# Case Study 1 — Project LANTERN (Part 1)

## Building a Financial Report Parsing Pipeline

### Setting

FinTrust Analytics helps analysts make sense of 10-K/10-Q filings. Today, analysts manually download SEC filings and dig through hundreds of pages of text, tables, and figures. The process is slow, repetitive, and error-prone.

Lina Chen, VP of AI Platforms, launches **Project LANTERN** to automate ingestion and parsing of SEC filings. Her immediate goal: create a **reproducible pipeline** that can:

- Download filings,
- Extract text, tables, and layouts,
- Attach metadata and provenance,
- Validate extracted values against structured XBRL,
- Benchmark costs and performance,
- Package everything under **Data Version Control (DVC)**.

By the end of Part 1, Lina expects her team to deliver a **layout-aware, XBRL-validated corpus** in Markdown and JSONL formats — reproducible, versioned, and ready for retrieval.

---

### Phase 1: Design (Labs 0–4)

**Key requirements:** coverage of text, tables, images; provenance; reproducibility; accuracy.

- **Lab 0**: Bootstrap a Git repo, directory structure (`src/`, `data/raw/`, `data/parsed/`, etc.), and fetch filings via `sec-edgar-downloader`. Include XBRL attachments.
- **Lab 1**: Parse text with `pdfplumber`, falling back to Tesseract OCR when needed. Persist word boxes for analysis.
- **Lab 2**: Extract tables using Camelot (lattice vs. stream) and `pdfplumber` table finder. Save CSVs and note method trade-offs.
- **Lab 3**: Run **LayoutParser** to detect block types (text, titles, tables, figures). Route each block accordingly; persist bounding boxes.
- **Lab 4:** Explore **Docling** for advanced PDF understanding — reading order, tables, formulas, unified DoclingDocument. Export to Markdown/JSON and compare against the traditional pipeline.

---

## Phase 2: Representation & Staging (Labs 5–8)

- **Lab 5**: Create a metadata schema (`doc_id`, `page`, `section`, `bbox`, etc.). Save `.jsonl` files and Markdown with provenance.
- **Lab 6**: Compare Markdown, JSON, and TXT outputs. Decide which format best supports downstream retrieval.
- **Lab 7**: Run a **Build vs. Buy** spike using AWS Textract and Google Document AI and Azure AI Document Intelligence. Compare accuracy and cost.
- **Lab 8**: Orchestrate everything under **DVC** with stages (`download → parse_pdfplumber → parse_docling → tables → export`). Version both code and data.

---

## Phase 3: Evaluation & Validation (Labs 9–11)

- **Lab 9**: Build a ground-truth set. Compute **word-error rate (WER)** for text and **precision/recall** for table cells. Add regression tests.
- **Lab 10**: Benchmark runtime, memory, and throughput. Estimate costs for both pipelines (open-source vs. managed APIs).
- **Lab 11**: Parse XBRL files with Arelle or python-xbrl. Align values with PDF-extracted tables and produce validation reports. Investigate mismatches.

---

## Outcome of Case Study 1

By the end of Part 1, the FinTrust AI team has:

- A **reproducible ingestion pipeline** tracked by DVC,
- Outputs in Markdown and JSONL with full provenance,
- Evaluation metrics and benchmark reports,
- Validation of extracted numbers against official XBRL,
- Clear trade-offs between **Docling** and the traditional path.

This corpus is **ready for retrieval**, but embeddings and question-answering will come in Case study 2.

# AI-Powered PDF Parsing System

Due: September 26th 03:59 pm EST

## Submission:

1. Github Repo Link
2. A 10 video recording of the demo posted on an Online platform and link added to Readme

## Additional notes:

1. Required attestation and contribution declaration on the GitHub page:
WE ATTEST THAT WE HAVEN'T USED ANY OTHER STUDENTS' WORK IN OUR ASSIGNMENT AND ABIDE BY THE
POLICIES LISTED IN THE STUDENT HANDBOOK
   - member1: 33.3%
   - member2: 33.3%
   - member3: 33.3%
2. Make sure you do not push anything to your GitHub after submission date (Editing Readme.md is ok but no code
pushing after deadline)
3. Create a Codelab document describing everything you did. In your GitHub you should have a readme.md files which would tell what all things are there in this GitHub repository.
4. Keep your repository private until the deadlines. Incase of any plagiarism cases both the teams which be equally held responsible.

