**A logo with a black background

Description automatically generated**

System Test Document

**Student 1 –** Lorena Gomez – 21734359

**Student 2 –** Darragh Manning - 21506373  
**Supervisor -** Tomas Ward **Project Name – TimelineXtract**

**Last modified on:** 01/05/2025

**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.

Through extensive prompt engineering, vector similarity matching, and real world testing using protocols from ClinicalTrials.gov, TimelineXtract delivers clean, structured outputs that streamline clinical research workflows.

# 

# Table of Contents

[**Table of Contents 2**](#_vlfy7ehxr4fh)

[**Introduction 3**](#_htm7frjytx9k)

[**Testing Strategy 3**](#_9s5b9lv88m3u)

[**Testing Tools and Frameworks 3**](#_52xld0uaf3d1)

[**Unit Testing 3**](#_2dw8466s8rds)

[Key Components Tested 3](#_1vawglfutc8c)

[**Integration Testing 7**](#_d5lee4lo0rab)

[Key End to End Tests: 7](#_2f08x6djsuxc)

[**System Testing 7**](#_l7u3c4lp3w8g)

[Core System Test Cases 7](#_2m4o332bgem5)

[Example System Test 7](#_cs586fgcqpui)

[**Ad Hoc Testing 7**](#_c67qmfrzjf05)

[**Similarity Evaluation & Metrics 8**](#_gpwbrd13kv6)

[**User Validation Testing 8**](#_y4r1ln7lp9oi)

[User Access Tests 8](#_extnydkubijy)

[**Robustness and Error Handling Testing 9**](#_kzepozdcpjjy)

[**Data Integrity Testing 9**](#_jqpls0vv65ar)

[**Testing Results Summary 9**](#_pflyoruaqj8s)

[**Future Testing Improvements 10**](#_5v8c7ntug99t)

[**Limitations and Accuracy Challenges 10**](#_735d19n88y1b)

[**How to Improve Accuracy and Precision 12**](#_nwjkdaj28qwi)

[**Conclusion 13**](#_6mouqs3o6w41)

# 

# Introduction

This document outlines the testing strategy, techniques, and results used to validate the quality and robustness of the *TimelineXtract* system. Given the nature of the project, a web-based application for extracting structured data from complex clinical trial PDFs, we employed a combination of **unit testing**, **integration testing**, **system testing**, **ad hoc testing, and formal testing** to ensure correctness, reliability, and usability.

# Testing Strategy

| **Testing Type** | **Purpose** |
| --- | --- |
| Unit Testing | Ensure individual components (e.g., database insertion, file handling) are correct |
| Integration Testing | Confirm end to end flows work correctly across components |
| System Testing | Validate that complete user workflows function as intended |
| Ad Hoc Testing | Real world manual testing to catch unexpected issues during development |
| Similarity Evaluation Testing | Quantitatively evaluate output accuracy against a manually created ground truth |

# Testing Tools and Frameworks

| **Tool** | **Purpose** |
| --- | --- |
| Python unittest | Unit and integration testing |
| Postman | API endpoint testing |
| Django Test Client | Backend testing automation |
| Sentence Transformers | Semantic similarity scoring |
| React.js Manual Validation | Frontend UI manual testing |

# Unit Testing

### Key Components Tested

* **PDF Extraction Utility:** extract\_and\_classify\_tables() tested with simple and complex PDFs.

**→** Example code (src/timelinextract/srcExtractor/tests/test\_pdf\_processing.py):

class PDFProcessingTests(TestCase):

@patch("os.makedirs")

@patch("builtins.open", new\_callable=mock\_open, read\_data=b"fake pdf content")

@patch("srcExtractor.utils.pdf\_processing.extract\_tables", return\_value={"success": True})

@patch("srcExtractor.utils.pdf\_processing.classify\_all\_tables\_in\_folder", return\_value={"success": ["table1.csv", "table2.csv"]})

@patch("srcExtractor.utils.data\_processing.convert\_valid\_files\_to\_json")

def test\_extract\_and\_classify\_tables\_success(self, mock\_convert\_json, mock\_classify, mock\_extract, mock\_file, mock\_mkdir):

"""Test successful extraction and classification of tables from a PDF."""

result = extract\_and\_classify\_tables("sample.pdf", "sample")

self.assertIn("valid\_files", result)

self.assertEqual(result["valid\_files"], ["table1.csv", "table2.csv"])

