Proof of Concept (PoC) Implementation Plan

Executive Summary

The PoC will be a **single, monolithic Python application** using **FastAPI** for rapid API development. Its primary function is to validate the **Tiered Processing Logic** and achieve the target **>90% data extraction accuracy**. It will use a combination of traditional OCR for speed and LLM/VLM for robustness, as required by the core assessment.

PoC Goal: Achieve >90% data extraction accuracy on a diverse set of sample timetables. **Target Duration:** 2–4 Weeks **Evolution Path:** Migrate functions into AWS Lambda/Step Functions (Implementation 2).

1. PoC Architecture and Technology Stack

The architecture is deliberately simple to avoid infrastructure overhead.

Architecture Overview

Component	Function	Technology	Rationale
API Gateway	Single REST endpoint for file upload.	**FastAPI (Python)**	High performance, familiar Python ecosystem, and quick development.
Document Reader	Reads and preprocesses uploaded files (PDF/Image).	**PyMuPDF / Pillow**	Handles various formats, converting everything to a standard image format for processing.
Tier 1 Processor (OCR)	Initial, fast text extraction and confidence scoring.	**Tesseract OCR (or similar)**	Quick and low-cost initial pass for high-quality documents.
Tier 2 Processor (AI Agent)	Robust, complex extraction using an LLM API call.	**Gemini API (Function-Calling)**	Handles low-confidence, complex, or handwritten documents.
Database	Stores extracted Timetable JSON and original document metadata.	**SQLite (Local) or Simple Dictionary**	Minimal setup, sufficient for PoC data persistence/logging.

Core Python Dependencies

- `fastapi` , `uvicorn` : API server
- `pydantic` : Data validation (for input and output JSON schema)
- `python-multipart` : File upload handling
- `pytesseract` , `Pillow` , `PyMuPDF` : Document handling and OCR
- `requests` or official client library: For calling the chosen LLM API (Gemini/OpenAI/etc.)

2. Core PoC Workflow (Tiered Processing Logic)

This is the critical process flow designed to ensure both speed and robustness.

Step 1: File Ingestion (API Endpoint)

1. **Teacher Upload:** User sends a document (PDF, PNG, JPEG) via a `POST /upload-timetable` endpoint.
2. **Preprocessing:** The file is converted into a standard format (e.g., a list of high-resolution images, one per page).
3. **Storage:** The original file and processing ID are stored locally (or in an S3-like mock layer).

Step 2: Tier 1 - Rapid OCR Extraction

1. The image is passed to a traditional OCR engine (e.g., Tesseract).
2. OCR returns raw text and per-word **confidence scores**.
3. A **Pipeline Confidence Score** is calculated (e.g., median word confidence, weighted by word length).
4. **Quality Gate 1 Check:**
   - **IF Confidence $\geq$ 98% (High Quality):** Skip to Step 4 (Data Validation).
   - **ELSE (Low Quality/Complex):** Proceed to Step 3.

Step 3: Tier 2 - Robust LLM/VLM Extraction (The Core Value)

This step leverages a Large Language Model (LLM) to handle complexity, handwritten text, and structural variability.

1. **Image/Text Preparation:** The original image (or raw OCR text) is sent to the LLM API.
2. **Prompt Engineering:** The prompt must include a strong **System Instruction** defining the desired JSON output structure and a **Few-Shot Example** of a timetable input and the corresponding JSON output.
   - **Function-Calling Agent:** The prompt instructs the LLM to use a `extract_timetable(timetable_json)` function and only return the structured JSON data.
3. **LLM Processing:** The LLM extracts data, normalizes time formats, and outputs a structured JSON object.
4. **Quality Gate 2 Check:**
   - If the LLM returns structured JSON, the process moves to Step 4.
   - If the LLM fails to return a valid JSON structure, the document is flagged for **Manual Review** (PoC will only log this failure).

Step 4: Final Validation and Output

1. The final extracted JSON (from either Tier 1 or Tier 2) is validated against the defined **Pydantic Output Schema** (ensuring all required fields like `event_name`, `start_time`, `end_time` are present and correctly formatted).
2. The final **Timetable JSON** and its processing metadata are saved to the PoC database.
3. The API returns the final JSON to the teacher's (mock) frontend.

3. Implementation Steps Checklist (Prioritized)

This checklist prioritizes the logic over infrastructure, ensuring rapid validation.

Phase 1: Setup and Simple OCR (Week 1 Focus)

- [ ] 1. Initialize Python environment and dependencies (FastAPI, uvicorn, etc.).
- [ ] 2. Define the **Pydantic Timetable Output Schema** (The target JSON structure).
- [ ] 3. Create a single `POST /upload-timetable` endpoint to accept a file.
- [ ] 4. Implement **PyMuPDF/Pillow** logic to convert PDF/Images to a standard, high-DPI image for Tesseract.
- [ ] 5. Implement **Tesseract OCR** extraction and return the raw text + confidence score.
- [ ] 6. **CRITICAL:** Implement the logic to calculate the **Pipeline Confidence Score** based on OCR output.

Phase 2: AI Agent Integration and Tiered Logic (Week 2 Focus)

- [ ] 7. Define the **LLM API Client** and an exponential backoff retry mechanism.
- [ ] 8. Develop the initial **LLM Prompt** with Pydantic-informed output structure for function calling.
- [ ] 9. Implement the **Tiered Logic Switch** (Step 2 in the workflow): If OCR confidence is low, trigger the LLM call.
- [ ] 10. Test with 10 sample documents (5 high-quality, 5 low-quality) and log accuracy.

Phase 3: Validation, Persistence, and Migration Prep (Week 3 Focus)

- [ ] 11. Implement final **Pydantic Validation** on the extracted JSON (from both Tiers).
- [ ] 12. Implement local **SQLite/Dictionary persistence** to log successful and failed extractions (for PoC analysis).
- [ ] 13. **Final PoC Test:** Run against a large, pre-labeled test set to validate the **>90% accuracy** hypothesis.
- [ ] 14. Refactor core processing functions into clean, decoupled Python modules, preparing for easy migration into **AWS Lambda functions** (e.g., `ocr_processor.py`, `llm_extractor.py`).