ChatGPT

FDD Extraction Pipeline Documentation Suite

README

Project Purpose: This repository provides a local pipeline for extracting structured data from Franchise Disclosure Documents (FDDs). It automates downloading FDD PDFs from state websites, parsing their content, and using Large Language Models (LLMs) to extract key information into JSON. The goal is to enable analysts and downstream applications to query franchise data (fees, terms, etc.) without manually reading PDFs. All processing runs locally: Anthropic's Claude (via Claude Code CLI) is the primary AI for extraction, with OpenAI's models as fallback for robustness.

Local Development Setup: Ensure you have Python 3.10+ and Node (for Playwright). Clone the repository and install dependencies (consider using a virtual environment):

```
git clone https://github.com/your-org/fdd-pipeline.git
cd fdd-pipeline
pip install -r requirements.txt
playwright install  # installs browser engines for Playwright
```

Create a `.env` file in the project root with configuration variables (see *Environment Management* below). Key variables include API keys for Anthropic Claude and OpenAI, and Supabase credentials. We centralize config in `config.py` via Pydantic BaseSettings, instead of scattering `os.getenv` calls in code. For example, `ANTHROPIC_API_KEY`, `OPENAI_API_KEY`, `SUPABASE_URL`, and `SUPABASE_SERVICE_KEY` should be set in `.env`. The Pydantic settings class will load them on startup (Pydantic can read `.env` files by specifying `env_file` in the settings config). This approach yields a clear, typed config object and avoids using `os.environ` throughout the codebase.

Quickstart: After installing and configuring, you can run the pipeline to fetch and process FDDs:

1. **Run the Scraper:** Launch the weekly scraping job. For development, you can invoke the Prefect flow or FastAPI endpoint that kicks off scraping. For example, if using Prefect locally, start the Prefect Orion server and an agent, then register and run the `weekly_scrape` flow (or simply run the script `scrape_flow.py` if provided). This will use Playwright to load state regulatory websites, navigate to FDD filings pages, and download new PDFs. The code uses headless Chrome by default, so no GUI opens during scraping.

2. **Process New Filings:** The pipeline will detect newly downloaded PDFs and process each. A Prefect flow (or sequence of tasks) will handle each file: first performing layout analysis with MinerU, then extracting structured data via Claude. This can be triggered automatically after scraping (as part of the flow) or manually by running the `process_new_filings` flow. Ensure the Claude and OpenAI API keys are set; the extraction service will call Anthropic's Claude model by default. Extraction

results are validated against a Pydantic schema, stored in the database, and the JSON output can be saved to Supabase storage or a local folder for inspection.

3. **View Results:** Once processed, the structured outputs (JSON) are available via the Supabase database. You can connect to the Postgres DB (using Supabase credentials) or query via supabase-py in a Python shell to see extracted fields. If an optional FastAPI service is running (see *Deployment*), you might also retrieve data through API endpoints.

Repository Structure: The project follows a **3-tier documentation system** to assist AI and developers. High-level docs live in `docs/ai-context/` (Tier 1) and component-specific docs (Tier 2) reside alongside code in module folders, with feature-level notes (Tier 3) near implementation. This structure helps our Claude Code assistant load only relevant context for a task, improving accuracy and efficiency. For example, a global `CLAUDE.md` file at the repo root contains coding standards and key info that Claude always loads. Component-level `CONTEXT.md` files outline each subsystem (scraper, extractor, etc.), and feature-level `CONTEXT.md` files capture detailed decisions and patterns in specific modules. New contributors or AI agents should start with the Project Overview below, then dive into component docs as needed.

Project Overview

System Scope & Architecture

This pipeline automates the end-to-end processing of FDD documents using a **modular, orchestrated workflow**. The major stages are: **Web Scraping**, **Document Layout Analysis**, **LLM-based Data Extraction**, and **Data Storage**. A workflow orchestrator (Prefect) coordinates these stages, invoking each component in sequence and handling scheduling and errors. The system is designed for weekly runs to capture newly published FDDs and maintain historical versions for each franchise.

Data Ingestion (Web Scraping): Every week, the pipeline visits state regulator websites that publish FDD filings. Using **Playwright** (a headless browser toolkit) in Python, it loads dynamic pages (handling login or form interactions if necessary) and finds links to FDD PDF files. Playwright's headless Chrome mode enables scraping of JavaScript-heavy sites that traditional `requests` + `BeautifulSoup` cannot handle. Once page HTML is loaded, **BeautifulSoup** is used to parse the DOM for PDF links and metadata (filing dates, franchise names, etc.). The scraper downloads each PDF to local storage (or directly uploads to Supabase storage) and records metadata in the database. The scraping stage is robust: it includes retry logic for page loads and downloads. Playwright's built-in wait/retry mechanisms help deal with dynamic content and ensure we capture all relevant files.

Document Parsing (Layout Analysis): For each PDF, we use **MinerU** (a Vision-Language Model tool by Opendatalab) to convert the document into a machine-readable JSON structure. MinerU performs OCR and layout analysis, segmenting each page into text blocks, lines, and spans with their bounding box coordinates. This yields a rich JSON output conforming to a predefined schema (see `MinerU_Json_Schema_With_Comments.json`). The schema includes page-level info and a hierarchy of blocks → lines → spans, with each element carrying text content and its position on the page. By enforcing a consistent JSON layout schema, we preserve tables, lists, and section headings structure from the PDF. This consistency is crucial for the LLM extraction step to reliably identify sections and fields across different

FDDs. *(MinerU has been proven effective in similar use cases – for instance, one user processed 800 PDFs with it entirely locally, demonstrating its suitability for large-scale OCR tasks.)*

AI Extraction (LLM Processing): The core of the pipeline is using an LLM to extract key FDD data points from the text. We primarily use **Claude** (via Anthropic's API) in a special "Claude Code" mode configured for CLI usage. Claude is prompted with the text (or relevant sections) of the FDD and instructed to output a JSON object containing specific fields (e.g., franchise fees, renewal terms, etc.) according to a Pydantic schema. We employ the **Instructor** library to facilitate this structured output generation. The prompt provided to Claude precisely defines the required JSON format and keys, sometimes even including a brief example of the desired JSON structure. This explicit formatting instruction helps increase consistency of Claude's outputs. If Claude's response fails schema validation (e.g., missing a field or format issue), the pipeline will log the validation error and **auto-retry** the prompt with adjusted instructions or formatting hints. It will attempt up to 3 retries on Claude (common errors like slight format deviations are often corrected on a second attempt). We log each failed attempt's output and error to both the console and a Supabase table for debugging and quality tracking. After 3 failures on Claude, the system falls back to **OpenAI's GPT-4** with function calling as a backup. GPT-4 is invoked with the same schema (passed as a function definition) to leverage its function-call JSON enforcement. This two-tier LLM strategy (Claude first, GPT-4 fallback) maximizes accuracy and reliability of extraction, as recommended in production AI systems to handle model variance.

Data Storage & Access: Extracted data and files are stored using **Supabase**, a platform that provides a Postgres database and object storage. We use the official `supabase-py` client to interact with both the DB and storage in a unified way. Each processed FDD gets an entry in a "Filings" table (or similar) in the Postgres DB, with fields like franchise name, year, filing date, and JSON data of extracted fields. The actual PDF and possibly the MinerU-produced JSON are saved in Supabase Storage buckets (with URLs or references stored in the DB). By using Supabase, we get an **integrated solution** for structured data and files, accessible through a single client library. We also use SQLAlchemy models in our code to define the database schema and allow complex queries or analytics on the extracted data. A lightweight FastAPI application may expose read-only endpoints to query this data (for example, fetching all data for a given franchise, or diffing two years of an FDD) for internal use or future integration with frontends.

Workflow Orchestration: The pipeline is glued together by **Prefect**, a Python workflow orchestration framework. Prefect coordinates scheduling the weekly scraping, parallelizing file processing, and handling failures. The architecture is event-driven: once scraping finishes and new PDF metadata is saved, Prefect can trigger downstream processing flows for each new file (or a batch flow for all). Each major step (scrape, parse, extract, store) is a Prefect *task*, and the overall logic is a Prefect *flow*. Prefect's task runner ensures that if any step fails (e.g., MinerU raises an error on a PDF), that task's state is recorded and can be retried or inspected without crashing the entire pipeline. We also take advantage of Prefect's scheduling to run the `scrape_and_process` flow every week at a set time, and its caching to avoid re-processing files we've seen before. The use of **FastAPI services** for some components means that Prefect might call HTTP endpoints as tasks (e.g., an internal endpoint for the scraper service to start a run, or an endpoint on the extraction service to process a given file). This decoupling via API calls ensures that each component can run in its own process or container, and Prefect just orchestrates the high-level sequence. The design follows a common pattern: using a FastAPI web app to trigger background tasks through Prefect, which is useful for long-running processes and monitoring.

Historical Tracking: One important feature is maintaining historical versions of FDD filings. The pipeline does not assume one unique FDD per franchise – franchises update their FDDs yearly (or more often), and multiple states might publish the same document. We keep **all versions** and label them by year and effective date so that users can compare changes year-over-year. For example, if "ABC Franchise – 2024 FDD" and "ABC Franchise – 2023 FDD" are both scraped, both remain in our database. In fact, if a state publishes an amended FDD mid-year, we would capture that as well (with an appropriate timestamp or version indicator). This aligns with best practices seen in franchise data platforms, where historic FDDs are retained for trend analysis. The system uses the franchise name and year (and a document issue date, if available in metadata) as part of a unique key to identify duplicates and versions.

Technology Stack

- **Python 3**: Core language for all components.
- **Playwright** (Python) and **Requests/BeautifulSoup**: for web scraping. Playwright handles dynamic content and simulates a browser, while BeautifulSoup parses static HTML.
- **MinerU (opendatalab)**: for PDF OCR and layout analysis, producing structured JSON of document content.
- **Anthropic Claude (via Claude Code)**: primary LLM for information extraction. Claude Code CLI is used to integrate Claude into our dev workflow, automatically pulling in repository context (see *Claude Integration* below).
- **OpenAI GPT-4**: fallback LLM for extraction, using OpenAI's function calling to enforce JSON output when Claude's output doesn't validate.
- **Pydantic & Instructor**: Pydantic defines data models and auto-generates JSON schemas for LLM prompts. Instructor library wraps LLM calls, enabling us to get a Pydantic-validated object directly from a completion and providing utilities like automatic re-prompting.
- **Prefect**: orchestrates flows and tasks, enabling scheduling, parallel execution, and retry logic.
- **FastAPI**: provides a lightweight API layer for triggering pipeline tasks and possibly serving results. Each major component can run as a FastAPI service (for example, a "scraper API" that, when called, initiates the scraping routine, and an "extractor API" that processes a given PDF by calling MinerU and LLM). These services make the system modular and easier to scale or deploy independently.
- **Supabase (Postgres + Storage)**: persists extracted data and raw files. Supabase's Postgres database (queried via supabase-py and SQLAlchemy) holds structured data, and Supabase Storage holds PDFs and large JSON artifacts. Supabase was chosen as it offers a seamless integration of database and file storage with a Python client, and because it's an open-source Firebase alternative well-suited for Python stacks.

