object field. This improves storage efficiency without compromising retrieval speed.

Styling and UI Libraries

Some key UI touches:

- Clean buttons and forms with hover/focus states
- Conditional rendering during file upload to show a loading spinner
- Rounded borders and soft color palette matching our clinical/professional theme

Security and Access

- **Google OAuth 2.0** was used for authentication. Only users with an @mail.dcu.ie email domain are allowed access to the system. This was enforced both on the frontend (client-side validation) and backend.
- **JWT Tokens** were issued using rest_framework_simplejwt, stored client-side, and attached to future requests for validation.
- **CORS** was configured using Django CORS Headers to allow safe communication between React and Django servers.
- **CSRF Protection** was managed by Django's built-in middleware, with @csrf_exempt used only in cases where authenticated POST requests were required without form submission.

Prompt Engineering with GPT-4

Extracting the correct set of questionnaires, their types (e.g., PRO, PREM, ClinRO), and schedules from clinical protocols was one of the biggest challenges. These details are deeply embedded in complex, variable, and sometimes vague language.

To solve this, we invested over a month exclusively on prompt engineering, drawing from both academic research and industry practices:

- We studied research like [Reynolds & McDonell, 2021] and OpenAI best practices to identify how prompt structure, system instructions, and example-driven prompting affected performance.
- 50 clinical trial protocols were used for testing.
- Prompts were designed to be:
  - Explicit about what to extract (e.g., longName, shortName, schedule).
  - Aware of the document context (mentioning it's a clinical trial protocol).
  - Resilient to variation in language, section titles, and formatting.

Prompt Goals:

- Extract information like Study Title, Sponsor, Therapeutic area, etc.
- Extract all relevant questionnaires, not just the first or most obvious one.
- Correctly classify them into PRO, PREM, ClinRO, or None.
- Capture questionnaire schedules.

This process of refining prompts and validating them with actual protocol outputs dramatically improved both recall and precision, and led to a robust system that adapts well to a wide variety of real world documents.

Example Final Prompt For General Information Extraction

Please analyse the attached file.

Extract the following structured information from the provided clinical trial protocol. The output should be in JSON format, including the required fields as described below. Ensure the extraction is accurate and based on the described locations in the protocol.

Required Fields:

0. Project Title: The title of the protocol. Extract the protocol name, typically found on the first pages of the protocol.
1. Sponsor: The organisation sponsoring the clinical trial. Extract the sponsor's name, usually found on the first page of the protocol.
2. Study Number: The unique identifier for the protocol. Extract the protocol number, found on the first page beneath the protocol objective.

3. Protocol Version and Date: Extract the version and date of the protocol, located on the first page beneath the protocol objective.
4. Study Title: The title of the clinical trial study. Extract the full protocol name, typically found on the first page of the protocol.
5. Phase: The phase of the clinical trial (e.g., Phase I, Phase II, Phase III).
6. Therapeutic Area: The therapeutic area or medical condition being studied.
7. Number of Patients: The anticipated number of patients to be enrolled in the study.
8. Number of Sites: The number of clinical sites participating in the study.
9. Indication: Extract the disease being treated, typically found in the aim of the study section.
10. Duration of Treatment: Extract the duration of treatment (e.g., weeks, days, or years), found in the schema, schedule of activities, or study design sections.
11. Schedule of Assessments: Extract the table number reference (and section number) for the schedule of assessments.

Instructions for Response:
1. Extract each piece of information accurately based on its location in the protocol.
2. Format the output in JSON with field names matching the structure below.
3. If a field is not mentioned or available in the protocol, state "Not Available."

Example Output:
```
{
        "Sponsor": "<<Sponsor Name>>",
                "Study Number": "<<Study Number>>",
        "Protocol Version and Date": "<<Version and Date>>",
        "Study Title": "<<Full Protocol Name>>",
                "Protocol Title":"<<Protocol Title>>",
                "Phase": "<<Phase>>",
                "Therapeutic Area": "<<Therapeutic Area>>",
                "Number of Patients (if known)": "<<Number of Patients>>",
                "Number of Sites (if known)": "<<Number of Sites>>",
                "Duration of Treatment": "<<Number of Weeks, Days, or Years>>",
                "Schedule of Assessments": "<<Table Number Reference and Section Number>>"
}
```

Final Note:
Once the response is generated, double-check the extracted information for accuracy and completeness.

