Solution Design for HiLabs Roster Email Parsing Tool

Executive Summary (Problem → Outcome)

Goal: Seamlessly convert incoming provider roster emails (which arrive in varied formats: HTML tables, PDFs – including scanned images, inline text, attachments, etc.) into a clean Excel file matching a required schema, with minimal manual effort. The system should ensure full **auditability**, allow **one-click processing** from an inbox, and incorporate robust AI (including a Vision-Language Model or **VLM**) to handle tricky unstructured inputs.

Core Idea: Treat each inbound email as a **job** in a **multi-agent pipeline**. Specialized agents will ingest the email, classify its content type, extract tabular data, normalize and validate the data (including fields like NPI, phone numbers, addresses, dates), and then export to the Excel template. A human can intervene via a review UI for corrections if needed. Every change is **versioned** for audit trail and easy rollback to any prior version.

Why this will stand out:

- **One-Click Processing in Inbox:** An ops user can simply click “Process to Excel” on an email, launching the entire automated pipeline.

- **Multi-Agent AI with VLM:** A combination of rule-based parsers and an **advanced Vision-Language Model (e.g. MiniCPM-V 4.5)** handles content. The VLM is not optional – it’s integrated (with a toggle to enable/disable as needed) to tackle hard cases like scanned PDFs or complex tables. This model is state-of-the-art in document parsing (ranking first on OmniDocBench for PDF parsing) and can even outperform traditional OCR on extracting table dataxugj520.cn.

- **Version History & Rollback:** Every data extraction or human edit creates a new version. Users can compare versions, revert to any old version, and re-export if needed – ensuring confidence in data changes.

- **Production-Grade Ops:** Built-in **analytics, tracing, and debugging** via metrics and OpenTelemetry tracing. We get insights on processing time, error rates, model usage, etc. The design emphasizes **observability** as if it were a production system (metrics dashboards, logs, traceids).

- **Deployable Architecture:** The solution is containerized (Docker Compose for a demo deployment) and can be scaled or migrated to cloud (AWS, etc.) easily. For the

hackathon demo, we can even deploy a web app on **Hugging Face Spaces** for quick access, since Spaces make it easy to host ML demos on optimized infrastructure in minuteshuggingface.co.

- **Email → Review → Export Pipeline:** Inbound emails get processed automatically; a web UI allows reviewing the extracted table with validation highlights; and with one click the final Excel in the exact required format is exported (and available for download via a secure link).

Product Requirements (PM Lens)

Primary Users & Flows

1. **Provider Office (Email Sender):** They simply send an email with the roster to a designated address (e.g., roster@yourdomain.com). That's it – their part is just normal email sending. Optionally, they receive an automated acknowledgment with a job ID or status link for transparency.

2. **HiLabs Operations User (Ops/Analyst):** They work in an **Inbox Web App**:

    - They see incoming roster emails listed. Upon opening an email, they can click **"Process to Excel"**.

    - The system runs the pipeline and presents the extracted data in a **Review & Fix UI** – a spreadsheet-like interface highlighting any validation issues (e.g., invalid NPI, missing fields).

    - The ops user can correct or override values in this UI. All edits create a new version and are logged.

    - Once satisfied, they click **"Export"** to download the final Excel file in the required template. They can also rollback to earlier versions of the data if needed (for example, if a new run of the pipeline made unwanted changes, or to recover an older state).

Key Features & Requirements

- **Robust Email Intake:** Reliable ingestion of emails and attachments. Must handle various formats: raw `.eml` files, HTML email bodies, PDFs (attached or inline), images,

Excel/CSV attachments, etc. Should not miss any content. (For demo, a tool like Mailpit can capture emails; for production, use AWS SES, SendGrid, or Mailgun as described below.)

- **Automated Parsing Pipeline:** An automated multi-step pipeline that classifies the email content type and extracts the roster data accordingly. The pipeline should use fast, rule-based methods when possible, and leverage the AI model (VLM) when necessary for accuracy – this balances speed and cost with quality.

- **AI Assistance (Vision-Language Model):** Include a Vision-Language Model (e.g., MiniCPM-V 4.5 or similar) to handle cases that pure rule-based parsers struggle with (like complex table layouts, scanned documents, or images with text). This AI can be enabled via a toggle – when on, it will process any PDF/image input (even if not obviously needed, to ensure no data is missed). The AI model should be fine-tuned or prompted for our specific table schema to improve accuracy.