Claude Integration & Documentation Strategy

Because this project heavily leverages Anthropic's Claude (especially via the Claude Code CLI for development), we have aligned our development practices with Claude's strengths. We maintain a top-level `CLAUDE.md` in the repo that includes important context and instructions that Claude should always consider (such as project conventions, key file descriptions, and coding style guidelines). Claude Code automatically loads this file at session start, ensuring the AI always has the fundamental project knowledge. The rest of our documentation follows the **three-tier system**: foundational docs (like this overview, architecture, standards) change rarely and give Claude and developers a stable context. Component-level docs outline how different parts (scraper, extractor, etc.) are designed and integrate, and are loaded when working in those areas. Feature-level docs reside near code to capture specifics without cluttering higher-level docs. This structured approach means our AI assistant can load just the relevant CONTEXT for a given

task – e.g., if we ask Claude to help modify the scraping logic, it will load the scraping component's context (Tier 2) and perhaps a feature doc on, say, "Handling infinite scroll pages" (Tier 3), rather than the entire project docs. This **intelligent context loading** improves both AI efficiency and accuracy. For human team members, the documentation tiers provide quick orientation: new engineers read the Tier 1 docs for a high-level understanding and standards, then delve into Tier 2/3 docs as they start modifying specific components.

System Integration Overview

This section describes how the components interact and how data flows through the system. The integration design emphasizes loose coupling between components via clear data contracts (schemas) and orchestration, which simplifies maintenance and scaling.

End-to-End Data Flow: The pipeline can be visualized as a sequence of steps, each handled by a specific component or service, with Prefect orchestrating the transitions:

1. **Initiation:** Prefect's scheduler triggers the **Scraper Service** (or a Prefect task calls the scraping function) on a weekly schedule. Alternatively, an engineer can start a run manually via a FastAPI endpoint (e.g., `POST /scrape` on the scraper service triggers an immediate run).
2. **Web Scraping Output:** The Scraper Service logs into or navigates each state site and finds new FDD PDF links. For each discovered file, it downloads the PDF content. After downloading a PDF, the scraper emits a message or triggers a callback (could be writing a DB entry or an event) containing the file's metadata (franchise name, year, state, date, etc.) and storage path. This event is picked up by the **Orchestrator**.
3. **Deduplication Check:** Before processing a new PDF, we perform a deduplication check. The Orchestrator (or a dedicated Deduplication module) queries the database to see if a similar filing already exists. The check uses fuzzy matching on the combination of franchise name and year, and possibly a content hash or RapidFuzz similarity on text snippets. If a duplicate is detected (e.g., the same franchise/year was already processed from another state or earlier run, with >90% text similarity), we mark the new file as duplicate and skip further processing for it. (Alternatively, we could still store it under the originating state's record but not run extraction again. The system is configurable to either skip exact duplicates or record them as separate entries linked to the original.)
4. **Layout Analysis Stage:** For each new, non-duplicate PDF, the Orchestrator calls the **Parser Service** – which might be an internal function or a separate FastAPI service for document parsing. This service runs the MinerU tool on the PDF to produce a JSON of the layout. The Parser returns the JSON (or a path to it) back to the Orchestrator. The Orchestrator then passes that JSON (or the extracted plain text content) to the next stage.
5. **LLM Extraction Stage:** The Orchestrator invokes the **Extraction Service**. This could be a FastAPI service running the LLM prompts, or a Prefect task that directly calls the LLM APIs. The input is the content from the parser: in practice, we often feed the LLM a text version of the FDD (extracted from the MinerU JSON). In some cases, we might insert specific markers or section headings (leveraging the structured JSON) to help the LLM identify key sections (Item 1-23 of the FDD, which are standardized sections). The Extraction Service loads the appropriate **prompt template** for FDD extraction from the repository (prompts are stored in the `prompts/` directory of the codebase for version control). It also loads the Pydantic **schema definition** for the expected output. Using the Instructor library, it calls Claude's API with the prompt and attaches the JSON schema in a way Claude can use (for OpenAI, this would be function parameters; for Claude, we include the schema

description in the prompt since Claude doesn't natively support function calling). Claude processes the prompt and returns a completion which the service attempts to parse/validate as the Pydantic model. If validation fails, the service may log an error explaining the mismatch (e.g., "field X missing") and then either adjust the prompt or simply prompt Claude again with an emphasis on the missing piece. The integration here is designed such that the LLM is aware of the schema *before* generating the answer, minimizing invalid outputs. After up to 3 tries, if a valid JSON object matching our schema is produced, we consider the extraction successful. If Claude cannot get it right, the service will now call OpenAI's API. With OpenAI, we use the function calling feature by constructing a function schema from the same Pydantic model (Pydantic can output a JSON Schema that we convert to the OpenAI function syntax). We then prompt GPT-4 in a similar manner; GPT-4 is very likely to return a correct JSON on the first attempt due to the rigid function spec. We parse that and continue. If even the fallback fails (extremely rare for known schema fields), the pipeline logs the failure and moves on, ensuring one stubborn file won't hold up others.

6. **Storage and Acknowledgment:** After extraction, the Orchestrator persists the results. The structured data is written to the Supabase Postgres (via SQLAlchemy models). We use a transaction per file, so all extracted fields for that document get saved atomically. The raw text or the MinerU JSON can be stored in a separate table or bucket (for audit or future use). The PDF itself, if not already in Supabase storage, is uploaded at this point (the scraper might have saved it to a local path; we now upload it to a Supabase storage bucket for permanent retention). We then update the "Filings" table record for this file to mark it as processed and attach foreign keys or links to the extracted data and file storage.
7. **Downstream Notifications:** With the new data stored, the pipeline can optionally notify other systems. For example, if there's an analytics dashboard or a search index, a message can be sent (perhaps via a message queue or a direct API call) to let it know new franchise data is available. In the current local setup, this might simply be logging and perhaps email notification to the team. Prefect's orchestration ensures that any failures in prior steps trigger retries or alerts. We have healthchecks set up so that if the weekly run fails, the team is notified to intervene.

Throughout this flow, **Prefect** is the conductor ensuring each part runs in order and handling errors. Each component (Scraper, Parser, Extractor) can be developed and tested in isolation – they communicate through well-defined inputs/outputs (files, JSON, database records). This separation follows a microservice-like integration pattern, which improves resilience (e.g., the Scraper can fail or be turned off without preventing manual re-processing of known PDFs; the Extractor can be updated independently to fix prompt issues and then re-run on stored PDFs if needed).

Cross-Component Patterns: Integration between components uses a mix of direct function calls and REST API calls. For example, within a Prefect flow running in a single process, we might call the MinerU parser via a Python function (if it's just a library call). But for the scraper and extraction, we treat them as services (the scraper opens a browser context, which is easier to manage as a separate process or container; the extraction with LLM calls can be memory heavy, so we separate it too). We thus use HTTP calls between the Orchestrator and these services. FastAPI makes this lightweight, and we keep the payloads lean (e.g., for extraction we send just necessary text or a reference to where the parser output is stored rather than the entire PDF content). Authentication between internal services is kept simple (they run on a trusted network or localhost in deployment, and we could use a shared secret or token if needed for security).

Data Formats: JSON is the lingua franca between components. The scraper outputs a JSON (or Python dict) of discovered filings (with fields like name, state, year, URL, file_path). The parser outputs JSON per the MinerU schema. The LLM outputs JSON per our extraction schema. Using JSON everywhere (and defining

schemas for each via Pydantic) ensures minimal transformation friction. Each component knows what JSON to expect and produce, which also makes it easier for testing (we can swap real components with dummy JSON fixtures to test downstream parts).

Error Handling & Logging: Integration-wise, if a component fails, Prefect will catch the exception or non-zero exit and mark the flow accordingly. We have set up retries for transient errors: e.g., if the Scraper times out on a state site, it can retry that site a couple of times. If MinerU fails on a PDF (which might happen for an unreadable PDF), we catch that, log the file name, and continue with others – such a file might be flagged for manual review. All errors and important events are logged to both console and Supabase (we use a `logs` table in the DB or simply rely on Prefect's own logging). For the LLM part, we specifically log validation errors (including the incorrect output snippet for analysis) – those are stored in a table for later improvement of prompts or schema. This integrated logging means the on-call developer (or an AI agent reviewing system status) can query the database to find all files that failed in a run and why. We also plan to compile recurring issues in the documentation, as part of the *Handoff & Continuity* process, so we can quickly apply known fixes.

Performance Considerations: Integrating these components locally requires careful resource management. The Scraper (with headless browsers) and the MinerU OCR can be CPU/memory intensive, and the LLM calls have latency (especially if going over network to external API). We mitigate this by pipelining where possible – e.g., as soon as one PDF is downloaded, we can start parsing it while others are still downloading (Prefect can run tasks concurrently if configured). Similarly, we can have multiple extraction tasks in parallel as long as we mind the rate limits of the LLM APIs. We configured Prefect to use task concurrency and even multi-threading for I/O-bound parts. Each component is largely I/O bound (scraping waits on network, LLM waits on API), so parallelism can improve throughput. We do impose a limit (like 3 parallel LLM calls) to avoid hitting API rate limits or exhausting memory. The FastAPI services are run with Uvicorn workers so they can handle multiple requests if needed (e.g., multiple extraction tasks calling the extraction service). Database writes are batched by the Orchestrator at the end of processing each file to reduce overhead.

In summary, the system integration is designed such that each piece can be understood in isolation but they work in concert through structured data exchange. The orchestrator (Prefect) is aware of the big picture and ensures the pieces line up, but each component handles its domain-specific logic (website quirks in scraper, document structure in parser, language understanding in extractor, etc.). This modular integration will also allow future enhancements – for example, if we switch out MinerU for a different parser or add a new state source, we can do so without overhauling the entire pipeline, as long as we adhere to the agreed data contracts between stages.

Component Specifications (Tier 2)

Below are the component-level documentation for the major subsystems of the pipeline. Each section outlines the component's purpose, how it works internally, and how it interfaces with other components.

Web Scraper Component (Data Acquisition)

Purpose & Overview: The Web Scraper component automates the retrieval of FDD PDFs from various state websites. It addresses the challenges of different web layouts, authentication, and dynamic content to

ensure we consistently capture new filings. This component is crucial as the entry point of fresh data into the system.

Current Status: **Stable.** The scraper currently supports a set of priority states (those that publish FDDs online). It handles both simple index pages and more complex search forms. As of now, it's running headlessly on a weekly schedule and reliably downloading files. Recent improvements include adding a headless browser for JavaScript rendering and a retry mechanism for robustness (e.g., if a site is down, it tries again after a delay).

Tech & Tools: We use **Playwright** for browser automation, chosen for its multi-browser support and reliable API. Playwright's ability to auto-wait for page readiness and its robust context handling (like dealing with new pages or downloads) has been leveraged. For simpler sites (where no JavaScript or login is required), we fall back to `requests` + `BeautifulSoup` as it's faster and lighter. The component is written in Python and can be invoked as a standalone script or via a FastAPI endpoint (`/scrape`). Configuration (like target URLs, credentials if any) is read from environment or a config file at runtime, allowing easy updates when a state site changes.