Prompts as Code: Version Control and Iteration

We treated prompt engineering as software development:

- Prompts were saved in .txt and .json files with version numbers.
- We added change logs explaining what changed and why (e.g., added new examples, made instruction more explicit).

This disciplined approach let us:

- Track what prompt versions led to the best results.
- Revert changes when performance dropped.

- Collaborate safely between team members.

Matching with Vector Similarity

One of the most innovative parts of this project is the cross validation of data between Adobe API and GPT-4 using vector similarity.

Step by step Matching Pipeline:

1. **Adobe API Table Extraction**:
   - Adobe PDF Services extracts multiple tables from the uploaded PDF protocol.
   - These tables are then converted from raw CSV format into structured JSON for easier querying and comparison.
2. **Schedule of Events Identification**:
   - Among the many tables extracted, the system identifies which table represents the *Schedule of Events*.
   - This is done through keyword matching (e.g., “Timepoint”, “Visit”, “Procedure”) and structural heuristics (e.g., row/column patterns).
3. **GPT-4 Questionnaire Extraction**:
   - GPT-4 is used to read the unstructured text in the protocol and extract questionnaire names, types (e.g., PRO, ClinRO), and their associated schedules.
4. **Similarity Matching**:
   - To verify the accuracy of the extracted questionnaire schedule, the procedures from the *Schedule of Events table* (from Adobe) are vectorised using sentence embeddings.
   - Similarly, the questionnaires extracted by GPT-4 are also converted into vectors.
   - Using similarity thresholds, we compare the procedures (Adobe) with questionnaire entries (GPT) to find strong matches.

A match is accepted if:

- The similarity score exceeds a threshold (e.g., 0.8)
- The matched items align semantically based on predefined rules (e.g., acronyms, common medical terminology)
- The timepoints and frequency match a defined pattern (e.g., “M0, M3, M6”)

Only questionnaires that pass the similarity check and satisfy all matching conditions are added to the final output JSON file. This ensures high accuracy, avoids duplication, and bridges the gap between different formats and language styles used across protocols.

Continuous Integration & Deployment (CI/CD) Pipeline

To ensure code quality, reliability, and smooth deployment, we implemented a CI/CD pipeline using GitLab CI/CD. This pipeline was designed to automatically test, lint, and deploy the system every time new code was pushed to the repository.

1. **GitLab CI/CD Overview**
    a. We created a .gitlab-ci.yml configuration file at the root of our repository. This file defined the stages and jobs for our pipeline:
        i. **Test Stage**: Runs all unit tests using python3 manage.py test srcExtractor/tests
        ii. **Lint Stage**: Checks code style consistency using flake8
        iii. **Deploy Stage**: Automatically deploys the project to our server upon successful tests and linting

2. **Pipeline Stages**
    a. **Unit Testing:** We used django tests to validate our backend components:
        i. MongoDB insert and retrieval logic
        ii. PDF upload handling
        iii. GPT and Adobe API integrations (mocked or isolated)
        iv. File compression and format validators
        v. These tests were automatically triggered in the **Test** stage of the pipeline:

```
test:
  stage: test
  script:
    - pip3 install -r requirements.txt
    - python3 manage.py test srcExtractor/tests
```

    b. **Code Style Linting:** To maintain a clean and consistent codebase, we used flake8. Any major formatting issues or bad practices were flagged during this stage:

```
test:
  stage: test
  script:
    - pip install flake8
    - flake8 .
```

c. **Deployment:** Once code is merged into the main branch:

i. The repository is mirrored to GitHub for easier integration with Render and Vercel

ii. The backend is run on Render and the Frontend is run on Vercel.

iii. Render automatically pulls the latest code from GitHub mirrored repository.

iv. Builds an environment configuration (e.g., build command, start command, environment variables) is managed through the Render dashboard.

v. Deployment happens seamlessly without needing manual intervention or shell scripts.

vi. Vercel also updates automatically and builds an environment through its own dashboard.

vii. On success, the updated app is live immediately at our Vercel link.