@patch("builtins.open", new\_callable=mock\_open, read\_data=b"fake pdf content")

@patch("srcExtractor.utils.pdf\_processing.extract\_tables", return\_value={"error": "Extraction failed"})

def test\_extract\_and\_classify\_tables\_extraction\_error(self, mock\_extract, mock\_file):

"""Test handling of errors during table extraction."""

result = extract\_and\_classify\_tables("sample.pdf", "sample")

self.assertIn("error", result)

self.assertEqual(result["error"], "Extraction failed")

@patch("builtins.open", new\_callable=mock\_open, read\_data=b"fake pdf content")

@patch("srcExtractor.utils.pdf\_processing.extract\_tables", return\_value={"success": True})

@patch(

"srcExtractor.utils.pdf\_processing.classify\_all\_tables\_in\_folder",

return\_value={"error": "No Schedule of Events table found in the protocol."}

)

def test\_extract\_and\_classify\_tables\_no\_valid\_files(self, mock\_classify, mock\_extract, mock\_file):

"""Test when no valid tables are found after classification."""

result = extract\_and\_classify\_tables("sample.pdf", "sample")

self.assertIn("error", result)

self.assertEqual(result["error"], "No Schedule of Events table found in the protocol.")

@patch("builtins.open", new\_callable=mock\_open, read\_data=b"fake pdf content")

@patch("srcExtractor.utils.pdf\_processing.extract\_tables", return\_value={"success": True})

@patch("srcExtractor.utils.pdf\_processing.classify\_all\_tables\_in\_folder", side\_effect=Exception("Unexpected error"))

* **PDF Processing and Validation:** handle\_pdf\_upload() tested for normal, missing, and invalid API responses.

**→** Example code (src/timelinextract/srcExtractor/tests/test\_pdf\_validation.py):

class PDFUploadTests(TestCase):

@patch("os.path.getsize", return\_value=1024)

@patch("srcExtractor.utils.pdf\_validation.extract\_text\_from\_pdf", return\_value={"text": "Sample text from PDF"})

def test\_handle\_pdf\_upload\_success(self, mock\_extract\_text, mock\_getsize):

"""Test successful PDF file upload and text extraction."""

result = handle\_pdf\_upload("test.pdf")

self.assertIn("success", result)

self.assertEqual(result["success"], "test.pdf")

def test\_handle\_pdf\_upload\_no\_file(self):

"""Test when no PDF file is provided."""

result = handle\_pdf\_upload("")

self.assertIn("error", result)

self.assertEqual(result["error"], "No PDF file in data.")

@patch("os.path.getsize", return\_value=1024)

def test\_handle\_pdf\_upload\_invalid\_file\_type(self, mock\_getsize):

"""Test when a non-PDF file is uploaded."""

result = handle\_pdf\_upload("test.txt")

self.assertIn("error", result)

self.assertEqual(result["error"], "Only PDF files are allowed.")

@patch("os.path.getsize", return\_value=0)

def test\_handle\_pdf\_upload\_empty\_pdf(self, mock\_getsize):

"""Test when an empty PDF file is uploaded."""

result = handle\_pdf\_upload("empty.pdf")

self.assertIn("error", result)

self.assertEqual(result["error"], "Uploaded PDF file is empty.")

@patch("os.path.getsize", side\_effect=Exception("File not found"))

def test\_handle\_pdf\_upload\_file\_not\_found(self, mock\_getsize):

"""Test when the provided PDF file does not exist."""

result = handle\_pdf\_upload("non\_existent.pdf")

self.assertIn("error", result)

self.assertEqual(result["error"], "Error validating the file: File not found")

* **MongoDB Database Operations:**Functions like add\_user(), add\_pdf() tested for edge cases (e.g., missing fields, duplicate entries).

**→** Example code (src/timelinextract/srcExtractor/tests/test\_user\_mongodb.py):

class UserMongoDBTests(TestCase):

def test\_is\_valid\_email(self):

self.assertTrue(user.is\_valid\_email("user@example.com"))

self.assertTrue(user.is\_valid\_email("user.name+tag@example.co.uk"))

self.assertFalse(user.is\_valid\_email("not-an-email"))

self.assertFalse(user.is\_valid\_email("missing@domain"))

self.assertFalse(user.is\_valid\_email("user@.com"))

def test\_add\_user\_missing\_email(self):

"""Should return error if email is missing"""

result = user.add\_user({})

self.assertIn("error", result)

self.assertEqual(result["error"], "Missing required field: email.")

def test\_add\_user\_invalid\_email(self):

"""Should return error if email format is invalid"""

result = user.add\_user({"email": "invalid-email"})

self.assertIn("error", result)

self.assertEqual(result["error"], "Invalid email format.")