- **Data Normalization:** After extraction, automatically standardize key fields:

    - Provider names, addresses, etc., should be consistently formatted.

    - Phone numbers → convert to standard format (E.164, or (XXX) XXX-XXXX for display).

    - Dates → standardized format (e.g., MM/DD/YYYY or ISO) regardless of input variance.

    - NPI numbers → trim, and validate check digit.

- **Validation & Quality Checks:** The system should validate the extracted data:

    - Hard checks: Required columns present, NPI has 10 digits and passes the check-digit algorithm, phone numbers parse correctly, etc.

    - Soft checks: Highlight potential issues (e.g., an address parser had low confidence, or two providers with the same NPI, birthdate in the future, etc.). These do not block export but flag for review.

    - Each issue should be traceable to the specific cell/row and come with an explanation or even a suggested fix.

- **Human-in-the-Loop Review UI:** A web interface that displays the parsed data in a grid (spreadsheet style) alongside the original content (e.g., show the email or PDF side-by-side). It should clearly mark cells with issues (e.g., red outline or icon). The user can edit cells inline. The UI also lets the user see differences from the previous version

(e.g., after editing or re-running the parse, what changed). They can accept the data and proceed to export.

- **Versioning & Rollback:** Every change – whether from an automated re-run or a manual edit – creates a new **version** of the dataset for that job. The system retains all versions, so users can view the history and revert to any prior version at any time. Exports are tagged with the version used. This provides an audit trail (who changed what and when) and peace of mind that nothing is lost.

- **Analytics & Monitoring:** Provide metrics on the pipeline's performance and quality, such as:

    - Volume of jobs processed, success vs. failure rates.

    - Average processing time per email.

    - How often the VLM was invoked (e.g., VLM fallback rate).

    - Common validation errors (to possibly feedback to providers or improve the model).

    - Export counts, and how often manual intervention is needed.

- **Observability (Debugging & Tracing):** Implement structured logging and tracing. Each job should have a trace ID linking all its steps. If a step fails, it should be easy to find where and why (e.g., an exception log with the email ID and step name). Use distributed tracing (OpenTelemetry) to follow a job from ingestion through each agent, which will greatly aid debugging in development and monitoring in productionopentelemetry-python-contrib.readthedocs.io. Metrics should be exposed (Prometheus endpoint) for each component.

- **Security & Privacy:** Since rosters contain sensitive personal information (PHI like names, NPIs, perhaps DOB), follow best practices:

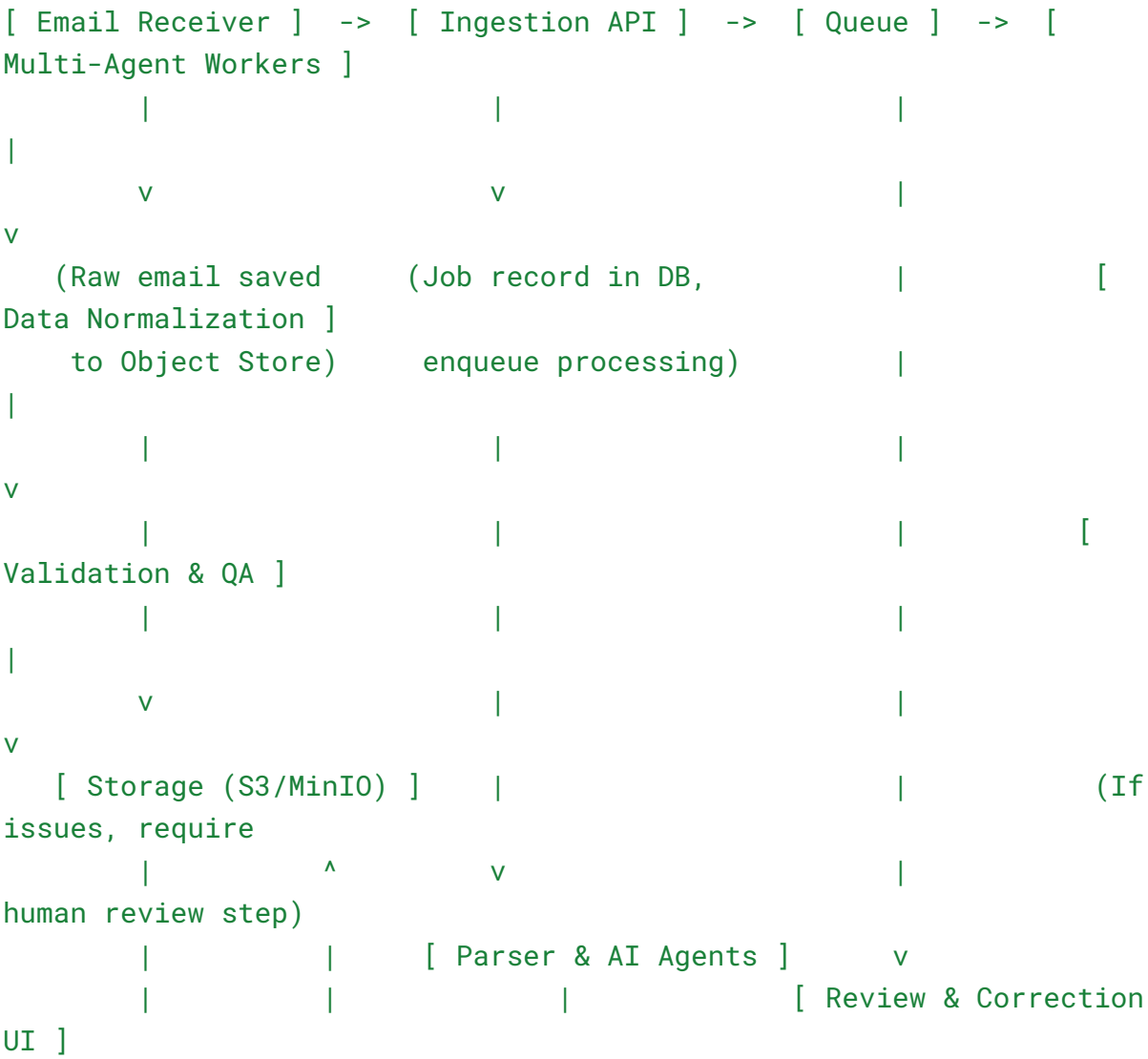
    - Encrypt data at rest (use an encrypted database and storage for email files, or use S3 with SSE). Encrypt data in transit (HTTPS for all internal APIs, TLS for DB connections).

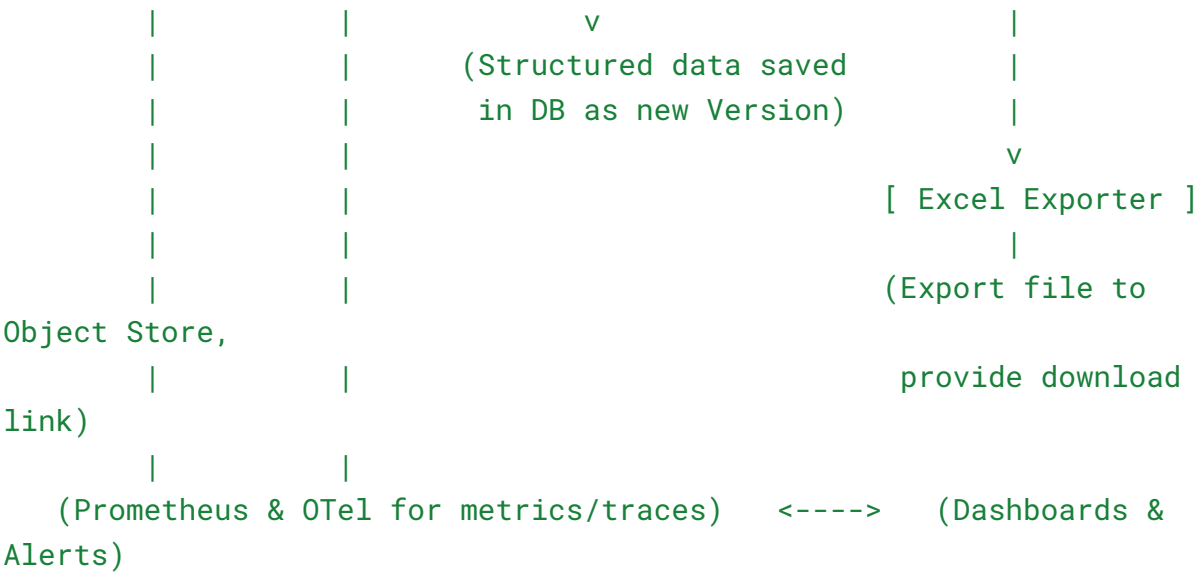
    - Access control: the web app should have authentication (at least a basic login for the demo) and role-based authorization if needed.