## Summary:

**Ingestion, Parsing & Embedding:**

You are expected to download SEC filings, extract text, tables and layouts, explore **Docling** for advanced document understanding, add metadata, choose storage formats, compare managed services, version their pipeline with DVC, evaluate accuracy, benchmark performance, validate with **XBRL** data and create semantic embeddings. By the end of this case study, you will have a reproducible ingestion and parsing pipeline ready for retrieval.

---

# Part 1 – Ingestion, Parsing & Embedding

## Part 0 — Course repo & dataset bootstrap (SEC filings)

**Goal:** Bootstrap a reproducible project and download a small set of SEC 10-K/10-Q filings. This mirrors the assignment's requirement to source data from EDGAR.

- **Setup:** Create a Git repository with directories like `src/`, `data/raw/`, `data/parsed/`, `notebooks/` and `reports/`. This lays the foundation for reproducible work.
- **Data download:** Use an official or helper library such as **sec-edgar-downloader**. The helper requires a company name and email address to construct a compliant User-Agent string so the SEC's EDGAR system does not throttle the crawler[1]. Download a handful of filings for one company over two fiscal periods and store the PDFs in `data/raw/`.

**Checkpoints:**

- Repository has a clear directory structure and a README explaining reproduction steps.
- `data/raw/` contains a few filings (<100 MB total).
- Download accompanying XBRL attachments. These will be used later to cross-validate extracted tables.

## Part 1 — Text extraction from PDFs (pdfplumber + OCR fallback)

**Goal:** Extract per-page text while preserving reading order and handle scanned pages with OCR. Students learn the limitations of simple parsing and how to recognise when to fall back to OCR.

**Core tasks:**

- Use **pdfplumber** to iterate through pages. Call `Page.extract_text`, which collates character objects into strings and inserts spaces and newlines based on character positions; it also supports experimental layout parameters `x_density` and `y_density`[2].
- Detect pages where no text is extracted and apply **Tesseract** (via `pytesseract`) to perform OCR. Save each page's text to `data/parsed/` and note which pages required OCR.

**Checkpoints:**

- Per-page `.txt` files exist for each PDF.
- A log records pages that needed OCR.
- Persist word bounding boxes using `page.extract_words()`; this can help with chunking and layout analysis.

## Part 2 — Table extraction (Camelot + pdfplumber table modes)

**Goal:** Extract structured financial tables and compare different methods. Financial statements often have borderless tables or complex layouts that challenge heuristics.

**Core tasks:**

- Use **Camelot** to detect tables. Try both `lattice` mode (which relies on ruling lines) and `stream` mode (which infers columns by grouping text spans). Compare outputs.
- Compare Camelot's output with **pdfplumber**'s table detection, which finds lines, merges overlapping segments, identifies intersections and groups cells into tables[3].
- Save each extracted table as CSV and note which method best preserves row and column structure.

**Checkpoints:**

- At least one clean CSV of a balance sheet or income statement.
- A brief analysis describing why one method was preferred (e.g., borderless tables work better in stream mode).
- Create a hybrid extractor that chooses lattice or stream based on simple heuristics (presence of ruling lines) or merges outputs from both.

## Part 3 — Layout detection for complex pages (LayoutParser)

**Goal:** Use deep learning to detect document structure—headings, paragraphs, images and tables—before extraction. This is essential for multi-column reports and ensures accurate reading order.

**Core tasks:**

- Install **LayoutParser** and load a pre-trained model (e.g., `PubLayNet/ppyolov2`). LayoutParser provides a unified API for extracting complicated document structures with just a few lines of code[4].
- For each PDF page, detect layout blocks. Each block is a `TextBlock` object containing bounding box coordinates and a type label such as "Text", "Title", "Table" or "Figure"[5].
- Visualise bounding boxes to understand page structure. Route text blocks to pdfplumber/OCR, tables to Camelot and figures to an image storage module. Store the type and bounding box in metadata for provenance.

**Checkpoints:**

- JSON output listing page number, block type and bounding boxes.
- Demonstrated layout-aware extraction (e.g., multi-column text flows correctly).
- Experiment with multimodal models like LayoutLMv3 for tasks such as caption extraction. This optional exploration is further developed in Case study 2.

