PDF Text Extraction System

February 15, 2025

1 Executive Summary

This document outlines the implementation and potential scaling design of a PDF text extraction system. The current implementation demonstrates core functionality using PyMuPDF for text-based PDFs and TesseractOCR with LSTM for image-based PDFs, while considering future scaling requirements. Some scenarios are provided to scale the application.

2 Current Implementation

2.1 Core Components

- **Backend (Implemented)**
  - FastAPI-based REST API
  - PyMuPDF for text extraction
  - TesseractOCR for image-based PDFs
  - Docker containerization
- **Frontend (Implemented)**
  - Next.js React application
  - PDF text visualization
  - Bounding box highlighting

2.2 Features

2.2.1 Backend

The backend implementation has a FastAPI-based system with several key features:

- **PDF Processing Pipeline**
  - Implements a dual-mode processing system:
    - Text-based PDF processing using PyMuPDF's text extraction
    - Image-based PDF processing using Tesseract OCR with LSTM
  - Automatic detection and switching between processing modes based on text content
  - Streaming download with 50MB file size limit for resource management
- **Performance Optimizations**
  - GZip compression middleware for response optimization
  - Memory-efficient processing with temporary file handling

  - Image resizing for OCR (max 2000px dimension)
  - Page limit of 2000 to prevent resource exhaustion
- **Text Processing Features**
  - Precise bounding box calculation for text blocks
  - Text block merging with gap analysis
  - Preservation of document structure with appropriate spacing
  - Handling of punctuation and whitespace

2.2.2 Frontend

The frontend is built with Next.js and React featuring a PDF viewing system:

- **PDF Viewer Component**
  - Built on react-pdf-viewer
  - Dynamic highlight overlay system for text blocks
  - Bi-directional synchronization between PDF and transcript
  - Responsive design with automatic focus and fit capabilities
- **Interactive Features**
  - Click-to-highlight functionality with visual feedback
  - Smooth scrolling and centering of highlighted text

3 System Design Considerations

3.1 Performance Analysis

Current single-instance capabilities:

- Processing Rate: Measured 60 seconds for a 40MB PDF $\approx$ 0.67 MB/sec
- Memory Usage: 2GiB per instance
- Estimated max processing time = 50MB / 0.67 MB/sec $\approx$ 75 seconds

3.2 Concurrent Requests

Current single-instance capabilities:

- Requirement: At most 1 req/sec
- Processing time per request: 60-75 seconds
- Maximum concurrent requests = 75 requests

4 Production Requirements

4.1 Infrastructure Needs

For production deployment:

- Load balancer for request distribution
- Optionally: Caching layer for frequent PDFs
- Monitoring and alerting system

4.2 Further Potential Optimizations

- **Caching Strategy**
  - Potentially cache processed PDF data using URL hashing in a database
- **Processing Optimizations**
  - Parallel processing of PDF pages
  - OCR quality vs. speed tuning
  - Memory usage optimization

5 Scenario 1: Infrastructure Analysis: API Gateway, Load Balancer, and Auto Scaling

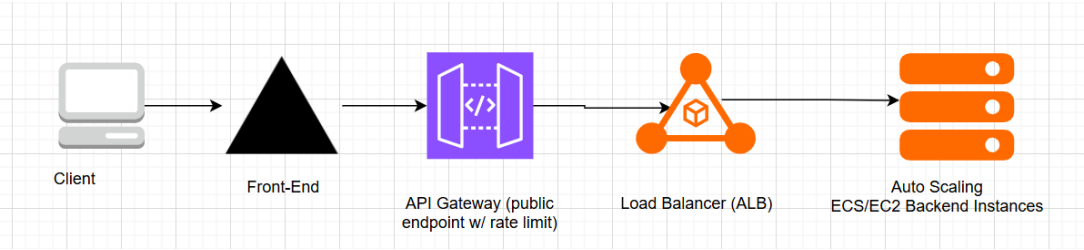

Figure 1: System Diagram for Scenario 1

5.1 API Gateway

The API Gateway acts as the secure entry point and rate limiter.

5.2 Load Balancer

The load balancer distributes incoming requests among EC2 instances. With Health checks for the backend you can ensure that each instance maintains an acceptable processing rate. If t_{health} is the maximum allowable response time for a health check, then instances with $t > t_{\text{health}}$ are removed from the pool.

5.3 Auto Scaling Metrics:

- **CPU Utilization:** Trigger scaling when average CPU $> 70\%$.
- **Memory Usage:** Scale out if memory exceeds 80%.
- **Request Latency:** Maintain average latency $L \leq t_{\max}$, e.g., $L \leq 500\,\text{ms}$.

6 Scenario 2: API Gateway, Queue, and Auto Scaling Instances

In this scenario, an intermediate queue decouples the API Gateway from the processing instances.

6.1 API Gateway

Remains the primary entry point. It forwards requests into the asynchronous queue rather than directly to backend instances.

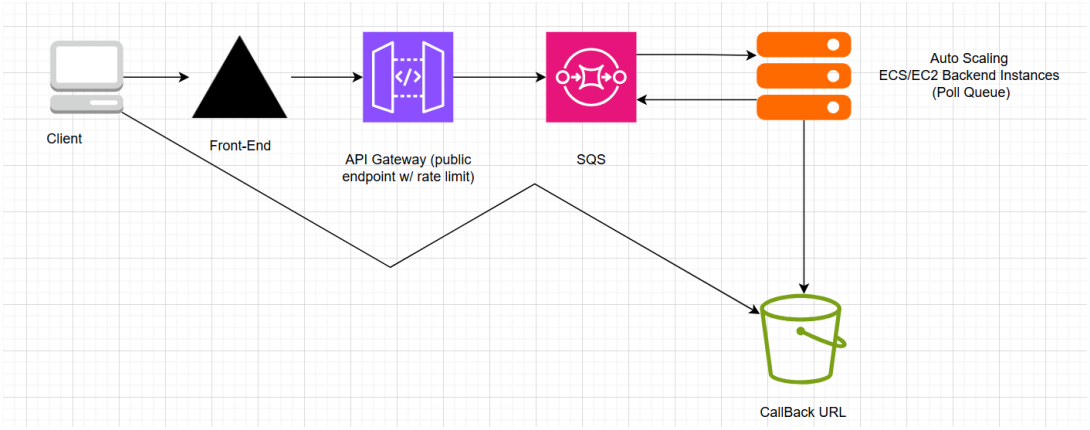

Figure 2: System Diagram for Scenario 2

6.2 Auto Scaling Instances

Instances are scaled based on queue metrics:

- **Queue Length Threshold:** If Q exceeds a predetermined value (e.g., 100 requests), initiate scaling.
- **Waiting Time Threshold:** If W exceeds a set threshold (e.g., 200 ms), additional instances are added.

6.3 Implementation

The backend and frontend implementation will need to be modified to compensate for the asynchronous nature of the queue. To return the request response, the client might poll an S3 bucket referenced by the Job ID, or recieve a callback URL, or use WebSockets.

7 Conclusion

The implemented solution demonstrates core PDF extraction functionality while this document expands on potential scaling scenarios.