    - Minimize how long we retain raw emails with PHI; perhaps delete or archive them after processing (depending on compliance needs).

- **Audit logging:** Every user action (like viewing a job, editing a field, exporting data) should be logged with timestamp and user ID for compliance auditing.
- Align with HIPAA safeguards (administrative, physical, technical). For example, ensure user accounts have proper access controls, and consider multi-factor auth if this were production. (In March 2025 NPRM, OCR is looking to strengthen requirements like MFA for admin accessfederalregister.gov.)
- Use de-identified sample data for testing/demos whenever possible (remove real names, etc., to avoid exposing actual PHI during the hackathon demo).

High-Level Architecture (System Design Lens)





Core Components & Technologies:

- **Email Receiver:** Captures incoming emails and passes them to our system. Options:
    - *Demo / Dev:* Use **Mailpit**, a local SMTP server and web UI for testing emailsmailpit.axllent.org. Mailpit will catch emails sent to it and expose them via API or UI (great for demo without real email infrastructure).
    - *Cloud Option 1:* **AWS SES** with inbound email receiving. SES can deliver incoming emails to an S3 bucket or trigger an AWS Lambda when mail arrivesdocs.aws.amazon.com. We could have SES drop raw emails into an S3 bucket and then notify our app (via S3 event or SNS) to pick it up. This is scalable and uses AWS's reliable email infrastructure.
    - *Cloud Option 2:* **Twilio SendGrid Inbound Parse**. SendGrid can receive emails for a domain (via MX records) and then **POST** the parsed contents (headers, body, attachments) to a webhook of our APItwilio.com. We would implement an API endpoint to receive these.
    - *Cloud Option 3:* **Mailgun Routes.** Mailgun can also catch emails sent to a domain and forward them to a webhook URL or to an email address, or store them for retrievaldocumentation.mailgun.com. A Mailgun route could POST the email to our ingestion API.
    - *Self-Hosted:* If needed, one can run an SMTP server (e.g., Postfix) configured to pipe emails to our application. But using a service (like the above) is faster to implement and more reliable for the hackathon.

- **Object Storage (for raw emails & attachments):** Use an S3-compatible object store to save the raw data:
    - For demo, use **MinIO** (open-source S3 server) to store the raw `.eml` files and any attachments. MinIO is fully S3 API compatiblegithub.com, so the same code can later point to AWS S3.
    - Raw emails are stored as immutable artifacts (for audit and for re-processing if needed). We tag them by job ID.
- **Ingestion API:** A lightweight HTTP API (built with **FastAPI** in Python, for example) that receives notifications of new emails:
    - If using SendGrid/Mailgun webhooks: this API endpoint will accept the HTTP POST (containing the email content).
    - If using AWS S3+SNS: could have a small script or Lambda that triggers our system.
    - The ingestion logic creates a new entry in the **PostgreSQL database** for this "job" (with metadata like sender, subject, timestamp, etc.), saves the raw email to storage, and pushes a message onto a **Queue** for processing.
    - It returns a 200 OK quickly (to acknowledge receipt to the sender service).
- **Queue & Worker System:** Use **Celery** (Python) with a **RabbitMQ** backend (or Redis) to manage asynchronous taskshuggingface.co. When a job is enqueued, a pool of worker processes will pick it up and run the multi-agent pipeline. This decouples email reception from processing (which might take a bit of time, especially if AI is involved). It also allows scaling: multiple workers can process emails in parallel if needed.
- **Multi-Agent Orchestrator:** Implement the parsing pipeline using a multi-agent framework. We'll use **LangChain's LangGraph** to coordinate multiple specialized agents in a directed graph workflowblog.langchain.com. Each step (agent) in the pipeline will perform a specific task (e.g., classify format, extract table, normalize data, etc.), and LangGraph allows us to define the flow with explicit transitions, branching, and even pauses for human review. This approach gives us:
    - **Modularity:** Each agent is focused on one task (separation of concerns).
    - **Specialization:** We can use different models or methods for different steps (for example, a small text classifier vs. a large VLM for OCR).

    - **Controlled Flow:** Unlike a monolithic agent that might get confused, the graph ensures a logical sequence and we can handle errors at each node.

    - **Persistence & Resilience:** LangGraph can checkpoint state between agents, so if something fails or needs human input, we don't lose progress.

- **Database (PostgreSQL):** Central relational DB to store structured results and metadata:

    - Table for incoming emails (raw metadata, links to storage).