Code Organization: The scraper code resides in `src/scraper/`. Key modules include: - `scraper/main.py`: The entry point with the Prefect flow (if any) or the CLI invocation logic. It sets up the run (dates, etc.) and calls state-specific routines. - `scraper/states/` directory: Contains submodules for each state's scraping logic, e.g., `ca.py`, `ny.py`. Each defines a function to fetch all FDD links for that state (some states provide an index, others require form submission). - `scraper/browser.py`: Wrapper around Playwright to manage launching the browser, context creation, and page navigation, abstracting common tasks like clicking through pagination or waiting for elements. - `scraper/parser.py`: (Not to be confused with the PDF parser) – this is for parsing HTML content of state pages. Contains BeautifulSoup usage to extract links and metadata from HTML once loaded.

Integration Patterns: The scraper component integrates with the rest of the system by outputting a list of discovered filings. It doesn't directly push data to the database; instead, it returns Python objects (dictionaries) that the Orchestrator or a calling function uses to update the DB. This design keeps the scraper stateless (it doesn't need DB access itself, which is intentional to reduce coupling). The Prefect flow that wraps the scraper will take the list of file info dicts and then handle DB insertion (marking them as pending processing). If using FastAPI, the `/scrape` endpoint triggers the process and streams back progress or returns once done; the actual results are stored, not returned via HTTP.

Error Handling: Each state's scraping function is wrapped in try/except to catch issues particular to that site. For example, if California's site structure changes, the CA scraper might throw an error – we catch it, log a clear message, and continue with other states. We use structured logging (each log message includes state code and context) to aid debugging. Network errors are retried with exponential backoff. There's also a global timeout per site to avoid one slow site hanging the whole run. The component has built-in health checks: if a state hasn't returned any filings in a long time, it logs a warning (perhaps the site layout changed, requiring maintenance).

Performance: The scraper tries to maximize concurrency without overloading any single site. It will launch multiple browser pages if multiple states can be scraped in parallel, but it respects politeness by adding delays if needed. For states with many filings, it handles pagination or scrolling. We've taken advantage of Playwright's asynchronous capabilities and, where safe, run multiple pages concurrently.

Quality & Testing: This component has unit tests for HTML parsing functions (using saved HTML snapshots of state pages) to ensure our selectors continue to find the correct links. We also do integration tests where possible: for instance, hitting a test site or a staging environment. Monitoring is also in place – if a usually busy state suddenly returns 0 files, that triggers an alert for review.

Document Parser Component (Layout Analysis)

Purpose & Overview: The Document Parser component (also called the Layout Analysis component) takes raw PDFs and produces structured JSON representations of their content. Its goal is to bridge the gap between unstructured binary PDFs and the structured data our LLM needs to see. By extracting text along with layout information, we preserve context (sections, tables, headings) that would otherwise be lost in a plain text extraction. This component enables targeted prompts to the LLM (for example, we could prompt section by section if needed, using the JSON structure).

Current Status: **In Progress.** The parser is functional using MinerU and processes documents reasonably well. However, we are iterating on it to improve accuracy of certain elements (like table detection and reading of blurry text). The JSON schema has been defined and is mostly adhered to by MinerU's output, but we occasionally encounter minor deviations (which we normalize post-hoc). Key milestones achieved include integrating the parser into the Prefect flow and validating that for well-scanned PDFs, text content is fully captured.

Technology & Approach: We selected **MinerU** (an open-source PDF-to-JSON tool) because it leverages advanced Vision-Language Models to extract both text and structure. Internally, MinerU likely uses OCR for text and ML models for layout segmentation (headings, paragraphs, etc.). We run MinerU via its Python API (`mineru` module) or a CLI call if needed. We configured it with our custom schema (provided in `MinerU_Json_Schema_With_Comments.json`) which is based on JSON Schema Draft-07. The schema defines entities like `page`, `block`, `line`, `span` with required fields (bounding boxes, content, etc.), and MinerU is configured to output conforming JSON. For example, each page object in the JSON contains an array of blocks (paragraphs, tables, etc.), and each block contains lines and spans with textual content. This hierarchy allows us to reconstruct where each piece of text was on the page and its styling (e.g., spans might later include font info if needed).

Integration & Data Flow: This component is typically invoked by passing it a PDF file path. It returns the JSON object (or saves it to a file and returns a path). In the pipeline, the Orchestrator receives that JSON and can either pass the entire JSON to the LLM or convert it to plain text with section markers. Initially, we pass plain text to the LLM for simplicity. In future, we may improve prompt targeting by using layout (e.g., telling the LLM "within Item 7 of the FDD, find X", which we can do if we parse out Item 7's text via the JSON).

Challenges: PDFs can be tricky – images, multi-column layouts, etc. The parser sometimes has difficulty if the PDF is a scanned image (OCR errors) or if there are complex tables. We have seen issues where a table's content is captured as a single line or where columns merge. To mitigate this, we enabled MinerU's table recognition (which attempts to output tables in HTML format within the JSON spans for easy parsing). Also, our JSON schema and Pydantic validation help catch anomalies: if MinerU output doesn't match the schema (say it produces a span missing a bbox), Pydantic will flag it. We then log and attempt to fix or at least be aware of it.

Performance: Parsing is one of the more time-consuming steps (OCR can be slow). MinerU on a typical FDD (200+ pages) may take a few minutes. To keep overall throughput, we run the parser in parallel for multiple files if possible and ensure it's only done once per file. We considered caching – for instance, if we parse a PDF once and store the JSON, we won't do it again unless the PDF changed. So the pipeline first checks if a parsed JSON exists for a given file (by looking up a hash of the file in the DB or if the JSON file is present in storage) to skip re-parsing on re-runs.

Output Usage: The result JSON is not stored in the DB due to size (it can be several MB of text for a big document). Instead, we store it in Supabase Storage (or leave it on disk with an index). We do store key metadata from it, like number of pages, maybe a text snippet for quick search, etc. The JSON can be downloaded for auditing. In our logs, if the LLM extraction fails or produces weird results, having the structured JSON lets us pinpoint if the issue was in parsing (e.g., maybe MinerU missed a section so the LLM never saw it).

Maintenance: We plan to swap out or upgrade this component if better tech emerges (like if Anthropic releases a native PDF parsing function, or if we incorporate Adobe's PDF Extract API). Because it's encapsulated, we can do so without affecting the rest of the pipeline. The key is to ensure the output matches the schema the LLM expects. This component's documentation (Tier 3) includes specifics of the JSON fields and example outputs, which should be updated if any parser changes.

Extraction Component (LLM Extraction Service)

Purpose & Overview: The Extraction component is responsible for turning unstructured text from FDDs into structured data fields using AI. It encapsulates all prompt engineering, LLM calls, and output validation. The component ensures that for each FDD document, we produce a JSON with the required fields (like `franchise_name`, `initial_fee`, `royalty_fee`, `renewal_term`, etc. – as defined in our Pydantic model for FDD data). Its purpose is to automate what an analyst might do by reading the document: identify and record key pieces of information.

Current Status: **Stable with improvements ongoing.** The extraction service using Claude is operational – it can successfully extract data for the majority of documents when the text is clear. The error rate (cases where model output needs retries) has been reduced by iterative prompt tuning. We recently implemented the OpenAI fallback which improved overall success to nearly 100% of filings processed (some previously stuck cases were resolved by GPT-4's stricter formatting). Ongoing work includes refining prompts for edge cases (like when an FDD has unusual terms or formats) and expanding the schema as needed.

Prompt Design & Schema: This component maintains the **prompt templates** and the **Pydantic schema** that define the extraction. The Pydantic model `FddExtract` (for example) lists fields such as `franchise_name: str`, `initial_franchise_fee: Optional[Money]`, `royalty_fee: Optional[str]`, `item_19_financial_performance: Optional[str]`, etc., covering all items we want to capture. Each field has a description used in the prompt (via Pydantic's `Field(..., description="...")`). We include these descriptions in the prompt so the model knows exactly what each field means and what format is expected. For example, for a money field, the description might be "Initial Franchise Fee (in USD, format as number or range)". We also enumerate expected answers where possible (e.g., if a question is yes/no, we use an Enum with values "Yes" or "No" plus an "Other/Not disclosed" option) to guide the model. The prompt typically says something like: *"Extract the following information from the FDD. Respond in JSON only, with the following keys: …"* and then either lists the keys with

explanations or even embeds a mini JSON schema. According to Anthropic's best practices, we **precisely define the output format** so Claude knows to stick to JSON without extra text. In fact, we often prepend something like: *"Important: output JSON only, no prose."*.

The **Instructor** library helps here by allowing us to pass a `response_model=FddExtract` when calling the LLM. If using OpenAI, Instructor leverages function calling under the hood; with Claude, it uses a validation approach (it will try to parse Claude's response into `FddExtract`, and if it fails, it can automatically trigger a reprompt with error info). This abstraction significantly simplified our code: instead of manually writing retry loops, we declare the model and call `client.chat.completions.create(...)` once – Instructor takes care of re-asks and returns either a validated Pydantic object or raises an error after max retries.

Retry & Fallback Logic: The extraction service is built to be resilient. After an initial attempt with Claude, if `model_validate_json` fails (Pydantic throws a `ValidationError`), our code catches it. We then log the validation error (including which field failed). We then modify the prompt slightly – often we'll add a system message like: *"The previous output had an error: it was not valid JSON or missing field X. Please output all fields and valid JSON."* – and call Claude again with the same context. This usually corrects the output on the second try. If it doesn't, we maybe try one more time. All these attempts are recorded. After 3 tries, we switch to GPT-4. With GPT-4, we construct a function call request with the model's schema (thanks to Pydantic's `model_json_schema()` which we convert to the OpenAI function parameter format). GPT-4 will then return a JSON in `message.function_call.arguments`, which we parse and validate. In essentially all cases, GPT-4 gets it right first try in terms of JSON structure (though it occasionally might hallucinate content if the prompt wasn't clear – but since it's reading the same input text, that's rare and we check critical fields). This tiered strategy ensures we very rarely have to mark a document as "could not extract." And even if we do, since we log it, we can follow up manually.

Additionally, the extraction includes some **post-processing validation**: For example, if a field should be a percentage but the model gave "5" (ambiguous), we might interpret that as "5%" or flag it. Where possible, we have the model include units in output (like "5%" or "$50,000") to avoid ambiguity. We use Pydantic validators to normalize these (stripping "$" and converting to numbers where applicable). If these validators fail, that surfaces as a validation error and triggers a retry with a prompt asking for clarification (e.g., "please include currency symbols" if missing). All these aspects are logged in Supabase for monitoring extraction quality.

Integration: The extraction service gets invoked via Prefect after parsing. If running as a FastAPI service (`extractor_api`), it exposes an endpoint like `POST /extract` where the request body includes either the text or a reference to the text. The response is the JSON or an error. Internally it uses the same logic described. For direct Prefect integration, we might call a task `run_extraction(text) -> dict`. The component is self-contained in that it knows how to talk to the Anthropic and OpenAI APIs (API keys are loaded from config).

