# ATOM GROUP

## Engineering Assessment

Backend & AI Engineer — Document Intelligence Pipeline

| | |
|---|---|
| **Role** | Backend & AI Engineer |
| **Duration** | 5 Hours |
| **Stack** | Python, FastAPI, OpenAI API |
| **Submission** | Public GitHub Repository |
| **Evaluation** | Code quality, architecture decisions, problem-solving |

## The Brief

You are building a Document Intelligence Pipeline — a FastAPI service that accepts document uploads, queues them for processing, extracts text content, runs AI-powered analysis, and streams real-time progress updates back to the client.

This is not a toy project. This simulates a real production service where multiple documents arrive, get queued, processed in order by background workers, and clients need visibility into what's happening at every stage.

## System Overview

Your service must handle the following flow:

### Document Lifecycle

**Upload** → A client uploads one or more documents (PDF or TXT). Each document is validated and saved.

**Queue** → Each uploaded document enters a processing queue with status `pending`.

**Process** → A background worker picks up documents one by one, extracts text content, and updates status to `processing`.

**Analyze** → After extraction, the worker sends the text to an LLM for analysis (summary, key insights, sentiment). Status updates to **`analyzing`**.

**Complete** → Results are stored and the document status becomes **`completed`**. If anything fails, the status becomes **`failed`** with an error message.

**Status Streaming:** The client must be able to connect to a streaming endpoint (SSE — Server-Sent Events) and receive real-time status updates as documents move through the pipeline. This is a core requirement, not a bonus.

## Technical Requirements

### 1. Document Upload & Validation

Build an endpoint that accepts single or multiple file uploads simultaneously.

- Accepted file types: PDF (.pdf) and plain text (.txt)
- Maximum file size: 10MB per file
- Validate file type and size before saving. Reject invalid files with clear error messages.
- Store uploaded files on disk in an organized structure (e.g., /uploads/{document_id}/filename.ext)
- Return a document ID and initial metadata for each uploaded file

### 2. Processing Queue & Background Workers

Implement a queue system that manages document processing order.

- Each uploaded document should be added to a processing queue automatically
- A background worker should continuously pick up queued documents and process them
- Documents should be processed in FIFO order (first in, first out)
- The worker must handle text extraction: read the raw content from PDFs and text files
- Track and persist document status transitions: **`pending`** → **`processing`** → **`analyzing`** → **`completed`** (or **`failed`**)

**Implementation Note**
You may use any approach for the queue and background processing: asyncio tasks, threading, an in-memory queue, or any lightweight solution. Do not use Celery or Redis —

we want to see how you architect this from simpler building blocks. Justify your approach in your README.

## 3. Real-Time Status Streaming (SSE)

Implement a Server-Sent Events endpoint that streams document status updates to connected clients in real time.

- Client connects once and receives updates for all their documents as they progress through the pipeline
- Each event should include: document ID, new status, timestamp, and any relevant metadata
- When a document completes, include the analysis results in the final event
- Handle client disconnection gracefully — clean up resources when a client drops

**Example SSE event stream:**

```
data: {"document_id": "abc123", "status": "pending", "timestamp": "..."}

data: {"document_id": "abc123", "status": "processing", "timestamp": "..."}

data: {"document_id": "abc123", "status": "analyzing", "timestamp": "..."}

data: {"document_id": "abc123", "status": "completed", "result": {...},
"timestamp": "..."}
```

## 4. AI-Powered Document Analysis

Once text is extracted from a document, send it to an LLM for analysis. Use the OpenAI API (key provided below).

- Generate a concise summary of the document (3–5 sentences)
- Extract key topics or themes as a list
- Determine overall sentiment: positive, negative, neutral, or mixed
- Identify any actionable items or recommendations if present
- Handle LLM API errors gracefully — retry once, then mark as failed with reason

**API Credentials**

**OpenAI API Key:** *[Will be provided at the start of your assessment]*

**Model:** `gpt-4.1`

Use the official OpenAI Python SDK (pip install openai). You are free to design the prompt however you see fit — prompt quality is part of the evaluation.