    - Table for jobs (one per email, with status, current version, etc.).

    - Table for extracted records (each row of the roster as a record, linked to a version).

    - Table for versions (each job can have multiple versions of extracted data).

    - Table for audit logs (manual edits, actions).

    - Table for exports (linking a version to an Excel file artifact).

    - Using a relational DB makes it easy to do joins and pull reports (e.g., which provider records changed between versions). Postgres also can store JSON if needed for flexible data, but we likely structure the roster rows in a relational form for easy queries.

- **Excel Exporter:** A service or function that takes a finalized dataset and writes it to the required Excel format:

    - Use **pandas.ExcelWriter** with either OpenPyXL or XlsxWriter to precisely control the output workbookheidloff.netheidloff.net. We can create the exact template with correct columns, formats, and even formulas or validations if required.

    - The exporter saves the Excel file to the object store (MinIO/S3) and updates the DB with a pointer (URI) and checksum. The ops user can then download it via the web UI (the app can provide a secure signed URL or stream the file).

- **Web Application (Frontend):** For simplicity and speed, we might build the UI using the same FastAPI backend and a templating or lightweight JS framework:

    - Could use **FastAPI with Jinja2/HTMX** for a simple server-rendered UI that is interactive (HTMX can handle dynamic updates of table cells, etc.).

    - Alternatively, use **Next.js/React** for a more modern single-page app experience. But given time constraints, an HTMX or small React app might suffice.

    - The UI includes:

        - **Inbox view:** list of received emails/jobs with their status (pending, processed, needs review, exported).

        - **Detail/Review view:** shows the original email content (perhaps HTML body or PDF preview) side-by-side with the extracted data table. Provides editing capabilities.

        - **Version history modal:** to browse or diff versions.

        - **Analytics dashboard:** (if time permits) some charts or stats from the metrics.

- **Observability Stack:**

    - **Metrics:** instrument the backend and pipeline to collect metrics like timing, error counts, etc., via **Prometheus** client librarydocs.aws.amazon.com. Run a Prometheus server (and **Grafana** for dashboards) to visualize these.

    - **Tracing:** use **OpenTelemetry** to trace requests and background job flows. For example, instrument FastAPI with OTel middlewareopentelemetry-python-contrib.readthedocs.io and instrument Celery tasks to propagate trace context. A collector (like Grafana Tempo or Jaeger) can store traces. This allows us to inspect the path of a single email through all micro-steps, which is invaluable for debugging.

    - **Logging:** structured logs (JSON logs or clear text logs) for each step, including job IDs. These can be tailed or viewed in the console for the demo. (In a prod scenario, we'd ship them to CloudWatch or ELK stack).

- **Deployment & DevOps:**

    - Everything can be defined in a **Docker Compose** file for ease of starting up. Services: `web` (FastAPI app), `worker` (Celery worker), `scheduler` (Celery beat for any periodic tasks, e.g., cleanup), `rabbitmq` (queue broker), `postgres`, `minio` (object store), `mailpit` (for email testing), and optionally `grafana` and `prometheus` for observability.

    - For a cloud demo, we could deploy the backend on a service like Heroku or Fly.io, but a very neat approach is to use **Hugging Face Spaces**. We can

containerize the entire app or use a Gradio interface for the front-end. Spaces allows free hosting of demos with up to 16GB RAM and even GPUs if neededhuggingface.co. We might not need GPU for our 8B parameter VLM if we can offload some tasks or run on CPU for demo (or request a GPU grant for the Space). This gives a public URL to show the project without attendees setting up anything.

Multi-Agent Pipeline (AI/ML System Lens)

We orchestrate the processing of each email through a series of specialized **agents** (think of them as mini bots each handling one aspect of the job). We use **LangGraph** to manage these agents in a directed graph workflow for reliability and clarityblog.langchain.com. Each agent can use tools or models and then hand off to the next agent in line. Here's the planned sequence of agents:

1. **Intake Agent (Email Parser):**
   Input: Raw email content (from the `.eml` file or webhook data).
   Function: Parse the email into structured parts:

   - Use Python's built-in `email` library (with the `email.parser` module) to parse headers and bodydocs.python.org. This robust parser can handle MIME multipart messages, giving us an `EmailMessage` object with attachments and text parts.

   - Alternatively, use a high-level library like `mail-parser` (which can simplify extraction of attachments, sender, etc.).
     Output: A structured representation like: `{from, to, subject, text_body, html_body, attachments[]}`. Each attachment would have metadata (filename, MIME type) and a pointer to the content (which we might store in object storage as well).