Performance & Cost: LLM calls are the most expensive step. To manage this, we consider the prompt size. Some FDDs are 200+ pages, which is too long for a single prompt (Claude's context is large, but we shouldn't send everything if not needed). We have implemented a simple heuristic to trim the input: for instance, we might extract only the sections of the text that are relevant. Because FDDs have numbered items (Item 1, Item 5, Item 19, etc.), we can split the text by these headers and, if necessary, send separate prompts for very large sections (though currently we usually can fit one full FDD in Claude 100k context, but

GPT-4 8k would require splitting, for example). The component checks token length and if it exceeds a threshold, it either uses Claude's 100k model or splits the input. In our configuration, we default to Claude's 100k context model to avoid splitting and losing context, even though that model may be slightly slower. We also implemented caching: once an FDD's data is extracted and stored, we don't re-run the LLM on it unless a change in prompt/schema occurs. This is enforced by the DB: we have a unique constraint on (franchise, year) and we won't process an entry that already has extraction data. To update an extraction (say we improved the prompt and want to re-run on past documents), we have a separate management command to do that explicitly.

Security & Privacy: All interactions with external LLMs are done over encrypted connections. We do not send any personally identifiable information – FDDs are public documents about businesses, but we still treat the data carefully. The Anthropic and OpenAI API keys are stored in `.env` and loaded via config (not hard-coded). We also have usage limits in place: environment configs include rate limit settings or monthly budget caps to ensure we don't overspend on API calls unexpectedly.

Testing: We test this component by using a set of sample FDD texts and expected outputs. We have unit tests that simulate the LLM by mocking the API response (for instance, feeding a known JSON string as if Claude returned it) and checking that our parsing and validation logic accepts it. We also test the retry logic by simulating a bad output first. For integration testing (with a real model), we use a smaller model or shorter prompt to avoid cost, just to ensure the pipeline flows. Manual spot checks are regularly done on actual outputs to verify correctness (e.g., did it pick the right numbers from Item 7?).

Database & Storage Component

Purpose & Overview: The Database & Storage component handles persistence of both the extracted structured data and the raw/unstructured data (PDFs, JSONs). It ensures that all information is safely stored and easily queryable. This component includes the Supabase Postgres database and Supabase Storage, as well as the data access layer in our code (SQLAlchemy models and supabase-py usage).

Schema Design: In the Postgres database, we have a few key tables: - `filings` – core table, one record per FDD document (per franchise per year per version). Contains metadata like franchise name, year, state, date_published, a reference to the PDF file, and status flags (processed, duplicate, etc.). It may also have some high-level extracted fields for quick reference (like initial_fee, etc., denormalized for filtering). - `fdd_data` – table with detailed extracted data (one-to-one with filings, but stored in a JSONB column or as separate columns). We chose to store the entire Pydantic output as a JSONB column here for flexibility, since the schema might evolve. This way, new fields can be added without altering the table, though important fields can also be columns if we need to index or filter by them. - `errors` – logs of extraction errors or other pipeline issues. Each entry links to a filing and has details like stage (`parse` vs `extract`), error message, timestamp. - possibly lookup tables for franchise (franchise name to an ID, so that if the exact name string varies we still group them – this is an enhancement we plan: a cleaned franchise identifier).

Supabase being Postgres allows us to enforce constraints – e.g., we have a uniqueness constraint on (`franchise_name_normalized`, `year`, `version_number`). This helps the Deduplication component: if a duplicate gets through and we try to insert a record with the same franchise and year, the DB will prevent it. We normalize franchise names (strip punctuation, casefold) to avoid minor naming differences causing duplicates.

Supabase Storage: We created a bucket (e.g., `fdd_pdfs`) in Supabase Storage where we store the PDF files. The scraper, after downloading a PDF, uploads it via the supabase-py client: `supabase.storage.from_("fdd_pdfs").upload(path_in_bucket, local_file_path)`. We use the franchise/year in the file path for organization (e.g., `acme_co/2024_FDD.pdf`). Supabase Storage provides us a URL for each file, which we save in the `filings` table. Similarly, we might have a bucket for parsed JSONs (though those can be regenerated, so storing them is optional). The storage has role-based access; our service is using an admin service role key (kept in .env) to read/write objects.

Data Access Layer: In our code, we use **SQLAlchemy** to define ORM models corresponding to these tables (for ease of complex querying and to have Python classes map to tables). We also use Supabase's own query interface for simpler operations or when leveraging Supabase specific features (like if we used Supabase's realtime or something – not heavily used yet in this pipeline). Supabase-py's query builder is a bit limited compared to full SQL, so for any complex joins or fuzzy search, we either use SQLAlchemy or direct SQL queries through the psycopg2 connection that Supabase-py wraps. One dev noted they prefer writing their own SQL or using SQLAlchemy rather than the supabase-py query DSL, and we follow that principle for non-trivial queries. For example, to find potential duplicates we might do a SQL query with a similarity function (if we had pg_trgm extension for text similarity).

We also added Pydantic models for data coming out of the DB, so that we have type-checked data moving in our application. The JSONB field for extracted data can directly be parsed by our `FddExtract` Pydantic model if needed, which is convenient.

Integration & Transactions: The pipeline writes to the database at defined points. We encapsulated DB writes in the Prefect flow after each major task. For instance, after scraping, we insert new `filings` records with status "downloaded". After parsing and extracting, we update those records with status "extracted" and insert into `fdd_data`. Using transactions ensures that if a pipeline run stops unexpectedly, we don't end up with half-inserted data. We mark duplicates with a flag in `filings` and a pointer to the original filing (self-referential relationship or just note the original's ID). This way no data is lost; duplicates are just linked.

Backup & Retention: Supabase automatically retains data (and we can configure backups), but we also periodically export critical tables to CSV for backup (just an extra precaution since this is important regulatory information). The PDF storage is the bulk of data; we monitor the storage usage and will archive older files if needed (but since these are not personal data, we are fine keeping them indefinitely for historical comparison).

Access and Security: The database and storage keys are in .env. In a production scenario, we would use more secure secret management, but for local and development, a .env loaded via Pydantic BaseSettings is acceptable. Supabase provides row level security (RLS) which we could configure if we had user-level access, but here our service uses the service role with full access and we haven't enabled RLS on these tables. That is fine since our service is the only client. If we were to expose some data publicly or to another app, we might create a restricted Supabase role or use the Supabase JS client with RLS in the future. For now, our integration is straightforward: direct service-key usage.

Performance: Postgres is more than capable of handling the volume of data (even if we ingest a thousand FDDs, that's a thousand rows, which is trivial). Query performance is good; we have indexes on franchise name, year, etc., to quickly find data. We need to be mindful if we do any fuzzy matching queries (like using

`ILIKE '%Acme%'` or trigram similarity) – those can be slower, but the volume is low. We might add a full-text index on the JSON data if we later want to search across all extracted text (that could be powerful – e.g., find all FDDs that mention "territory"). Supabase being hosted means our data is on the cloud; our pipeline running locally communicates over the internet to it. This is fine as long as internet is reliable; if we needed completely local, we could run a local Postgres and MinIO (S3) for storage. In fact, for an offline environment, we can swap Supabase with a local stack by changing configuration (because supabase-py can point to a self-hosted Supabase or a local Postgres with some adjustments).

Maintenance: The DB schema might evolve if we decide to store more info. Because we use Alembic (if set up) or manual migration scripts, we keep track of schema changes. Supabase also provides a UI to modify tables, but we prefer migrating via code for reproducibility.

This component's interplay with others is straightforward: other components *ask* this component for data or give it data to store. For example, deduplication will query `filings` table for existing records (maybe using fuzzy functions if needed), the orchestrator will insert/update records, and any reporting tool would read from the DB. The storage is accessed mainly by the scraper (upload) and possibly by the parser (if we stored JSON outputs, though those could also be regenerated on the fly from PDF + parser if needed).

Orchestrator & API Component (Workflow Management)

Purpose & Overview: The Orchestrator component ties everything together. It is essentially the Prefect flows and the optional FastAPI orchestration layer. Its responsibility is to coordinate the timing of operations and provide a user interface (CLI or API) to trigger and monitor the pipeline. It doesn't do heavy data processing itself but invokes tasks in other components that do.

Prefect Workflows: We have defined Prefect flows to represent the main workflows: - `weekly_scrape_and_process_flow`: The primary flow that the scheduler runs weekly. It might internally call sub-flows or tasks: `scrape_task` → for each state, and then for each result, `process_file_task`. - `process_file_task`: A sub-flow or task that takes a filing record (or PDF path) through parse and extract and store. - We might also have utility flows like `reprocess_failed_flow(filing_id)` for manual re-runs or a `deduplicate_audit_flow` to scan and mark duplicates post-hoc.

Prefect v2 (Orion) allows us to schedule flows easily and to observe runs in the Orion UI. We run a local Prefect Orion server that persists state (so we can see historical runs, failures, etc.). The Orchestrator uses **Prefect's task dependency** management to ensure sequence (scrape must finish before extraction starts, etc.), but we also set it up so that tasks can run concurrently where appropriate (multiple parse tasks at once, multiple extraction tasks, etc., with bounds).

One advantage of Prefect is its **caching and result storage**. We use result caching for the parse task: if a parse task for a given PDF has been run and the PDF hasn't changed, it can skip redoing it (Prefect can use a hash of the input file as a cache key – we configured something along those lines). We also considered caching extraction results similarly, but since we already guard via the DB, it's less crucial.

FastAPI Integration: We have a **FastAPI app** that serves as a control plane and potentially as a data access API. It has endpoints such as: - `GET /health`: to check system status (could query DB for any failed runs,

etc.) - `POST /run` or `/scrape`: to manually trigger the main flow (this could enqueue a Prefect flow run asynchronously). - `GET /filings{?query}`: to fetch data from the DB (for example, all filings for a given franchise, or search by state or year). This allows easy retrieval of results for internal users without directly querying the database.

This API is not meant for public use but as an internal tool. We run it with Uvicorn, and it's integrated with the Prefect logic by either embedding the Prefect calls or by using Prefect's API. One pattern we used is the [Background Tasks with FastAPI and Prefect pattern][prefect-fastapi], where the API endpoint adds a background task to start a Prefect flow and immediately returns a flow run ID for monitoring. Clients (or the UI) can then poll another endpoint for the status or results.

Configuration Management: The Orchestrator is where the configuration of the whole pipeline is consolidated (in `config.py`). It loads env variables (via Pydantic BaseSettings) and constructs config objects for subcomponents, grouping settings logically (e.g., scraping settings group, AI settings group, DB settings group). This makes it easy to manage different environments – for example, in a dev environment, you might set `SCRAPER_HEADLESS=False` to see the browser, or use a sandbox Anthropic key. We strictly avoid calling `os.getenv` outside config initialization, meaning all components receive config via dependency injection (passed in or imported as a config singleton). This consistent strategy helps avoid subtle differences between dev/prod and eases debugging (all settings are printed at startup except secrets).