3. **Benefits of Our Pipeline**

   a. **Automated Testing**: Ensures our code is always production ready

   b. **Early Bug Detection**: Bugs are caught before merging

   c. **Code Quality Enforcement**: Cleaner, more maintainable codebase

   d. **Fast, Repeatable Deployments**: Saves time and avoids human error

Sample Code

This section showcases core parts of the *TimelineXtract* system, focusing on the interaction between the frontend and backend, PDF processing, GPT prompting, and database operations.

1. Upload Handler (React + Django)

React.js – Upload Button & Request (frontend/src/pages/Upload.js)

```
function Upload() {
 const [file, setFile] = useState(null);
 const [loading, setLoading] = useState(false);
 const navigate = useNavigate();
 const handleUpload = async () => {
   if (!file) return alert("Please select a PDF");

   setLoading(true);
   const formData = new FormData();
   formData.append("pdf_file", file);

   const user = JSON.parse(localStorage.getItem("user"));
   if (user) {
     formData.append("email", user.email);         // from google.auth
   }

   try {
         const  response  =  await  axios.post("http://127.0.0.1:8000/srcExtractor/upload/",
formData);
     localStorage.setItem("response", JSON.stringify(response.data));
     navigate("/response");
   } catch (error) {
     // Extract and display the error from the backend
        const  errorMessage  =  error.response?.data?.error  ||  "Upload  failed.  Please  try
again.";
     alert(errorMessage);
     setLoading(false);
   }
 };
```

Django – Upload Endpoint (srcExtractor/views/upload_pdf.py)

```
@csrf_exempt
def upload_pdf(request):
   if request.method != 'POST':
       return JsonResponse({"message": "Waiting for a PDF to be uploaded."}, status=200)

   try:
       # Step 1: Validate Request
       pdf_file = request.FILES.get('pdf_file')
       if not pdf_file:
                 return  JsonResponse({"error":  "No  PDF  file  provided  in  the  request."},
status=400)

       temp_file_path = save_temp_pdf(pdf_file)
       data = {
           "file_name": os.path.basename(temp_file_path),
           "temp_file_path": temp_file_path
       }

       handle_pdf_upload_response = handle_pdf_upload(temp_file_path)
       if "error" in handle_pdf_upload_response:
           return JsonResponse(handle_pdf_upload_response, status=400)
```

```
        # Step 2: Authenticate User
        email = request.POST.get("email")

        if not email:
            return JsonResponse({"error": "Missing email."}, status=400)

        data.update({"email": email})

        # Step 3: Add user to the database
        """ Code for the upload of user to database """
        user_status = add_user(data)

        if "error" in user_status:
            return JsonResponse(user_status, status=500)

        data.update({"user_id": user_status.get("user_id")})
        # Further steps: Adobe -> GPT-4o -> Matching -> Save
```

2. PDF Extraction with Adobe API

# srcExtractor/utils/pdf_processing.py

```
def extract_and_classify_tables(pdf_file_path, pdf_file_name):
    try:
        tables_result = extract_tables(pdf_file_path, pdf_file_name)
        if "error" in tables_result:
            return tables_result

                                                                      valid_files          =
classify_all_tables_in_folder(f"table_extraction_output/extracted_{pdf_file_name}/tables")

        if not valid_files:
            return {"error": "No Schedule of Events table found in the protocol."}

        if "error" in valid_files:
            return valid_files

        # Convert valid tables to JSON
        if valid_files["success"]:
                                        convert_valid_files_to_json(valid_files["success"],
f"table_extraction_output/json_{pdf_file_name}/")

        return {"valid_files": valid_files["success"]}
    except Exception as e:
        logging.error(f"Error extracting and classifying tables: {e}")
        return {"error": "Failed to extract or classify tables"}
```

This pipeline uses Adobe's API to extract tables from raw PDF pages, then classifies them based on keywords and structure.