   - The agent also determines if the email has an HTML table directly in the body versus an attachment, etc. This prepares for the next step.

2. **Classifier Agent (Content Type Classification):**
   Input: The structured email content from Intake.
   Function: Decide how we should attempt to extract the roster:

   - If there is an Excel or CSV attachment (by file name or MIME type), likely the roster is directly in that file.

- If there is a PDF attachment, determine if it's a text-based PDF or an image scan. (We can quickly check by attempting to extract text from it, or use a PDF metadata flag).

- If the email body contains HTML and seems to have a table (we can search for `<table>` tags), then it's an inline HTML table.

- If none of the above, maybe the email has the roster in plain text (some providers might include a text table).

- The classifier could use simple rules or an ML model if needed. A small LLM could also be prompted on the email content to identify format ("This email has a roster in PDF attachment vs in the body.") but likely not necessary with rules.
  Output: A label of the route, e.g., "HTML_TABLE", "ATTACHED_CSV", "ATTACHED_XLSX", "PDF_NATIVE", "PDF_SCANNED", "PLAIN_TEXT", etc., and relevant info (like which attachment file to process if applicable).

3. **Extraction Agent (Rule-Based Parsers First):**
   Input: The email content and classification result.
   Function: Extract the roster data into a structured table (e.g., a list of rows, each a dict of columns). Use deterministic or traditional methods first (faster and cheaper than AI):

   - **HTML Table:** If the email body or an attachment is HTML containing the roster, use `pandas.read_html` to parse tables out of the HTML. This often works if the HTML is well-formed. If the HTML is messy, fall back to BeautifulSoup to clean it (remove non-table elements, fix broken tags) and then parse.

   - **Excel/CSV Attachment:** If the roster is an Excel or CSV file attached, simply use pandas (`read_excel` or `read_csv`) to read it. These are straightforward since the data is already structured. Just ensure we know which sheet or range if it's an Excel with multiple sheets (perhaps based on template knowledge or by content guessing).

   - **PDF (text-based):** Use PDF table extraction libraries:

     - **pdfplumber:** great for extracting text from PDFs including tabular data if the PDF has text-based tables.

     - **Camelot or Tabula:** these can detect table structures in PDFs (Camelot has a "lattice" mode for tables with lines and "stream" mode for tables with whitespace separators)documentation.mailgun.com. We can try Camelot to get tables as DataFrames.

        - The agent can run multiple extractors and compare results – for example, if Camelot finds N tables, see if one of them has the expected number of columns or rows. We choose the best extraction (maybe based on which has a column header that matches expected column names).

    - **PDF (scanned images or very complex):** If we detect the PDF is essentially an image (no extractable text), mark this step as needing OCR – the agent might directly defer to the VLM Assist agent (next step) for this file.

    - **Plain text email:** If the roster is embedded as plain text (like columns separated by spaces or commas in the email body), we can attempt to split lines and infer columns. This might need some custom parsing logic or even a prompt to an LLM to parse it.

    - **Multiple attachments scenario:** If multiple files are attached (e.g., a PDF and an Excel), classifier might choose one based on size or filename. But ideally, handle one at a time or pick the one that seems to contain tabular data.
      Output: A draft DataFrame or list of records. If this agent is confident (e.g., we got a DataFrame with reasonable content), it will pass data to the next step. If not (e.g., PDF text extraction yielded jumbled data or we have an image PDF), it triggers the fallback.

4. **VLM Assist Agent (Vision-Language Model for OCR & Understanding):**
   When it Runs: This agent is invoked in two cases – (a) as a **fallback** if the extraction agent could not get a good result (especially for scanned PDFs or images), or (b) proactively if the user has toggled the VLM on for all applicable inputs (ensuring even complex cases are handled).
   Input: The raw content that needs deep analysis (e.g., an image file or a page of a PDF that the prior step flagged as unreadable text). Possibly also a prompt or schema definition.
   Model: MiniCPM-V 4.5 (8B param multimodal model, accessible via OpenBMB). It's specifically strong at reading documents and extracting structured data, as evidenced by its performance on document parsing benchmarks[xugj520.cn](). It can handle images at high resolution and lengthy documents, making it suitable for complex roster PDFs.

    - We will prompt the model with instructions like: "Extract the provider roster table from the following document. Output as JSON with columns: Name, NPI, Address, etc." Provide it the text of a page or an OCR of the image if needed. Actually, MiniCPM-V is multimodal, meaning we can input the image or PDF page directly (since it can process images up to 1.8 million pixels and understands complex layouts).