Integration & Communication: The Orchestrator primarily communicates with components via function calls or API calls. For instance, after scraping, it might simply call the next Python function for parsing if running in one process, or it might make an HTTP call to a parsing service if that's running separately. We abstracted these calls behind interfaces. For example, there's an `ExtractorClient` class. If configured to local mode, it calls the Python function directly; if configured to remote mode, it calls the HTTP endpoint. This allows flexibility in deployment (monolith vs microservices). In our current local dev, we often run it in monolith mode (one process does all via direct calls, which is simpler for local debugging). In a deployed scenario with Docker, we might split them and use HTTP between.

Error Propagation: Prefect will catch exceptions in tasks and mark task/flow state accordingly. Our FastAPI layer, if it triggers a flow, will catch if the flow fails to start or if it returns a failed state, and translate that into an HTTP error code or a message. We ensure that any exception in deeper components gets surfaced through logs and statuses, so that an operator checking the Prefect UI or our API knows something went wrong. For example, if extraction fails for a file, Prefect will mark that task as failed but the flow might continue with others; at the end the flow might be marked "completed with some failures". We aggregate such info in the final flow result (maybe compiling a report of failures). The `/health` endpoint can then report "X filings processed, Y succeeded, Z failed".

Testing & DevOps: For this component, the focus is on ensuring the whole pipeline runs smoothly. We test flows by running them on a subset of data (e.g., a test mode that only scrapes one state or processes one known PDF). We use Prefect's ability to simulate runs (there's no strict simulation mode, but we can run flows synchronously in dev). We also wrote smoke tests that call our API endpoints locally to see if they trigger the right actions (these aren't unit tests but integration tests requiring the system up).

From an operations perspective, the Orchestrator is what an on-call person will often interact with: e.g., use an API or CLI command to re-run a flow, or use the Prefect UI to retry a failed task. We documented

common runbook items (like “if a flow fails due to network issue, just retry via UI”; “if a particular file keeps failing extraction, mark it skipped and investigate separately”).

Scaling & Future Work: As volume grows, we might move the orchestrator to run on a server or use Prefect Cloud. In that case, the Prefect flow definitions remain mostly the same, but we’d deploy agents that run tasks (for example, one agent could be dedicated to running extraction tasks possibly in parallel on multiple processes). Prefect allows distributing work across multiple workers which could speed up things if we have many docs at once. Another future improvement is to integrate notifications: e.g., integrate with PagerDuty or simply send an email if a flow run fails, so that the team knows without checking logs. This is part of the support process (discussed in *Handoff & Continuity*).

In summary, the Orchestrator component is the command center of the pipeline – it ensures each piece plays its part at the right time and provides mechanisms to control and monitor the whole process. It is built with Prefect flows for robust orchestration, and FastAPI to expose control and data access interfaces in a modular fashion.

Feature Documentation (Tier 3)

In this section, we dive into specific features and technical implementations in the pipeline, providing a closer look at how certain challenges are addressed.

Weekly Scheduled Scraping & Historical Versioning

Feature Overview: This feature ensures that the pipeline automatically checks for and ingests new FDD filings on a weekly basis, while retaining all historical documents for each franchise. Rather than a one-time import, this is a continuous process. The scheduling is handled by Prefect (or an external scheduler like cron invoking the flow), set to run e.g. every Monday at 3am. Historical versioning means that if a franchise updates their FDD yearly (or multiple times a year), all those versions are stored and distinguishable.

Architecture & Implementation: Using Prefect’s scheduling, we’ve set up a deployment of the `weekly_scrape_and_process_flow` with a Cron schedule. The Prefect Orion server (if self-hosted) handles triggering. On trigger, the flow executes the scrape as described in the Scraper component. It filters out already-seen files by comparing to what’s in the database (this is done by checking a combination of franchise name and year mostly, because often the new year’s documents are what we look for; but we also check exact filenames when available). Newly found items proceed to processing.

To implement historical tracking, the database design is key. Each `filings` record has fields `year` and `issue_date`. For initial version, `issue_date` might be the date on the cover of the FDD if we parse it, or the date the regulator posted it (which we have from scraping). If the same franchise has multiple filings with different issue_dates or years, we keep them all. In queries, one can retrieve the list of all filings for “Acme Corp” sorted by year to see the progression. This approach is similar to Wefranch’s concept where they list historic FDDs for comparison.

State-specific Considerations: Some states might only keep the latest document online. For those, if we miss a document, it could disappear. To mitigate this, we attempt to scrape even if something changed (we don’t solely rely on “new since last time” because if last run failed, a new run should try again). We

effectively re-download what's currently available and trust our deduplication to not double-store. This way, if a site replaced an old PDF with a new one under the same name, we would still capture the new content (though dedup logic that uses just name would need to consider content differences – which we do via fuzzy matching on text if needed). For states that list multiple past years (some do provide archives), our scraper goes through all available, which initially populated historical backlogs, and subsequent runs focus on new additions.

Cron and Timezones: We scheduled weekly scraping at a time when state sites are least likely to be updated or have downtime (early Monday). Prefect's scheduler takes care of running on time. The pipeline records the run time so we know when last executed. If a run is missed (e.g., server down), we can run it manually later – it will still pick up new stuff as described.

Testing & Simulation: To test the scheduler and flow, we manually triggered flows at non-weekly intervals in dev (e.g., daily) and also simulated missed runs by turning off scheduling and then running after two weeks to ensure it still catches up. We also tested historical capture by running the scraper on known archives and verifying multiple entries go to DB.

Monitoring: The pipeline uses logging to output how many filings were found and processed each run. Over time, this gives a trend. If a week has zero new filings (which could be normal seasonal behavior), it logs that. We have an alert if, say, three consecutive scheduled runs find nothing, to check if perhaps the scraper is broken or states changed their pages (in practice, there's almost always some new or updated filings weekly across all states).

Version Differencing (Future): While we store historical data, a future feature is to automatically diff an updated FDD against the previous version to highlight changes (like fee increases, etc.). Though not implemented yet, the data model allows it: since we have all versions, we could have a job (maybe another flow) that compares the JSON outputs of successive years and stores differences or annotations. This is beyond current scope, but it's a direction aligned with having historical data.

Deduplication of Filings (Fuzzy Matching)

Feature Overview: Duplicate filings can occur when, for instance, the same FDD is filed in multiple states. Our deduplication feature identifies these cases to prevent redundant processing and storage. Unlike a simple file hash check, this uses fuzzy matching on metadata and content, because the PDFs might not be binary-identical across states (some states add covers or stamps). The logic concentrates on matching franchise name, year, and content similarity.

Deduplication Logic: When a new filing is discovered, we create a normalized key: `norm_name = franchise_name.lower().replace("&","and")` (and other normalizations), and take the year (usually from the document title or the context of where it was found). We query the database for any existing filing with the same normalized name and year. If none, clearly not a duplicate, proceed. If one or more exist, we then compare the content. We might already have the text content from the MinerU parse of the existing one (or we can do a quick PDF text extraction via PyMuPDF for comparison). We then compute a similarity score. We use **RapidFuzz** (a faster successor to FuzzyWuzzy) to compute something like `token_set_ratio` between the two documents' texts. If the score is above a threshold (say 95%), we consider them duplicates. We chose a high threshold because different year FDDs could be quite similar (e.g., 2023 vs 2022 might be 80% similar but are not duplicates, they are different versions). But the same

document filed in TX and NY will be virtually 100% identical except perhaps a cover page. So 95% catches that without conflating year-to-year updates.

If a duplicate is found: - We mark the new one's record as `duplicate_of` the original's ID. - We do not run parsing or extraction on it. Instead, we might simply copy the extracted data reference to this new record, or just not store any data for it besides pointing to original. (Currently, we mark and skip, not copying data, since one could always query via the original franchise entry. But for completeness we might fill it in.) - In logs, we note "Skipped duplicate: Franchise X 2024 filing in State Y is duplicate of State Z's filing".

There is also a nuance: If the same franchise files two different documents in the same year (maybe an initial and an amended FDD), our dedup logic could misidentify them as duplicates if we relied only on name/year. That's why content check is important. In such a case, content would differ significantly, so they won't be flagged as duplicates. They'll both be processed and stored, and they would have maybe different issue_dates. We consider those separate versions, not duplicates.

Edge Cases & Tuning: We tuned the threshold by testing on a small set of documents. We also considered file size and page count as quick checks: if two PDFs have exactly the same page count and a similar size (within a few kilobytes), that's a strong hint of duplicate. If page counts differ, likely not duplicate. So our code does: if name/year match and abs(pages_a - pages_b) <= 1 and fuzzy_ratio > 95% then duplicate. If fuzzy ratio is borderline (say 90-95), we might log it for manual review but still process it assuming it could be a revision.

We avoid false positives because that could cause missing data (worst-case if we wrongly skip a new version assuming it was duplicate). That's why we lean towards requiring very high similarity.

Implementation: In the Prefect flow, right after scraping and before processing, we run a dedupe check on the list of new filings. This can mark some filings as duplicates. We then only map the non-duplicates into the processing sub-flow. The duplicates are inserted into DB with duplicate flag and `duplicate_of` set, but not sent through parse/extract tasks. This ensures we save compute.

We also implemented a CLI command `find_duplicates` to retroactively identify duplicates in the DB. This can be run periodically to catch any duplicates that might have slipped through initial check (perhaps if they were ingested in separate runs and an earlier run didn't have the later file to compare). This scans the DB by franchise and year grouping and flags duplicates similarly, possibly updating records. It uses the same RapidFuzz logic on stored content.

Dependencies: RapidFuzz (or FuzzyWuzzy) is our tool of choice for string similarity. It's fast enough for our scale (comparing a couple hundred documents quickly). In the future, for very large text, we might use hashing (like simhash or MinHash) to approximate similarity more efficiently. But currently, performance is fine.

Testing: We tested with known duplicates: e.g., if we manually input the same PDF from two states, does it catch it? Yes. We also tested near-duplicates (like one doc with a couple pages difference) to ensure it doesn't false-positive. The tests for the dedupe function use sample text strings and verify the output (using a small sample from StackOverflow as inspiration for fuzzy matching logic). We ensure the function returns correct booleans and indexes.

Outcome: This feature reduces redundant processing significantly. For example, if a franchise registers in 10 states, without deduplication we'd process the same content 10 times and store 10 copies of data. Now, we process once and note 9 duplicates. It saves API costs and storage, and when presenting data we won't double-count franchises. Users can see that a particular state's filing is duplicate of another's (which might indicate a multi-state region, etc., though that detail may not be very relevant to end users).

LLM Prompt Strategy & Schema Enforcement

Feature Overview: This covers how we craft prompts for the LLM to reliably extract the needed data, and how we enforce the output schema. It's a critical feature for ensuring data quality from AI. The approach involves carefully engineered prompts, usage of the Instructor library for schema validation, and iterative refinement of prompts based on error logs.