3. GPT-4 Questionnaire Extraction

# srcExtractor/services/gpt/questionnaire_extraction.py

```python
def send_to_chatgpt(pdf_file_path):
   protocol_prompt = PROMPTS["protocol_extraction"]
   questionnaire_prompt = PROMPTS["questionnaire_extraction"]

   api_key = os.getenv('CHATGPT_API_KEY')

   if not api_key:
         return {"error": "API key not found. Please set the CHATGPT_API_KEY environment
variable."}

   openai.api_key = api_key

   try:
       # Step 1: Upload file
       with open(pdf_file_path, "rb") as file:
           file_response = openai.files.create(file=file, purpose='assistants')

       print(f"Starting OpenAI extraction for file: {pdf_file_path}")

       # Step 2: Extract protocol information
       protocol_content = openai_run(openai, pdf_file_path, file_response, protocol_prompt)

       if not protocol_content:
           return {"error": "Failed to extract protocol information."}

       # Step 3: Extract questionnaires
               questionnaire_content = openai_run(openai, pdf_file_path, file_response,
questionnaire_prompt)

       if not questionnaire_content:
           return {"error": "Failed to extract questionnaire information."}

       # Step 4: Merge and save the extracted data
       extracted_protocol = extract_json(protocol_content)
       extracted_questionnaires = extract_json(questionnaire_content)
       merged_data = extracted_protocol[:-5] + ',' + extracted_questionnaires[1:]

       save_status = save_merged_data_to_json(merged_data, pdf_file_path)

       combined_response = protocol_content + ' ' + questionnaire_content

       if "error" in save_status:
           return {"error": save_status}
       return {"response": combined_response}
   except Exception as e:
       return {"error": f"Failed to process PDF with GPT-4o: {str(e)}"}
```

We carefully crafted the prompt and used a low temperature to ensure deterministic and accurate outputs.

4. MongoDB Insert & Query

# srcExtractor/services/mongodb/pdf.py

```python
def add_pdf(data):
   pdf_collection = mongodb.get_collection("PDFs")
   user_collection = mongodb.get_collection("Users")

   user_id = data.get('user_id')
   file_name = data.get('file_name')
   file_object = data.get('temp_file_path')

   # Validation
   if not all([user_id, file_name, file_object]):
       return {"error": "Missing required fields: user_id, file_name, or file_object."}

   if not mongodb.is_valid_object_id(user_id):
       return {"error": "Invalid user_id format."}

   if not mongodb.is_user_id_valid(user_id, user_collection):
       return {"error": "User ID does not exist in the Users collection."}

   if not isinstance(file_name, str) or not file_name.strip():
       return {"error": "file_name must be a non-empty string."}

   if len(file_name) > 255:
       return {"error": "file_name must be 255 characters or less."}

   unique_file_name = get_unique_file_name(file_name, pdf_collection)
   compressed_data = compress_file(file_object)

   if "error" in compressed_data:
       return compressed_data

   new_pdf = {
       "user_id": ObjectId(user_id),
       "file_name": unique_file_name,
       "file_object": compressed_data
   }
   try:
       result = pdf_collection.insert_one(new_pdf)

       return {
           "message": f"PDF '{unique_file_name}' uploaded successfully.",
           "pdf_id": str(result.inserted_id)
       }
   except Exception as e:
       return {"error": f"Failed to upload PDF metadata: {str(e)}"}
```

Problems Solved

Throughout the development of *TimelineXtract*, several technical and usability challenges were encountered and addressed effectively:

Extracting Relevant Questionnaire Data with GPT-4

At the beginning, GPT-4 produced inconsistent and incomplete outputs when tasked with extracting questionnaire information from clinical trial protocols. Sometimes it missed questionnaires, sometimes it misunderstood the type (e.g., mixing PRO and PREM), and often it failed to recognise subtle scheduling details.

How we solved it:

- We spent months on prompt engineering, testing different prompt templates, examples, and structures.
- We studied prompt engineering research to optimise the phrasing, context setting, and instruction layering.
- We built prompts that specifically instructed GPT-4 to output:
    - Long and short questionnaire names
    - Correct type (PRO, PREM, ClinRO)
    - Precise scheduling/timing information
- Iterative testing across real world clinical protocols led to a prompt structure that produced high accuracy, reliable results.