## 5. Data Persistence

Store all document metadata, status history, and analysis results. You may use any of the following approaches:

- JSON files on disk (organized per document)
- SQLite database
- Any other lightweight solution that does not require a separate database server

Whichever method you choose, ensure that:

- Data survives server restarts (no purely in-memory storage for results)
- You can query documents by status (e.g., show all pending documents)
- Status history is preserved (not just the current status)

## 6. Authentication

Implement simple JWT authentication to protect all endpoints except login.

- POST /auth/login — accepts a username and password, returns a JWT token
- All other endpoints require a valid Bearer token in the Authorization header
- Use hardcoded credentials for simplicity (no database needed for users)
- Tokens should expire after 1 hour

# API Endpoints

Your service must expose the following endpoints:

| Method | Endpoint | Description |
|---|---|---|
| **POST** | `/auth/login` | Authenticate and receive JWT token |
| **POST** | `/documents/upload` | Upload one or multiple documents |

| | | |
|---|---|---|
| **GET** | `/documents` | List all documents with current status |
| **GET** | `/documents/{id}` | Get document details, status history, and analysis results |
| **GET** | `/documents/{id}/status` | Get current processing status of a document |
| **GET** | `/documents/stream` | SSE endpoint — stream real-time status updates |
| **DELETE** | `/documents/{id}` | Remove a document and its associated data |

## What to Submit

Create a public GitHub repository containing your complete solution. Your repository must include:

### Working Code

- The project should run locally with minimal setup (ideally a single command)
- Include a requirements.txt or pyproject.toml with all dependencies
- The server should start without errors and all endpoints should be functional

### README.md

Your README is part of the evaluation. It should contain:

- Clear setup and run instructions
- **Architecture overview:** explain how the queue, workers, and streaming components interact
- **Design decisions:** why you chose your specific approach for the queue, persistence, and streaming
- **Time breakdown:** how you allocated your 5 hours
- **Trade-offs:** what you would do differently with more time or in a production setting

### Bonus Points

- Deployed version (Render, Railway, Fly.io, or similar)
- Dockerfile or docker-compose setup
- Meaningful tests (even a few well-chosen ones matter more than 100% coverage)

- WebSocket support alongside or instead of SSE
- Rate limiting on upload or analysis endpoints
- Chunked upload support for large files

## Suggested Time Allocation

| | |
|---|---|
| **Hour 1** | Project setup, test file uploads, design your queue architecture, set up data models |
| **Hours 2–3** | Build the upload endpoint, implement the queue system and background worker, text extraction |
| **Hour 4** | Integrate LLM analysis, build the SSE streaming endpoint, connect all the pieces |
| **Hour 5** | Error handling, edge cases, write README, deploy if time allows |

## Evaluation Criteria

We evaluate your submission holistically. Here is what we are looking at:

| Criteria | What We Look For | Weight |
|---|---|---|
| **Architecture & Design** | How you structure the queue, workers, and data flow. Clean separation of concerns. | **25%** |
| **Code Quality** | Readable, well-organized code. Proper error handling. Consistent patterns. | **25%** |
| **Streaming & Real-Time** | SSE implementation works correctly. Status updates are timely and reliable. | **20%** |
| **AI Integration** | Effective use of the LLM. Good prompt design. Graceful error handling. | **15%** |
| **Documentation** | README quality. Clear reasoning about trade-offs and design decisions. | **15%** |

## Important Rules

**Read Carefully**

1. You have exactly 5 hours from the end of your introductory call. Late submissions will not be reviewed.

2. This is an individual assessment. Use of AI coding assistants (Copilot, ChatGPT, etc.) for code generation is not permitted. We will ask you to walk through your code and explain your decisions.
3. You may reference documentation, Stack Overflow, and similar resources — that is expected.
4. If anything in this document is unclear, ask before you begin. We want you to succeed.
5. Share your GitHub repository link via LinkedIn (our primary communication channel) once complete.

*We are looking for engineers who think clearly, build pragmatically, and communicate their decisions well.*

**Good luck!** 🚀