@patch("srcExtractor.services.mongodb.user.mongodb.get\_collection")

def test\_add\_user\_already\_exists(self, mock\_get\_collection):

"""Should detect existing user and return message"""

mock\_collection = MagicMock()

mock\_collection.find\_one.return\_value = {"\_id": "abc123", "email": "user@example.com"}

mock\_get\_collection.return\_value = mock\_collection

result = user.add\_user({"email": "user@example.com"})

self.assertIn("message", result)

self.assertEqual(result["message"], "User 'user@example.com' already exists.")

self.assertEqual(result["user\_id"], "abc123")

@patch("srcExtractor.services.mongodb.user.mongodb.get\_collection")

def test\_add\_user\_success(self, mock\_get\_collection):

"""Should create new user if not found"""

mock\_collection = MagicMock()

mock\_collection.find\_one.return\_value = None

mock\_collection.insert\_one.return\_value.inserted\_id = "newid456"

mock\_get\_collection.return\_value = mock\_collection

result = user.add\_user({"email": "newuser@example.com"})

self.assertIn("message", result)

self.assertEqual(result["message"], "User 'newuser@example.com' created successfully.")

self.assertEqual(result["user\_id"], "newid456")

# Integration Testing

### Key End to End Tests:

* Backend:
  + Uploading a PDF → GPT data extraction → Adobe API table extraction → MongoDB storage → Display on frontend.
* Frontend:
  + Authentication → Token validation → File upload → Results presentation.

**Result:** All integration points functioned correctly. Minor JSON validation bugs were fixed during development.

# System Testing

### Core System Test Cases:

| **Test Case** | **Description** | **Expected Outcome** | **Status** |
| --- | --- | --- | --- |
| TC-001 | Upload valid clinical protocol | JSON extracted and displayed | Passed |
| TC-002 | Upload non-PDF file | Error message shown | Passed |
| TC-003 | Login with non-DCU email | Access denied | Passed |
| TC-004 | View extracted timeline | Correct UI rendering | Passed |

### Example System Test:

**Input:** Clinical trial PDF with multiple questionnaire schedules.  
**Action:** Upload through the web interface.  
**Expected Output:** Structured JSON with questionnaire names, types, and schedules.  
**Actual Output:** As expected, extracted questionnaires were correctly structured.

# Ad Hoc Testing

→ Uploaded incomplete, corrupted, and oddly formatted PDFs.

→ Manually checked outputs.

→ Tested UI under different browsers and screen sizes.

# Similarity Evaluation & Metrics

One major part of the validation was quantitative testing against a manually built ground truth:

* We manually extracted questionnaires, types, and timelines from **50 real clinical protocols** from ClinicalTrials.gov.
* Used a custom semantic similarity script to compare predictions vs ground truth.
* Metrics computed:
  + **Precision** = True Positives / (True Positives + False Positives)
  + **Recall =** True Positives / (True Positives + False Negatives)
  + **F1-Score** = 2 x (precision x recall) / (precision + recall)

**Result:**

| **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% |

# 

# User Validation Testing

### User Access Tests:

| **Test Case** | **Description** | **Expected Outcome** | **Status** |
| --- | --- | --- | --- |
| UV-001 | Login with DCU email (@mail.dcu.ie) | Successful login | Passed |
| UV-002 | Login with non-DCU email | Rejected with error message | Passed |
| UV-003 | Missing email field in token | Error response from backend | Passed |
| UV-004 | Re-login existing user | No duplicate entry in database | Passed |
| UV-005 | Email faking attempt | Caught and rejected | Passed |

# Robustness and Error Handling Testing

| **Test Case** | **Description** | **Expected Outcome** | **Status** |
| --- | --- | --- | --- |
| RE-001 | Upload corrupted PDF | Error handled gracefully | Passed |
| RE-002 | Adobe API timeout | Retry or error displayed | Passed |
| RE-003 | GPT-4 API failure | Informative error message shown | Passed |
| RE-004 | MongoDB connection loss | Proper error reporting/logging | Passed |