Matching Procedures Between Adobe and GPT Extractions

One of the hardest problems was matching the questionnaires and procedures extracted from the tables (Adobe API) and text sections (GPT):

- Names were inconsistent (e.g., "PROMIS Fatigue Short Form" vs. "PROMIS Fatigue").
- Structures were different (Adobe extracted flat tables; GPT extracted structured JSON).
- References were indirect or scattered across the documents.

How we solved it:

- We used vector similarity algorithms (text embeddings and cosine similarity) to match procedures even when names weren't identical.

- We designed a set of custom rules and thresholds to define when two entries should be considered a "match" (e.g., similarity score > 0.6, matching context).
- We preprocessed names (e.g., lowercasing, removing common suffixes) to normalise them before comparison.

This made it possible to link the extracted timelines and procedures into a unified, coherent output for users.

Understanding Clinical Trial Domain Complexity

Clinical trial protocols are extremely diverse, and initially, we lacked a deep understanding of the field. Key challenges included:

- Different naming conventions (e.g., "Assessment Visit 1" vs. "Baseline Visit").
- Complex types of questionnaires and different definitions depending on the trial phase or therapeutic area.

How we solved it:

- We studied clinical trial design through public resources like ClinicalTrials.gov.
- We read real protocols across oncology, cardiology, and neurology to understand how questionnaires were referenced and scheduled.
- We built domain knowledge ourselves, which then allowed us to better fine tune GPT prompts and design correct matching logic.

Without this manual research effort, it would have been impossible to generalise the system to new, unseen protocols.

Table Extraction Challenges

Tables in clinical trial protocols were messy and inconsistent, presenting several major issues:

- Non standardised headers (e.g., "Visit" vs. "Timepoint").
- Merged cells, multiline headers, and variable structures across protocols.
- Tables sometimes spanned multiple pages or were embedded inside sections unrelated to assessments.

How we solved it:

- We added post processing logic after using the Adobe PDF Extract API, specifically:
    - Unmerged cells programmatically where needed.
    - Standardised headers across tables.

    - Filtered irrelevant tables and kept only assessment related ones.
- We also visualised intermediate results to manually validate that our extractions made sense for users and for downstream processing.

Table extraction was one of the most time consuming and difficult engineering challenges in the project.

Connecting GPT and Adobe Extracted Data

Although GPT-4 and Adobe API could individually extract valuable data, connecting their outputs into a coherent final JSON was extremely tricky:

- They produced different structures (CSV tables vs. JSON paragraphs).
- They used different wording for the same questionnaires or procedures.
- They labeled schedules differently (e.g., "Baseline and Month 12" vs. "Visit 1, Visit 5").

How we solved it:

- We built a text similarity engine to align GPT extracted questionnaires with Adobe extracted timepoints.
- We normalised both datasets before matching.
- We created mapping logic that merged the matched results into a final, user friendly JSON format.

This was one of the most critical integrations to achieve reliable end to end results.

Generalising to New Protocols

Because every clinical trial protocol was different (different structure, terminology, and format), building a system that generalized was a big challenge.

How we solved it:

- We iteratively tested the system on many protocols from ClinicalTrials.gov.
- We built flexibility into the prompts, table matchers, and validation rules to handle variation.
- We introduced threshold based matching and fallback rules to catch edge cases.

This made *TimelineXtract* robust enough to handle a wide variety of protocol styles.

Operational Challenges: Timing and Resource Management

As a university project, time was limited:

- We had to balance *TimelineXtract* development with other modules and exam preparation.
- Testing prompt engineering, running Adobe API extractions, and refining logic were all very time intensive.
- Cloud API costs also had to be minimised, requiring careful scheduling of test runs.

How we solved it:

- Strict weekly milestones and clear task breakdowns.
- Prioritisation of critical features (e.g., reliable extraction) over extras.
- Parallelisation of some tasks (e.g., prompt testing while running extraction scripts).