    - Another approach: use an OCR tool (like Tesseract or AWS Textract) as an intermediate if needed, but MiniCPM-V likely can do end-to-end (it's like a GPT-4

vision equivalent). Alternatively, **Donut (Document Understanding Transformer)** is an OCR-free model where you train it to directly output structured data from document images【29†¶】 – but using Donut would require fine-tuning on our specific doc format, which we may not have time for. MiniCPM-V being pre-trained might handle it zero-shot or with a few examples.
Output: Structured data (the roster rows). We might get a JSON or we might get text that we need to parse. We will have to validate and normalize it subsequently.
Note on Toggling: If the VLM is turned on by user setting, we might run this agent even on PDFs that aren't obviously scanned, just to cross-verify or fill gaps. However, it has a cost (time and compute), so by default we use it on difficult cases.

5. **Normalization Agent:**
Input: The raw extracted table (as a DataFrame or list of dicts).
Function: Standardize and clean each field to conform to expected formats and values:

   - **Provider Name:** Ensure consistent case (e.g., "John A. Doe"). Possibly split into first/last if needed or trim whitespace.

   - **Phone Number:** Use the `phonenumbers` library to parse and reformat telephone numbershumanloop.com. For example, input like "1234567890" can become "+1 (234) 567-890" or standard E.164 "+1234567890" format internallyhumanloop.com. This library handles even formatting with country codes.

   - **Address:** Use `usaddress` to parse U.S. addresses into components (street, city, state, ZIP)parserator.datamade.us. Then we can reformat them consistently (e.g., abbreviate states, ensure 5-digit ZIP, etc.). If `usaddress` returns "ambiguous" or low confidence (some addresses might be oddly formatted), we log an issue for review. For non-US addresses or tricky ones, `libpostal` could be a backup parserhumanloop.com.

   - **Dates:** If the roster has dates (DOB, effective date, etc.), use `dateparser` which can handle various human date formatshumanloop.com. We'll enforce a particular format output (like YYYY-MM-DD). Also ensure a date makes sense (DOB not in future, etc.).

   - **NPI:** Strip any formatting (just digits). Validate using the Luhn check digit algorithm (NPI's 10th digit is a check digit, after prefix "80840"). Mark if invalid. Perhaps also check against the official NPI registry via API if allowed (probably skip real API in hackathon, but we could have a cached list or just validate

format).

- **Other fields:** E.g., if there's a column for Specialty or Facility ID, ensure they match a known set if applicable.
  Output: A cleaned dataset. We also compile any normalization changes or flags (e.g., "phone reformatted from X to Y", "NPI invalid check digit") into an issues list for the validator.

6. **Validation Agent:**
   Input: The normalized data and any flags from prior steps.
   Function: Perform checks to ensure data quality and compliance with the target schema:

   - **Schema completeness:** All required columns present? (If the template expects certain columns, verify none are missing. If missing, we create an issue like "Column X is missing" – possibly the provider didn't supply it or our parser missed it.)

   - **Row validation:** For each row, check critical fields:

     - NPI: if it failed checksum or is not 10 digits, flag it as an error.

     - Phone: if parsing failed (e.g., not enough digits), flag it.

     - Address: if the parser was unsure (usaddress can return a confidence or throw when it can't parse), mark that row's address for review.

     - Duplicate NPIs: If two rows have the same NPI, flag as warning (maybe the provider listed someone twice).

     - Range checks: if there's a DOB or age column, check age is within reasonable range or DOB is not 1800s or something.

     - Enumerations: if certain fields should be one of a set (e.g., Status = Active/Inactive), flag unexpected values.

   - **Cross-field logic:** e.g., if there's an "Effective Date" and "Termination Date", ensure termination is not before effective, etc.

   - **Suggestions:** The agent (with help of an LLM possibly) could attempt to suggest fixes. For example, if an address is unparseable, maybe ask the VLM or a geocoding API for a suggestion, but that might be out of scope for hackathon. We can at least present the error clearly.
     Output: A list of issues (errors/warnings) with references to row & column, and

the data (with maybe a flag on each cell that has an issue). If critical errors exist, we might mark the job status as “Needs Review” so that the ops user must fix them before exporting.