# Data Integrity Testing

| **Test Case** | **Description** | **Expected Outcome** | **Status** |
| --- | --- | --- | --- |
| DI-001 | User uploads multiple PDFs | PDFs linked correctly to users | Passed |
| DI-002 | Same file uploaded twice | Unique filenames ensured | Passed |
| DI-003 | MongoDB linking across collections | Users, PDFs, Queries, and Outputs correctly linked | Passed |

# Testing Results Summary

| **Category** | **Outcome** |
| --- | --- |
| Upload functionality | 100% Passed |
| Authentication | 100% Passed |
| Data extraction accuracy | Medium (Precision/Recall > 60%) |
| Data integrity | No missing links detected |
| UI usability | Smooth under real usage |
| Error handling | System stable under faults |

# Future Testing Improvements

To further validate and strengthen the performance of TimelineXtract, we propose the following future testing initiatives:

**→ Expand Ground Truth Dataset:** Extend the manually constructed evaluation set from 50 to 100+ protocols, including more diverse therapeutic areas and formats. This will improve statistical reliability and help identify edge cases.

**→ High-Load Simulations:** Conduct stress testing with multiple users uploading large PDF files simultaneously to evaluate system stability, performance under load, and scalability of backend services.

**→ Automated Frontend Testing:** Integrate tools like **Cypress** or **Selenium** to automate UI regression testing, especially around file uploads, user authentication, and result rendering.

# Limitations and Accuracy Challenges

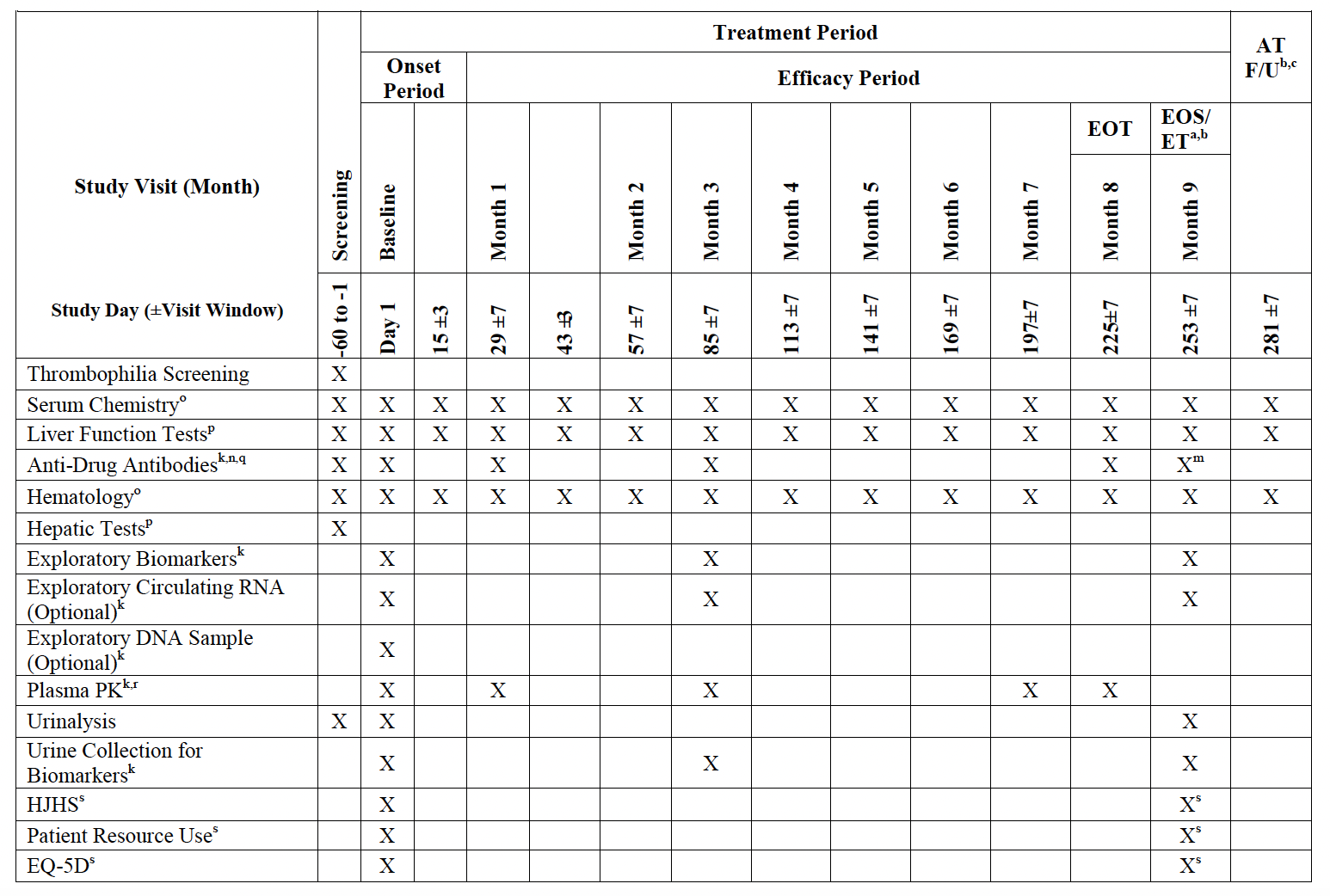
Despite achieving decent performance, our system’s accuracy and recall on timepoints extraction is not perfect due to several technical challenges:

**→ Unpredictability of AI Output:** GPT-4, while powerful, is inherently non-deterministic. Achieving 100% precision and recall requires extensive prompt tuning and context control, which is possible but time consuming. Additional research into few-shot or retrieval-augmented prompting could help.

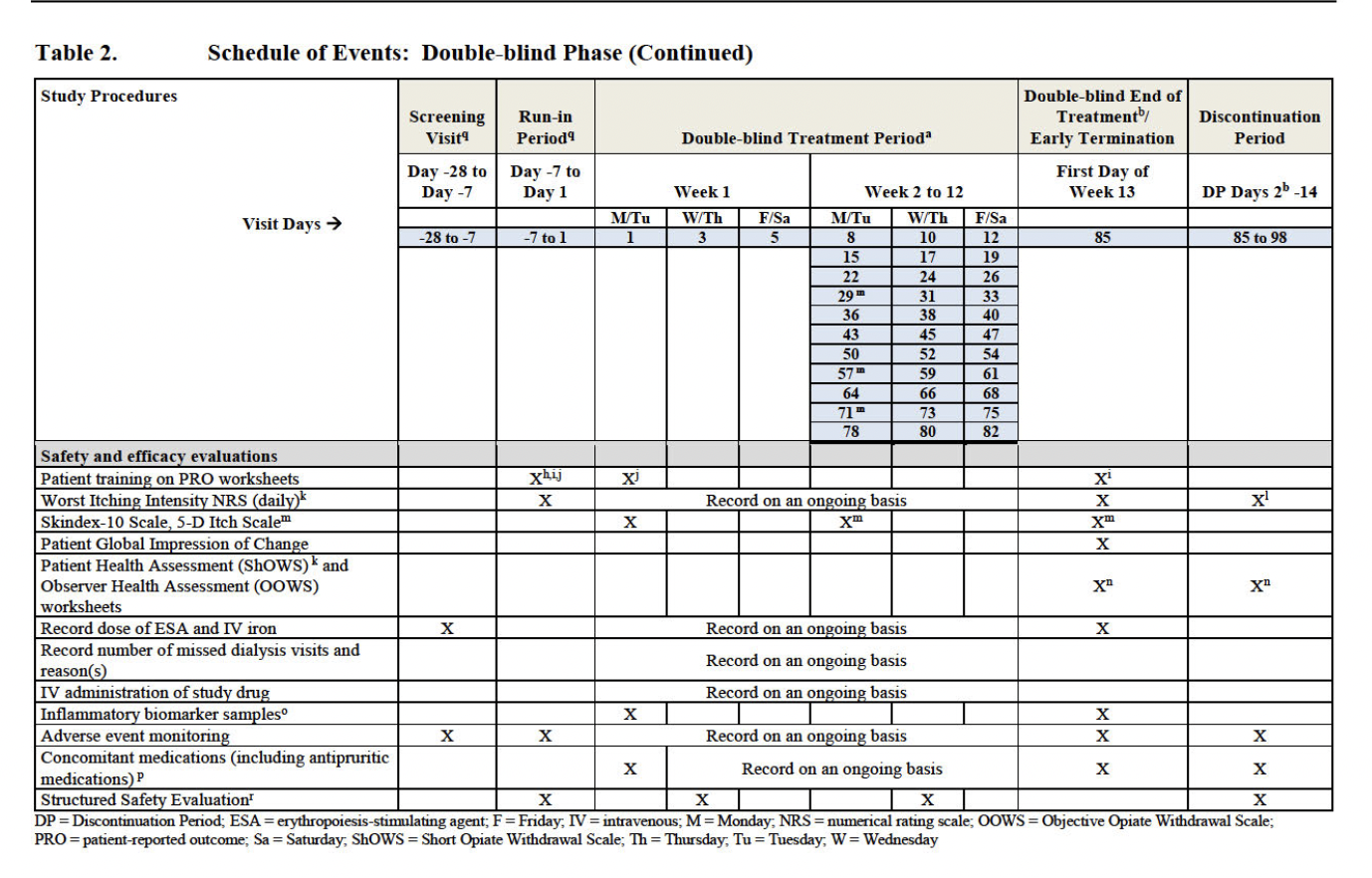
**→ Complexity of Table Formats:** The table extraction process is the most error prone stage due to:

* **Merged cells**, which confuse the row/column alignment.
* **Vertical or rotated text**, which Adobe’s Extract API struggles to interpret.
* **Multipart or broken tables**, where questionnaire procedures are split across multiple disconnected tables.

Tables with vertically written procedures or nested structures often led to empty or misaligned timepoints being extracted. Example:



This is another example of a *Schedule of Events* table where the structure is hard to read even for a human:



**→ Text-based Timelines Not Counted in Metrics:**Our evaluation only measured the time points extracted from tables. However, in some protocols, the correct schedule was found by GPT-4 in the narrative text and not in a table.  
  
 **Example:**  
{

"longName": "Godin Leisure-Time Exercise Questionnaire",

"shortName": "GLTEQ",

"type": "PRO",

"questionnaireSchedule": "Baseline and at end of study visit",

"questionnaireTiming": [

"Questionnaires (Informed Consent)(Procedures)"

]

}

In this case, the system correctly extracted the schedule from the narrative text (“questionnaireSchedule”), but the associated table (“questionnaireSchedule”) failed to give usable timepoints. This discrepancy lowers the recall score, even though the system ultimately produced accurate information.

# How to Improve Accuracy and Precision

To improve accuracy in future versions of *TimelineXtract,* if we had more time, we could:

**→ Table Extraction Upgrades:**

* Use layout aware models like [**LayoutLMv3**](https://arxiv.org/abs/2204.08387) or **PubLayNet** trained object detectors for better understanding of complex table structures.
* Implement post processing logic to identify and normalise rotated or merged cell content.

**→ Enhanced Prompt Engineering:**

* Introduce dynamic prompt templates that adapt based on protocol structure and table contents.

**→ Domain Specific Synonym Matching:**

* Add a medical synonym dictionary to better match varied questionnaire names (e.g., “EuroQol 5D” vs. “EQ-5D”).

**→ Feedback Loop for Active Learning:**

* Use clinician or user feedback to retrain or refine matching heuristics based on validated matches.

# Conclusion

Testing was a very important part of *TimelineXtract’s* development. Through a combination of structured unit testing, integration testing, comprehensive system testing, semantic output evaluation, and ad hoc exploratory validation, we built a reliable and robust system capable of extracting high quality clinical trial data from complex, real world documents.

We validated not only the correctness of individual components (such as file parsing and API integration) but also the end to end behavior of the full pipeline, from upload to AI processing to final output display. Using a manually curated ground truth dataset of 50 clinical protocols, we implemented semantic similarity scoring with sentence transformers to evaluate precision, recall, and F1 score, giving us measurable insights into the effectiveness of our AI extraction pipeline.

In parallel, we tested system robustness with malformed files, edge cases in user login, and challenging PDF layouts. These helped us detect and mitigate potential failures early. Our authentication system, database structure, and UI/UX flow were also tested and refined with the user in mind, ensuring security, traceability, and clarity throughout the workflow.

While some components such as table extraction and timepoint recognition remain challenging due to the variability of clinical documents, we identified clear paths for improving accuracy through better data fusion, smarter models, and broader validation sets.

Ultimately, our testing approach gave us strong confidence that *TimelineXtrac*t is not just functional, but reliable and secure. Future enhancements to testing (including load simulations, expanded ground truths, and automated UI validation) will allow the system to scale and maintain its performance across more demanding use cases.