## Part 4 — Advanced PDF understanding with Docling

**Goal:** Explore **Docling**, a specialised library for advanced PDF understanding and content normalisation. Docling provides features like page layout and reading order detection, table and formula extraction and unified document representations. Students learn when Docling can replace or augment earlier methods.

**Core tasks:**

- Install **Docling** and load a PDF into a `DoclingDocument`. Docling offers advanced PDF understanding: it recovers page layout and reading order, table structure, code and formulas; it presents a unified `DoclingDocument` representation and can export content as Markdown or JSON[6].
- Parse one or two filings with Docling. Examine how Docling identifies complex structures compared to your pdfplumber + LayoutParser pipeline. Pay attention to reading order (does it handle multi-column flows? footnotes?), tables (does it merge cells correctly?) and formulas.
- Convert the `DoclingDocument` into Markdown and JSON. Compare these outputs to the ones you generated with custom code. Note any differences in structure, completeness or fidelity.
- Explore Docling's plug-and-play integrations. Docling can be run locally or as a server and integrates with other pipelines[6].

**Checkpoints:**

- A notebook or script that loads a PDF with Docling and exports Markdown/JSON.
- A comparison note highlighting where Docling excels (e.g., formula detection) and where your custom pipeline still performs better.
- Investigate how to plug Docling into your DVC pipeline as an alternative parsing stage. Evaluate performance and accuracy trade-offs.

## Part 5 — Metadata & provenance tagging

**Goal:** Attach provenance metadata to every extracted piece of text or table. This enables traceability in downstream QA and ensures that answers can cite exact locations.

**Core tasks:**

- Define a metadata schema: `doc_id`, `company`, `fiscal_year`, `page`, `section`, `block_type`, `bbox`, `text`, `source_path`, etc.
- As you parse each block (from LayoutParser or Docling), emit a JSON record following the schema. Save all records in a `.jsonl` file per document.
- Create a simple function to reassemble a report section by grouping records with the same section label; store this as Markdown with headings and tables for readability.

**Checkpoints:**

- `.jsonl` files exist for each document with consistent keys.

- Markdown files summarise each section.

## Part 6 — Storage formats: Markdown vs JSON vs TXT

**Goal:** Decide how to represent parsed content. Students will see the trade-offs between human-readable Markdown, machine-readable JSON and plain text.

**Core tasks:**

- Convert parsed blocks into Markdown, preserving headings, lists and tables. Markdown retains semantic structure and is well understood by LLMs, making it effective for RAG pipelines.
- Convert the same content into JSON objects (with keys for text, table rows, metadata). JSON is ideal for programmatic access.
- Save a plain text version as a baseline; note how structure is lost.

**Checkpoints:**

- Three formats exist for at least one parsed document.
- A short note explains which format you plan to use in your pipeline and why (e.g., Markdown for RAG because it preserves section context).

## Part 7 — Build vs Buy experiment: AWS Textract & Google Document AI & Azure AI Document Intelligence

**Goal:** Understand the pros and cons of managed document extraction services. While your open-source pipeline may suffice for most filings, commercial APIs can be valuable for difficult cases.

**Core tasks:**

- Run the same pages through **Amazon Textract** (AnalyzeDocument API) or **Google Document AI** (Document OCR / Form Parser) or **Azure AI Document Intelligence**. These services extract text and tables with high accuracy and output JSON. Note costs and data-privacy considerations.
- Compare table structure and OCR quality to your open-source pipeline.
- Document pricing models (per-page costs) to appreciate cost considerations.

**Checkpoints:**

- Side-by-side comparison of at least one page or table.
- A brief discussion on whether to incorporate managed services in the pipeline (e.g., as a fallback for scanned documents).
- Integrate one of these services as an optional fallback in your parser.

## Part 8 — Staging pipeline & versioning with DVC

**Goal:** Turn your ad-hoc scripts into a reproducible pipeline and version both code and data.

**Core tasks:**

- Install **Data Version Control (DVC)**. DVC manages and versions large files and models and organises machine-learning workflows into reproducible pipelines[7].
- Create a `dvc.yaml` with stages: `download` → `parse` → `tables` → `layout` → `docling` → `export`. Define dependencies and outputs. Running `dvc repro` will execute the pipeline end-to-end and cache intermediate results.
- Commit `dvc.lock` and `.dvc` files to git to preserve data lineage.