Prompt Components: The extraction prompt to Claude (or GPT-4) typically has the following structure: - **System Message (for OpenAI)** or the first user message contains instructions about format: e.g., *"You are an AI reading a franchise disclosure document. Extract the following information and output as JSON."* For Claude, we include similar instruction at the top of the single prompt since it follows a single conversation turn model by default. - **Schema or Field List:** We then either list each field with a description, or provide a mini JSON template. Initially, we used a descriptive list like: - *Franchise Name: The legal name of the franchise as stated in the document. - Initial Franchise Fee: The dollar amount of the initial fee...* etc. We found that Claude sometimes would return a nicely formatted JSON but occasionally include extra commentary or miss a field if it didn't find it. To combat that, we switched to giving an **example output structure**. For instance, after the descriptions, we show:

```
{
  "franchise_name": "",
  "initial_franchise_fee": "",
  "royalty_fee": "",
  "item_19_financial_performance": ""
}
```

and say "Fill in the JSON with the values or empty string if not found." This technique of pre-filling or showing the format often helps the model focus. - **Document Content:** We then include the text of the FDD (or relevant parts). To avoid extremely long prompts, we sometimes pre-select sections like Item 5 (fees) and Item 6 (other fees) if we specifically want those fields. However, to be safe, our default prompt passes the whole text and relies on the model to find the info. We leverage that FDDs have a standard structure (Items 1-23) in the text.

Function-aware Prompting: We treat the prompt as if defining a function spec for the model, especially when using GPT-4. With Claude not having native function calling, our method is to provide the JSON schema in text. We derived the JSON Schema from Pydantic (using `model_json_schema()`) and inserted it into the prompt: e.g., *"The output must be a JSON object conforming to this schema: {schema here}."* This essentially simulates function calling by giving Claude the exact keys and types expected. According to OpenAI and Anthropic guidance, providing a JSON schema can significantly improve structured outputs. Indeed, our tests showed Claude adhered more closely when we included the schema explicitly.

Instructor Library Use: By using `instructor.patch(OpenAI())` or similar for Anthropic (Instructor has integration for Anthropic as well in newer versions), we allow the library to handle a lot of the heavy lifting. With OpenAI, we saw in the Pydantic example that you can just pass `response_model=YourModel` and get back a validated object. For Claude, under the hood, Instructor will likely call the model and then attempt to parse, and if it fails, it might re-prompt. If it doesn't automatically re-ask, our code does so manually as described. The key benefit is we use Pydantic's error messages to adjust prompts. For example, if Pydantic says "value is not a valid integer" for a field, we know the model gave something unparsable for an int. We then add to the prompt: *"Ensure all integer fields are digits (no words)."* These targeted instructions, added upon retries, gradually improved the model's adherence.

Field-by-Field Strategies: Some fields required special strategies: - For currency fields, sometimes the model would output "$50,000" which is fine as a string, but we might want numeric. Our schema currently allows string for money to keep things simple (since ranges like "$50,000 - $100,000" are best as string). But we ensure `$` is present. So for money fields, we explicitly say in the prompt "include the dollar sign and commas as in the document". - For yes/no fields, we used `Literal["Yes","No","N/A"]` in Pydantic which means if the model outputs anything else, it fails validation. Initially model might say "No (there is no such clause)" which is not exactly "No". We tweaked the prompt to say *"Answer 'Yes' or 'No' explicitly."*. Also, by using enumerations in Pydantic, we guide the model's output. - For lists (like a list of states or a list of required purchases in the contract), we instruct to output as JSON arrays. If the model gave them as a single string or a numbered list in text, Pydantic would error (type mismatch expecting list). So our prompt example includes `"territories": ["CA","NY",...]` for example.

Chaining and Multi-step Reasoning: We considered using a "chain-of-thought" to have the model reason then output. But since we want a clean JSON, chain-of-thought should not be in the final answer. One approach is to allow the model to produce thoughts in a hidden field, which is something Instructor suggests (the Maybe pattern or a chain-of-thought field) [1]. We experimented by giving Claude a hidden scratchpad: like adding a field `_analysis` in the schema where it can dump its reasoning. This way, it could use it to think through, and we could ignore that field later. However, in practice, we found Claude often didn't need that for straightforward extraction, and we didn't want to risk it outputting the reasoning in a place we didn't expect. So we opted for direct prompting. But we kept the idea in mind: the schema could include an optional field for reasoning that we would strip out before storing. This matches a tip to include a chain-of-thought field in the schema for complex logic [1].

Logging & Iteration: The first few times we ran extraction on various FDDs, we encountered output issues (like the model sometimes combining multiple fields, or giving a narrative answer instead of JSON due to forgetting format). We iteratively improved prompts by analyzing those failures. We maintain a prompt version (comment in the prompt template file) and note changes in our docs. We treat prompt tweaks like code changes – tracking them in version control with commit messages explaining why (e.g., "Added note to output percentage as number to fix validation error on royalty fee"). This is in line with treating prompts with version control discipline. In fact, we have a `PROMPTS.md` log where each prompt version is listed with changes, providing a history (for example, "v1.1 - added explicit JSON example, improved field descriptions for clarity").

Claude vs OpenAI Prompt Differences: We maintain slightly different prompt phrasing for Claude and OpenAI to play to their styles. Claude tends to follow format if you say "Output JSON only." but sometimes it still wraps in `json` markdown. We strip those in post-processing if present (just regex out triple backticks). We also discovered an Anthropic tip: including something like `\n\n{` at the end of the user prompt can

nudge Claude to start directly with a JSON object (this is a known trick to increase JSON output consistency, effectively "prefilling" the assistant response with a `{`). We implemented this: after the prompt text, we put something like: *"Now output the JSON:\n`json\n{"* as part of user message. Claude then usually continues writing JSON. We then remove the leading`json in post. This improved reliability.

For GPT-4, since we use function calling, the prompt is a bit different (we don't need to say "in JSON" because the function spec ensures it). We simply instruct it to extract and let the function mechanism handle format. That's another reason GPT-4 rarely fails schema – it's forced.

Auto-Retry Strategy: As described, the Instructor library plus our own logic implements auto-retry. We set a counter and loop until success or max tries. After each failure, besides logging, we adjust the prompt. If the failure is generic (like not JSON at all), we just reiterate "Remember: JSON only, no explanations." If it's specific (missing field X), we explicitly tell it to include that. This targeted feedback is possible because Pydantic errors tell us which field is missing or wrong type. It's akin to how a human would say "Oops, you left out field Y, add it." This often solves it on the next attempt. We saw success in a similar pattern in literature, where structured output frameworks "re-ask" with error info.

If after 3 tries with Claude it doesn't work (very rare by that point), we log "Claude failed for filing X, switching to OpenAI." Then the GPT-4 path kicks in, which nearly always yields valid JSON. We then log that we had to use fallback for that file. We keep stats on how often fallback is needed as a quality metric for Claude prompt performance.

Outcome: With this rigorous prompt and schema strategy, our pipeline's extraction accuracy and consistency are very high. When we spot mistakes (for example, maybe the model mis-identified a fee because the FDD used unusual wording), we either adjust the schema (add a field if we missed something) or the prompt (add to instructions like "If an item is not stated, use `null`" or such). We also incorporate best practices from Anthropic's and OpenAI's documentation regularly – for example, Anthropic's docs suggest being clear, using examples, and even using role instructions effectively. We followed these by making the instructions explicit and providing the JSON snippet example.

We also ensure **no prompt secrets or proprietary info** are in the prompt (just in case, though these FDDs are public documents anyway). All prompts are stored in the repository under `prompts/` and thus versioned. This also means if something goes wrong or if outputs drift over time (maybe a model update changes behavior), we have a clear record of what prompt was used and can roll back or adjust with context. This is essentially "prompt management", analogous to code management, an emerging best practice.

Claude Code & AI Development Best Practices

Feature Overview: This part of the project isn't a user-facing feature, but rather an internal practice: using Claude Code and other AI-assisted coding effectively. It involves how we structure our repository and prompts for AI agents (Claude) to be productive in helping develop and maintain the pipeline. This includes prompt versioning strategies, documentation for the AI (as covered by the 3-tier docs), and ensuring continuity between AI-assisted sessions.

Claude Code CLI Usage: Our team uses Claude Code (Anthropic's agentic coding CLI) during development. This tool automatically loads context like `CLAUDE.md` and relevant `CONTEXT.md` files for components when we ask Claude to perform coding tasks. We've fully embraced this by writing comprehensive CONTEXT docs (Tier 2 and Tier 3 as above) so that Claude always has the info it needs. For example, when using Claude Code in the `scraper/` directory, we ensure there's a `CONTEXT.md` summarizing the scraper component architecture and guidelines (like "don't overload target servers", etc.). Claude Code recognizes those and includes them in its prompt to itself.

Folder & Prompt Organization: We maintain a `prompts/` directory with all LLM prompts (for extraction, for any other tasks if they arise). Each prompt file is named clearly (e.g., `extract_fdd_prompt_v1.2.txt`). The version is in the filename to allow quick switching or comparison. We found that as prompts evolved, keeping old versions commented in one file got messy, so instead we archive them by version. LaunchDarkly's guide on prompt management suggests using clear labeling and documentation for prompt changes, which we follow by having a changelog in our docs as mentioned.

We also ensure prompt files are small and modular. If in future we add another LLM task (like summarizing Item 19 financial performance narratives), we'll make a separate prompt file for that, rather than one giant prompt doing everything. This adheres to the principle of modular prompts, which keeps them easier to maintain and test.

Version Control and Testing of Prompts: Every prompt change is committed to git. We often test a new prompt version on a handful of documents to see the effect before rolling it out broadly. Because LLM outputs can be nondeterministic, we incorporate manual review in prompt updates – basically, verify that under the new prompt, known tricky cases now pass validation, and that nothing new breaks. We also keep a suite of "prompt test cases" – short scripts that load a prompt and call the model (possibly on a smaller example or with `max_tokens=0` if just testing function calling structure, etc.) to validate basic format adherence. While not as rigorous as unit tests, these give quick feedback (like if we accidentally left an extra curly brace in the prompt example, etc., we'd catch it).

Continuous Integration for Prompts: In our CI pipeline, we included a step that runs a few sample extraction tasks with a dry-run flag (not saving to DB) just to ensure the models return something parseable. This way, if a prompt change makes things significantly worse, we'll catch it before deploying. This is a nascent area, but it's akin to unit tests for prompt quality.

Escalation Path for AI Issues: If our AI outputs are consistently wrong for a certain field and prompt tweaks aren't helping, we have an escalation procedure: escalate in this context means involve a human or a more powerful analysis. For example, if an LLM can't parse a particularly convoluted Item 19, we mark that field as "needs manual review" in the output and log it. Then a human analyst can later fill it in. We document such cases in the *handoff* section so support staff or future developers know these limitations and what to do (like contact the legal team for those edge cases, etc.). Thankfully, such cases are few.

Team Collaboration via AI: Claude Code allows multiple developers to benefit from improved prompts and docs. Everyone on the team follows the practice of updating documentation when making code changes, which keeps AI assistance relevant. For instance, if someone changes the DB schema, they update `project-structure.md` (Tier 1) and the relevant context file. This way, the next time someone asks Claude "How is the data stored?", it won't hallucinate or rely on outdated info – it has the updated structure

doc. We treat our documentation as part of the development process, not an afterthought, especially because an AI is consuming it to help us.