This helped us complete the project on time despite the heavy workload.

Results

The *TimelineXtract* system achieved its intended functionality and surpassed our expectations in several areas.

Functional Achievements

- **Successful PDF Upload**: Users can upload protocols seamlessly via the frontend.
- **Accurate Data Extraction**: Both GPT and Adobe's API reliably extract questionnaire and schedule information.
- **Database Integration**: All entities (Users, PDFs, Queries, Tables, Output) are stored in MongoDB and linked by unique identifiers.
- **Frontend Presentation**: Extracted data is displayed in a structured and readable format for users to review.

Example Output (JSON)

```
{
  "Project Title": "NIS-PSO-TARGET Non-Interventional Study Protocol",
  "Sponsor": "LEO Pharma France",
  "Study Number": "RCB: 2020-A00652-37",
  "Protocol Version and Date": "Not Available",
  "Study Title": "NIS-PSO-TARGET Non-Interventional Study Protocol",
  "Phase": "Not Available",
  "Therapeutic Area": "Psoriasis",
  "Number of Patients": "150",
  "Number of Sites": "Approximately 25 sites",
  "Indication": "Psoriasis vulgaris",
  "Duration of Treatment": "52 weeks",
    "Schedule of Assessments": "Not Available",
    "questionnaires": [
        {
            "longName": "Psoriasis Area and Severity Index",
            "shortName": "PASI",
            "type": "ClinRO",
            "questionnaireSchedule": "Collected at baseline, 12 weeks, and 52 weeks.",
            "questionnaireTiming": [
                "Base-line Visit (Day 0)",
                "3 months Visit (week 12-16)",
                "12 months visit (week 48-56)"
            ]
        },
        {
            "longName": "Dermatology Life Quality Index",
            "shortName": "DLQI",
            "type": "PRO",
            "questionnaireSchedule": "Collected at baseline, 12 weeks, and 52 weeks.",
            "questionnaireTiming": [
                "Base-line Visit (Day 0)",
                "3 months Visit (week 12-16)",
                "12 months visit (week 48-56)"
            ]
        },
        {
            "longName": "PSO-TARGET Quality of Life Component Grid",
            "shortName": "PSO-TARGET",
            "type": "PRO",
            "questionnaireSchedule": "Collected at baseline and 12 weeks.",
            "questionnaireTiming": [
                "Base-line Visit (Day 0)",
                "3 months Visit (week 12-16)",
                "12 months visit (week 48-56)"
            ]
        }
    ]
}
```

Performance & Evaluation

We talk more in detail about the performance and evaluation in the *System Test Documentation.*

Overall Metric	Avg Precision	Avg Recall	Avg F1-Score
Questionnaire Evaluation	72.60%	76.12%	72.78%
Timepoints Evaluation	62.80%	61.88%	57.21%

Future Work

While *TimelineXtract* is fully functional, there are several exciting opportunities to improve and extend the system:

Smarter Table Recognition

Although Adobe's PDF Extract API performs well, it occasionally doesn't classify complex clinical tables correctly or struggles with nested structures. A future improvement would involve:

- Training a custom deep learning model using frameworks like **PyTorch** or **TensorFlow**.
- The model would specialise in detecting and segmenting tables within clinical trial protocols, making it more robust against unusual formats.
- This could be combined with layout aware models like **LayoutLM** for even better understanding of visual structure.

Interactive Timeline Visualisation

Currently, the extracted schedules are presented in JSON format. To make the data more user friendly and insightful, we plan to:

- Integrate visualisation libraries such as **D3.js** or **Recharts**.
- Create interactive timeline charts that show questionnaire schedules dynamically.
- Allow users to filter, zoom, and explore timelines based on phases, visits, or types of questionnaires.

This would enhance usability, especially for clinicians or researchers who need to quickly understand patient assessment schedules.

Expert Review Annotations

We recognise that clinical expertise is crucial for validating extracted protocol information. In future iterations, we plan to:

- Collaborate with clinicians, research coordinators, or CROs (Clinical Research Organisations).
- Build an annotation system where experts can review, correct, and comment on extracted data.
- Use this expert validated dataset to fine tune prompts or retrain models, boosting overall system accuracy.