**Checkpoints:**

- A working DVC pipeline that reproduces parsed outputs from raw PDFs.
- Data and model artifacts are stored and versioned.
- Configure a GitHub Actions workflow to run a smoke test on every pull request.

## Part 9 — Evaluation: parsing quality & regressions

**Goal:** Measure how well your parser extracts text and tables, and catch regressions when you make changes.

**Core tasks:**

- Create a small ground-truth dataset: manually transcribe a few pages of text and key financial tables.
- Compute word-error rate (WER) or character-error rate for text extraction. For tables, compute cell-level precision and recall by comparing your CSV output to ground truth.
- Write unit tests that assert metrics stay above a threshold.
- Use `dvc metrics diff` or a custom script to detect when metrics degrade after code changes.

**Checkpoints:**

- A report summarising extraction accuracy.
- Tests that fail when you intentionally break the parser.
- Visualize distribution drift (e.g., chunk lengths or numeric token ratios) using `matplotlib`.

## Part 10 — Cost & throughput benchmarking

**Goal:** Understand performance and cost trade-offs. Scaling from a few filings to thousands can require careful planning.

**Core tasks:**

- Measure runtime per page and memory consumption for your pipeline on a representative batch (e.g., 50–100 pages). Log failures (pages with empty output or parsing errors).
- Estimate cost if using cloud APIs (Textract/Document AI). Use public pricing to compute per-page or per-document costs.
- Identify bottlenecks (OCR, table extraction, layout detection, docling) and note how they scale with page complexity.

**Checkpoints:**

- A `benchmarks.md` file with timing and cost estimates.
- Recommendations for hardware (CPU vs GPU) or concurrency to handle large volumes.

## Part 11 — XBRL extraction & validation

**Goal:** Cross-validate key numbers extracted from PDFs with structured XBRL data. Students learn to use tools like **Arelle** or `python-xbrl` to parse XBRL files and compare them with the tables extracted from PDFs.

**Core tasks:**

- Use `sec-edgar-downloader` or the EDGAR API to download XBRL attachments for the same filings. Many 10-K/10-Q filings include machine-readable XBRL (XML) files alongside PDFs.
- Parse the XBRL file using **Arelle**, `python-xbrl` or another XBRL library. Extract key financial line items (e.g., Revenue, Net Income, Total Assets) and store them in a DataFrame.
- Align your parsed PDF tables (from Part 2/Docling) with the XBRL concepts by mapping table labels to XBRL taxonomy names. This may require building a mapping dictionary.
- **Cross-verification:** Validate that numerical values in your CSV tables match those in the XBRL file. A DataTracks guide recommends cross-verifying extracted values against the original financial statements and applying validation rules to ensure consistency and accuracy[8]. When extracting XBRL, always check that values align with the original statements and employ validation rules to catch discrepancies[9].
- Report any mismatches or discrepancies. Investigate whether the discrepancy stems from OCR errors, table parsing mistakes or tagging differences.

**Checkpoints:**

- A notebook that loads an XBRL file, extracts numeric values and compares them to your parsed tables.
- A summary of matches and mismatches; discussion on potential causes and fixes.
- Automate the mapping between PDF table labels and XBRL concepts across multiple filings. Explore using natural language similarity or taxonomy lookup to automate matching.

---

**References:**

[1] sec-edgar-downloader — SEC EDGAR Downloader 5.0.3 documentation

https://sec-edgar-downloader.readthedocs.io/en/latest/

[2] [3] GitHub - jsvine/pdfplumber: Plumb a PDF for detailed information about each char, rectangle, line, et cetera — and easily extract text and tables.

https://github.com/jsvine/pdfplumber

[4] [5] Deep Layout Parsing — Layout Parser 0.3.2 documentation

https://layout-parser.readthedocs.io/en/latest/example/deep_layout_parsing/

[6] Docling - Docling

https://docling-project.github.io/docling/

[7] Data Version Control · DVC

https://dvc.org/

[8] [9] As short article about How to Extract, See and Use XBRL Data

https://www.datatracks.com/us/blog/extract-see-use-xbrl-data/

[10] [11] [12] whylogs Overview | WhyParts Documentation

https://docs.whylabs.ai/docs/whylogs-overview/

[13] Weights & Biases (W&B) Explained | Ultralytics

https://www.ultralytics.com/glossary/weights-biases

[14] Track experiments | Weights & Biases Documentation

https://docs.wandb.ai/tutorials/experiments/