Best Practices Recap: - *Keep instructions clear and prioritized:* We learned to put the most important instruction at the end of Claude prompts because Claude tends to focus on the last user message instruction (hence we repeat "output JSON only" at the end). - *Use examples and self-descriptive fields:* Pydantic's field descriptions help the model understand without lengthy prose. We write descriptions like "Royalty Fee (percentage of revenue, e.g., 5%)" – so the model sees the format expected. - *Optional fields and defaults:* We define optional fields in the schema with None default if they may not exist, so the model knows it can leave them null. If a field is not in the document, we instruct the model to output null or an empty string – and our Pydantic model accepts that. This avoids the model trying to hallucinate a value for something absent. - *Enumerations:* As mentioned, for any field with limited set values, we use Enum or Literal in Pydantic, which automatically constrains outputs. The model usually picks one of the provided values if the description says "choose one of: Yes or No". - *Tool use awareness:* Claude Code can also use tools. We did not heavily use the tool-calling (MCP) in this project, but it's good to know that if we had a complex transformation, we could register a "function" for Claude. For example, we could have a function for "calculate similarity" and Claude might call it if it were reasoning about deduplication. We haven't needed that, but we keep the possibility in mind for future enhancements (like automatically calling a secondary model or using Python code). - *Prompt versioning:* We tag prompt versions and can roll back if a change makes things worse. Having version history allows A/B testing as well. LaunchDarkly's guide suggests practices like A/B testing prompt changes and monitoring performance. We haven't done formal A/B in production (since it's a small-scale internal pipeline), but if this were a user-facing feature, we might indeed test prompt variants to see which yields better extraction accuracy.

By adhering to these practices, we ensure that the AI components remain **reliable, understandable, and maintainable**. The combination of strong schema validation and disciplined prompt management means we treat the AI's behavior as part of the software – something we specify, test, and version, rather than a magical black box. This reduces surprises and makes the pipeline more robust.

Environment & Configuration Management

Overview: Managing configurations (like API keys, database URLs, feature flags) in a consistent way is crucial for running the pipeline in different environments (development, testing, production). We employ a centralized configuration system using Pydantic's BaseSettings to load environment variables and group settings logically. This section outlines how we handle environment variables, `.env` files, and ensure that configuration is not duplicated or scattered in code.

Central Config (`config.py`): We have a single `config.py` module that defines Pydantic BaseSettings classes for each group of settings: - `GeneralSettings` – global settings like `ENV` (development/production), logging level, etc. - `DatabaseSettings` – contains `supabase_url`, `supabase_service_key`, etc. - `APISettings` – Anthropic and OpenAI keys, maybe model names or timeouts. - `ScraperSettings` – things like headless mode toggle, etc. - `PrefectSettings` – e.g., whether to use Prefect Cloud or local, and relevant API keys if any.

Each of these classes inherits from BaseSettings, so they automatically read environment variables that match field names. We set `env_prefix` in SettingsConfigDict for some classes to avoid collisions (e.g.,

`DatabaseSettings` might use prefix "DB_" so it reads `DB_SUPABASE_URL`). This grouping by prefix is how we separate different process configs if needed (though in our case, one process uses mostly all settings). We also use Pydantic's `.env` support to load from a `.env` file in development easily. In `SettingsConfigDict`, we specify `env_file='.env'` so it picks up that file automatically without any extra code.

No `os.getenv` **Outside Config:** We strictly fetch values only via these Pydantic settings. This means anywhere in the code where we need, say, the OpenAI API key, we do `config.api.openai_api_key` (assuming we instantiated a `Settings` object that nested these groups, e.g., `Settings = BaseSettings` that includes all sub-settings). This prevents the all-too-common scenario of different parts of code using different environment variable names or loading logic. By centralizing, we also easily support default values and type conversion (Pydantic will automatically convert types, e.g., an env var `"true"` to bool True).

Storing Secrets: For local dev, secrets (API keys) go in `.env` which is gitignored. In a deployed environment (say a server or container), we would supply these via actual environment variables at runtime (like in Kubernetes secrets or Docker `-e` flags). Because our config reading will check environment first by default, that works seamlessly (Pydantic gives precedence to actual env vars over .env file, which is ideal for production).

We are cautious to never commit secrets. Also, when sharing the repo or documentation, we ensure `.env.example` is provided with placeholder values for required settings (so new developers know what to set).

UV Tool Integration: The mention of "UV" likely refers to a tool in the Prefect/Marvin ecosystem (as seen in the Prefect example, `uv` CLI was used to bootstrap a project). In our context, **Astral** UV could be a tool to manage environment variables across dev/prod or to scaffold config classes. While we haven't explicitly used a tool named UV in our process (aside from that reference), we follow the concept of a unified environment config. Possibly, "UV" stands for a pattern or just was a formatting bullet. In any case, our approach aligns with best practices: a single source of truth for configurations and environment-specific overrides via env files or actual env.

Grouping by Process: If in future we split components into different processes or microservices, each might not need all config. For instance, the Scraper service doesn't need OpenAI API key. With Pydantic, we can easily define separate config classes for each service and only load relevant ones. We can also use environment variable prefixes to differentiate if running in same environment. For example, if we ran scraper and extractor in the same container (less likely), we could prefix environment variables like `SCRAPER_PLAYWRIGHT_HEADLESS=false` vs `EXTRACTOR_MODEL=claude`. But currently, since one process runs all, we don't have separate prefixes per service (except DB, API as mentioned).

Environment-specific Behavior: We have an `ENV` setting (e.g., `ENV=development` or `production`). In code, we occasionally branch on this. For instance, if `ENV=development`, we might enable verbose logging or not purge logs. Or use a smaller model for testing (like use Claude Instant for faster but cheaper dev runs, whereas production uses Claude 2). These toggles are all in config.

We also have `DEBUG_MODE` flag that can be turned on to skip calling real APIs and use stub data (for testing pipeline end-to-end without costs). This flag is read from env as well, and if true, our LLM extraction function will load a sample JSON from disk instead of calling API.

Loading Order: By default, Pydantic BaseSettings will load from environment variables, then .env, then default values, in that order of priority. We rely on that. We explicitly document in the README which env vars are needed. If something is missing, Pydantic will raise a ValidationError at startup, causing the app to fail fast with a clear message about missing config. This is better than a random failure later due to a missing key.

Dotenv vs Environment: In local development, running via `uvicorn` or scripts, the .env is automatically loaded (because we set `env_file` in Pydantic config). In production, we likely won't even have a .env file, but will pass real env vars. Because environment vars override .env in Pydantic's logic, a .env left in a container wouldn't override deployed secrets – environment wins, which is what we want.

Example: If `SUPABASE_SERVICE_KEY` is set in the environment of the container, that is used. If not, perhaps the .env (with a dummy or dev key) would be loaded which might cause an issue if not intended. So for production images, we do not include any .env (or we ensure it's overwritten). Our documentation for deployment clearly states to supply env vars.

Benefits: This configuration approach yields: - Type safety: If someone accidentally types an integer for a setting that should be str, Pydantic will warn/convert appropriately. - Centralized documentation of config: By reading `config.py`, one sees all possible settings and their meaning (we include comments or Field descriptions). - Ease of adding new config: Just add a field in the Pydantic class, optionally with a default. Immediately it can be set via env without further code changes elsewhere.

No Hard-Coding: We avoid magic numbers or strings in code. E.g., instead of scattering the Supabase table name in code, we define it in config (if it may change). This might be overkill for stable things, but it's a habit to minimize code changes for config tweaks.

External Config (12-factor): This approach aligns with 12-factor app principles of separating config from code. It also means the same container can run in staging vs prod just by env differences (no code difference). This simplifies deployment.

Summary: Our environment management uses `.env` for local dev convenience and environment variables for deployed environments, loaded through Pydantic BaseSettings to provide a robust configuration system. This ensures consistency and reduces errors, as settings are loaded once and then used throughout via a config object, rather than calling `os.getenv` everywhere (which can lead to inconsistent defaults or typos). As one Reddit user succinctly put it, using Pydantic for settings is preferable to sprinkling os.environ calls and is more maintainable – our implementation follows that advice.

Deployment & Infrastructure

Overview: This section describes how to deploy the FDD extraction pipeline on a local server or development machine, with considerations for containerization, infrastructure, and scaling. The current

deployment approach is tailored for local use (running on a single machine), but we outline how it can be containerized and what infrastructure components are involved (Prefect, FastAPI, Supabase).

Local Deployment

For local development or a single-machine deployment, the simplest method is to run the components directly with Python processes: 1. **Supabase Setup:** Because Supabase is a cloud service, you need a project (with the Postgres DB and storage) set up. Obtain the `SUPABASE_URL` and `SUPABASE_SERVICE_ROLE_KEY` from your Supabase project settings. Alternatively, run a local Postgres database and use that URL (our config can accommodate a direct Postgres URL, though storage would then be a local file system or an S3 bucket you manage). 2. **Prefect Orion:** Start the Prefect Orion server (if using Prefect 2.x). This can be done with `prefect orion start`. This provides a web UI at http://127.0.0.1:4200 where you can see flow runs. (If you prefer not to use the UI, you can also run flows directly without starting Orion, in which case states are ephemeral). 3. **Launching Services:** You can start the FastAPI app (which includes the scraper and possibly orchestrator) by running `uvicorn app.main:app --reload`. This will serve the API at http://127.0.0.1:8000. The orchestrator (Prefect flows) can be triggered via this API or directly via CLI. For instance, you could run `prefect run --name "weekly_scrape_and_process_flow"` if you've registered it, or simply execute the script that contains the flow (which will in turn call the tasks). 4. **Background Worker (if needed):** If using Prefect with an agent, you might need to start a Prefect agent to pick up scheduled runs, e.g., `prefect agent start -q 'default'`. In our local dev, we often run flows in-process, so an agent isn't strictly necessary unless using the scheduling without always-on process.

Everything can technically run in one process/thread, but for better parallelism, you might have the FastAPI app and the Prefect flow running concurrently. In development, we often just run the flows from a Jupyter notebook or directly in the Python process for manual control.

Docker Deployment

To isolate the environment and ensure consistency, you can use Docker. We provide a `Dockerfile` that defines a multi-stage build: one stage for dependencies (possibly using poetry or pip), and final stage with the slim runtime image (e.g., Python 3.X-slim).

For example: - The image includes necessary system deps for Playwright (like chromium library dependencies) and for any other binary (Tesseract or Poppler if needed by MinerU). - It then installs Python deps and downloads Playwright browsers (`playwright install` during build or entrypoint). - Copies the project code.

We also include `docker-compose.yml` to coordinate containers: - `app` service: runs the FastAPI app (which inherently runs Prefect flows when triggered). - `worker` service (optional): could run a Prefect agent or a dedicated process for heavy tasks. - `supabase` is not run in docker-compose since Supabase is external (but if using a local Postgres, we could define a `db` service with a Postgres image, and maybe a MinIO for storage if needed).

However, since our pipeline integrates with cloud Supabase, we just ensure the app container can reach the internet to call Supabase and LLM APIs.