This feedback loop would dramatically improve trustworthiness and clinical validity.

Improving Matching Accuracy and Prompt Quality

While *TimelineXtract* achieves a good performance, we see great opportunities to further improve accuracy:

- **Testing on a broader set of protocols** from different disease areas, trial phases, and sponsor styles.
- **Enhancing the matching algorithms** with better text similarity measures (e.g., using Sentence Transformers or hybrid vector approaches).
- **Refining GPT-4 prompt structures** to maximise recall and precision when extracting questionnaires and matching schedules.

Through iterative testing and continuous prompt engineering, we aim to make *TimelineXtract* even more generalisable, accurate, and reliable across the vast landscape of clinical research protocols.

Work Distribution

Task	Description	Contributor
Project Planning	Defining scope, features, research goals, and timeline	Lorena & Darragh
Frontend Development & Styling	Building UI using React.js, including Upload, Display, Error, Login, and Home pages; Creating minimalist professional design using CSS	Lorena
Google OAuth Integration	Setting up secure Google authentication restricted to mail.dcu.ie users	Lorena
Backend Development (Django)	Setting up Django server, API endpoints, CSRF handling, error management	Lorena
PDF Upload and Processing Pipeline	Handling uploads, saving temp files, validating PDFs	Lorena
Adobe API Integration	Extracting structured content (tables) from PDFs with semantic accuracy	Lorena
Prompt Engineering for GPT-4 & Integration	Designing and testing multiple prompt templates to extract correct questionnaire data; Sending extracted content to GPT-4 for questionnaire identification	Lorena
GPT-4 Integration	Sending extracted content to GPT-4 for questionnaire identification	Lorena
Questionnaire and Timeline Matching	Implementing vector similarity matching between Adobe tables and GPT outputs	Lorena
MongoDB Integration	Designing and implementing collections: Users, PDFs, Queries, Tables, Outputs	Lorena
Data Validation & Error Handling	Adding strict validation for users, uploads, and database consistency	Lorena

Testing with Multiple Protocols	Testing TimelineXtract with different real world clinical trials (from ClinicalTrials.gov)	Lorena
Security Measures	CORS setup, JWT handling, restricting file uploads and user access	Lorena
Deployment Preparation	Prepare the necessary environments for deployment & research of deployment tools	Darragh
Deployment Execution	Make deployment of the system	Darragh
Results Evaluation	Analysing extracted outputs for accuracy and performance	Lorena
Documentation Writing	Writing Technical Manual & Testing Documentation	Lorena
Documentation Writing	Writing User Manual	Darragh
Poster Creation	Design & Creation of Poster	Darragh

References

1. C. Rosemblat, T. Coden, and A. Rindflesch, “Automatically extracting clinical trial purpose from abstracts,”, LHC publications, National Library of Medicine, 2007. [Online]. Available: https://lhncbc.nlm.nih.gov/LHC-publications/PDF/pub2007060.pdf

2. Z. Liu, Y. Lin, H. Shang, and H. Yu, “Clinical Trial Information Extraction with BERT,” Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021. [Online]. Available: https://www.researchgate.net/publication/355343817_Clinical_Trial_Information_Extraction_with_BERT

3. J. White, A. Soviany, M. G. de Oliveira, and C. Apidianaki, “A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT,” arXiv preprint, arXiv:2302.11382, 2023. [Online]. Available: https://arxiv.org/pdf/2302.11382

4. T. N. Tran, S. Nguyen, and L. P. Nguyen, “Prompt Engineering for Large Language Models in Healthcare: Opportunities and Challenges,” Journal of Medical Internet Research, vol. 25, p. e50638, 2023. [Online]. Available: https://www.jmir.org/2023/1/e50638/

5. R. Shin, C. Suh, and D. Song, “Prompt Engineering or Fine-Tuning? Evaluating Large Language Models for Automated Software Engineering,” arXiv preprint, arXiv:2310.10508, 2023. [Online]. Available: https://arxiv.org/pdf/2310.10508