Running with Docker Compose: 1. Build the image: `docker-compose build`. 2. Set environment variables: You can create a `.env` file for docker-compose (not to be confused with our application .env) or directly set environment in compose file. We include in compose something like:

```
environment:
  - SUPABASE_URL=${SUPABASE_URL}
  - SUPABASE_SERVICE_KEY=${SUPABASE_SERVICE_KEY}
  - ANTHROPIC_API_KEY=${ANTHROPIC_API_KEY}
  ...
```

and instruct users to fill those in an `.env` file that Docker Compose will use. 3. Run: `docker-compose up`. This will start the container(s). The FastAPI app will be accessible on host (we map port 8000). Prefect Orion could be running inside the app container or we might run it separately (for simplicity, we often just run it in the same container on a different thread; but one could also run Orion as a separate container – Prefect provides a Docker image for Orion server).

Infrastructure Patterns: Our deployment is relatively simple (a single machine), but it's set up in a way that can scale: - Stateless services (scraper, extractor) can be replicated if needed behind the scenes, since state is mostly in the DB. - The heavy stateful part is the database (Supabase) which is managed separately (cloud or a single Postgres instance). - If we needed to scale out, one could run multiple workers for Prefect tasks – e.g., have one container handling scraping tasks and another handling extraction tasks. Prefect or an external queue (like Redis+RQ or Celery) could coordinate. Right now, Prefect can handle parallel tasks threads, so a single process is fine.

Continuous Deployment: In a production scenario, one could integrate with CI/CD (e.g., GitHub Actions) to build the Docker image on pushes and maybe push to a registry. The deployment could then be as simple as pulling the new image and restarting the container. Because we externalize config, deployments won't override keys (they come from environment at runtime).

Monitoring & Logging: We rely on Prefect's UI and logs for now in local deployments. In a more production environment, we'd integrate with monitoring: - Use Prefect Cloud or another orchestrator so that we get alerts on failures. - Export logs to a logging service (the app logs go to stdout, so Docker captures them; one could send them to CloudWatch or similar). - Supabase provides some monitoring on DB (but that's less critical given low volume).

CI/CD Pipeline: We have a GitHub Actions workflow (for example) that runs tests and perhaps lints the documentation. Since documentation is part of our context for AI, we ensure any changes to docs are reviewed and consistent. There's also an Action to build the Docker image. For deployment, because this is local-focused, we might not have automated deployment (maybe the user just runs compose manually). But if this were on a server, we could use something like Watchtower to auto-update the container on new image, etc., or a simple SSH + docker-compose pull.

Security: - The API is not exposed publicly in our default deployment (we bind FastAPI to localhost or a dev-only port). If it were, we'd secure it (at least basic auth or token for endpoints, especially anything that triggers flows). - We make sure not to hard-code secrets in images (only via env). For instance, the

Dockerfile does not `ENV OPENAI_API_KEY ...` as that would bake it in; we only pass at runtime. - Supabase service role key is powerful, so we restrict access to the environment file.

Infrastructure as Code: If deploying to cloud, one could use Terraform or similar to set up a VM and perhaps a container orchestrator. But for now, a single VM running Docker compose is sufficient. We included a reference in docs for deploying on an EC2 or similar: essentially install Docker, set env vars, and run.

Deployment Documentation: In `deployment-infrastructure.md` (this section effectively), we highlight all of the above for an engineer who needs to deploy it or for devops. We emphasize that because it's local-first, deployment is not complex: no need for Kubernetes unless scaling out, a single machine can handle it.

Escalation & Support: We note in documentation that if something goes wrong in deployment (like pipeline errors), the on-call should check Prefect UI or logs, and possibly escalate as per the support plan (next section). We also mention any scheduled downtime needs (like if Supabase is down for maintenance, pipeline might fail that week, which is acceptable – just rerun later).

In summary, the pipeline can be run locally for development via simple commands, and can be containerized for reproducibility. It interfaces with external services (LLM APIs, Supabase) which means deployment requires ensuring network access and correct credentials. Because of small scale, a straightforward Docker Compose on a VM is our target deployment scenario, keeping things simple yet encapsulated. We've documented this so that any engineer can set it up with minimal hassle, focusing on config values and letting our code handle the rest (thanks to environment-driven settings and orchestrated workflows).

Handoff & Support Continuity

Overview: This section outlines how we handle transferring knowledge of the system to new team members or between AI development sessions, how we maintain continuity in operations, and what the escalation path is when issues arise. We utilize a living document (handoff.md) to track ongoing tasks, decisions, and context so that no knowledge is lost between sessions (especially important given AI-assisted development). We also define how issues are escalated if on-call support cannot resolve them.

Task Management and Documentation

We maintain a **Task & Session log** (in a file like `handoff.md` or an internal wiki) that is updated at the end of each development session or each major pipeline run. This log includes: - *In-Progress Tasks:* e.g., "Implement improved table parsing – Assigned to Alice – Status: debugging edge cases." We list what's being done, by whom, and any blockers. - *Pending Tasks/Backlog:* e.g., "Add diffing of year-over-year FDDs – Priority: Low – can be done after core features". This ensures future developers or even an AI agent picking up the project know what features or fixes were planned. - *Completed Tasks:* We log tasks as completed with dates and outcomes, e.g., "Extract Item 3 (Litigation) info – Completed Oct 1, 2025 – Outcome: now capturing litigation history, 5 new fields added". This gives historical context and rationale for changes, which is helpful when debugging or extending features.

By keeping this updated, if a new developer takes over, they can quickly see the state of things – what's done, what's left, known issues.

Design Decisions Log

We document significant **architectural and design decisions** in the handoff file as well. For example: - Decision: Use Supabase instead of self-hosted Postgres – *Rationale:* wanted quick setup and integrated storage. - Decision: Not to parse certain FDD items due to complexity – *Rationale:* low ROI, can revisit later. Each includes date and alternatives considered if any. This way, future maintainers understand why we did things, preventing them from undoing a decision without realizing consequences. It also helps an AI assistant—Claude can read these and not suggest solutions we deliberately ruled out.

We also track **technical debt** and issues in this doc. For instance: - "Issue: MinerU fails on scanned PDFs with handwriting – Workaround: currently skip those, perhaps integrate a better OCR in future – Priority: Medium". - "Debt: Scraper uses some hardcoded state URLs – should refactor to config – not urgent."

On-Call and Escalation Path

For operating the pipeline, we have a simple on-call rotation (since the team is small, it might just be primary and secondary). The on-call person (or team) is responsible for the weekly run and any immediate troubleshooting.

Escalation Path: If an issue arises that the on-call cannot fix quickly (e.g., a state website changed layout and the scraper breaks in a non-trivial way, or the LLM API starts returning errors consistently): 1. **Triage:** On-call checks logs, identifies the component (scraper, parser, extractor, etc.) causing trouble. 2. **Attempt Quick Fix:** If it's something minor (like an expired credential or a simple parsing tweak), on-call fixes it (we ensure they have access and some training). 3. **Escalate to Developer:** If it's a complex code issue, escalate to the engineering team (which might be just us or a specific person knowledgeable about that part). This is done by creating an urgent issue in our tracker and contacting the person (phone/Slack). 4. **External Escalation:** If the issue lies with external services (e.g., Supabase outage, Anthropic API problems), on-call will escalate to those providers. For instance, contact Supabase support if the database is down. If the Anthropic API is failing widely, possibly switch to OpenAI as primary temporarily (we have that fallback). 5. **Management Escalation:** For very serious issues (like we consistently cannot process FDDs and it's affecting deliverables), escalate to project leadership to decide on resource allocation (maybe bring in more devs, etc.).

We note the **escalation contacts** in the documentation: e.g., - Anthropic support email, - Supabase support, - Internal contacts (like "Alice – scraper expert", "Bob – deployment/DevOps"). So, if the on-call at 3am sees the pipeline failing due to, say, a new anti-bot measure on a state site, they know Alice wrote most of that scraper and can call her if urgent.

We also make sure to note the **secondary on-call** in case primary is unreachable.

Issue Logging: Every incident gets logged in an "incident log" in the handoff doc with time, what happened, and how it was resolved. This creates a knowledge base of recurring issues. For example, if three months later the same error happens, a new on-call can see "Oh, in Jan it failed with error X, and the fix was to

update the Playwright browser version." This aligns with recommended support processes to document recurring issues for quick resolution.

We avoid overloading documentation with too many trivial issues, focusing on recurring or major ones (balance between thoroughness and signal-to-noise).

Session Handoff (AI context): When using Claude in multi-session development, we rely on the `handoff.md` to preserve context. At the end of an AI session, we update it with what was done and what needs to be done next. Next time we invoke Claude, we load `handoff.md` so it can see the last context and continue seamlessly. This prevents the AI from redoing work or making conflicting changes due to lack of session memory. Essentially, `handoff.md` serves as Claude's memory between sessions, summarizing state and next steps.

Continuous Support: The pipeline runs weekly, but support is basically needed on-call at that time. If a run fails, we can rerun after fixes. Because it's not real-time, a few hours of delay are acceptable. We do communicate if a run is missed or delayed to any stakeholders (e.g., "this week's data will be a day late due to maintenance").

Handoff to New Team or Consultant: If, say, our team transitions the project to another, the documentation suite (this entire file) and the handoff log are the primary artifacts. We ensure they are up-to-date. We would also have a meeting or at least a detailed handoff note pointing to: - Key resources (repo, Supabase project, etc.). - Credentials management (where API keys are stored, who to ask for access). - Current status of the project (maybe include a short paragraph in handoff: "System is in maintenance mode / still under active dev on these features / etc."). - Escalation path (if new team takes over on-call, define their rotation).

Escalation Example: Suppose the state of California revamps their site and our scraper suddenly can't find any PDFs (and logs errors). The on-call gets an alert (maybe by noticing logs or a health check that fewer documents than expected were scraped). They check and confirm the CA scraper is broken. Quick fix not obvious (it might need re-writing parsing logic). According to escalation path, they notify the engineering team (which might involve waking up a developer if critical). If it's not immediately critical (maybe we can afford to skip CA for a week), they might just create a Jira ticket and mark it high priority for next workday, informing stakeholders that "California data will be updated next cycle due to site changes." The decision to escalate immediately vs wait would depend on criticality; we capture these criteria in our support guide (e.g., if more than 20% of sources fail, escalate immediately; if minor source, can wait a bit).

Continuous Improvement: After each incident, we evaluate if we can improve the system to prevent it or detect it earlier. For example, if CA changed site, perhaps add a monitoring script that pings known page elements weekly outside the pipeline schedule, to catch changes proactively. Or if LLM output started failing more, perhaps time to update the prompt. We list such improvements in the backlog (pending tasks) so they're addressed.

In conclusion, we strive to ensure that no matter if an AI or a human is continuing the work, they have the necessary context: through updated documentation, structured logging of tasks/decisions, and clear guidelines on where to escalate issues. The combination of these practices aligns with industry recommendations for maintaining pipelines and ensuring smooth hand-offs. The system should not live

only in one person’s head; it’s in our documents and processes, making the project resilient to personnel changes and capable of continuous operation with minimal interruption.

---

1 General Tips for Prompt Engineering - Instructor
https://jxnl.github.io/instructor/concepts/prompting/