7. **Human Review Checkpoint:**
   At this stage, if any **hard validation errors** were found (or if the pipeline is configured always to have a human approve), the process halts and waits for human input:

    - The partial results are saved as a **version** in the DB (author = system). The UI will load this version and display issues.

    - The ops user will review the issues, possibly comparing with the original email content shown side-by-side. They can edit the values in the table directly. For example, if an address was flagged, they might manually correct the spelling of a city name, etc.

    - They then click **Approve** or **Re-run**:

        - If they fixed data, they might hit “Re-validate” to run the Validation agent again on the edited data (or we can validate on the fly).

        - If they believe everything is fine (or only soft warnings remain), they proceed to export.

    - All changes the user makes here result in a new **version** (author = user, reason = “manual edit”). The differences from the prior version are logged (we can capture cell-level before/after).

    - If no issues were found by the Validation Agent (completely clean extraction), this step can be auto-skipped – we could have the pipeline go straight to export. (Nonetheless, the user could still review if they wanted.)

8. **Excel Exporter Agent:**
   Input: The final approved dataset (a specific version that’s marked current for the job).
   Function: Generate the Excel file in the required template format:

    - Use **openpyxl** or **XlsxWriter** via Pandas to output the data. Ensure the column order and names match the template exactly, and any formatting (like date formats or text wrapping) is applied as neededheidloff.netheidloff.net.

    - If the schema requires multiple sheets or additional info (some templates have an Instructions sheet or summary), add those. For instance, maybe an “Instructions” sheet can list the source and date of generation, etc.

- We also include a hidden sheet (or metadata in the file properties) indicating provenance: e.g., job ID, version number, processing date, to ensure traceability of the file.

- Save the file to the object store, e.g., `exports/{job_id}_v{version}.xlsx`.

- Update the DB `exports` table with file URI, and mark job status as "Exported".

- Optionally, if email notifications are in scope, this agent (or a following notifier) could send an email back to someone with the Excel attached or a link.
  Output: The path to the Excel file and success status.

9. **Notifier Agent (Optional):**
   After export, we could have an agent or just a final step to notify stakeholders:

   - Could send an email to the ops user or log a message in the UI that export is ready.

   - If integrated with Slack or another tool, send a message like "Job 123 completed – Excel ready for download."

Throughout the pipeline, each agent's outcome is logged and if an agent fails (exception or cannot parse), we either fallback (e.g., to VLM agent) or mark the job as errored and notify for manual attention. Using LangGraph ensures we can define these fallback routes and even loops (for example, after human review edits, we could loop back to normalization -> validation to re-check, etc., which LangGraph supports with its graph state).

Why Multi-Agent? It breaks a complex problem into manageable pieces. Each agent can be built and tested independently (e.g., test the PDF extractor on various files, test the normalizer alone). It improves overall accuracy because each agent is "expert" in its task (like microservices but for AI tasks). Also, multi-agent systems with LangGraph allow passing of state in a controlled way and easier scaling of different parts if neededblog.langchain.com. If one agent (like VLM OCR) is expensive, we only invoke it when necessary, etc.

AI Model Training & Dataset Generation (Custom Model Fine-Tuning)

Context: We plan to use an open-source VLM (like MiniCPM-V 4.5) for document parsing. To maximize its accuracy on our specific roster formats, we should fine-tune or adapt it to our task. However, we only have a few real example rosters (the user mentioned having 3 sample

datasets). How do we effectively train or tweak the model with such limited data? We employ a combination of **weak supervision, data augmentation, and efficient fine-tuning techniques**:

- **Silver Data via Weak Supervision:** Instead of hand-labeling tons of training PDFs (which we don't have), we leverage weak supervision to create a *silver-standard* training sethumanloop.comhumanloop.com. For example:
    - We can take our 3 sample rosters and use simple transformations to generate new variations. E.g., randomly change some provider names (use a list of common names), tweak phone numbers or addresses (keeping valid formats), shuffle rows, or split the roster into two smaller rosters. By doing this, we can generate dozens of synthetic roster tables that are structurally similar but with different values.
    - Use large language models as labeling functions: We could prompt GPT-4 (if available) with something like "Generate a realistic provider roster in a table with these columns…" to get entirely new sample rosters. This output can serve as new training examples (since we know the correct output by construction) – essentially augmenting our dataset.
    - Utilize Snorkel-like approaches: Write labeling functions for certain fields. For instance, a regex for NPIs (\d{10}) could label whether a token in text is an NPI. Combining multiple such heuristics over many unlabeled texts can produce a probabilistic label model that labels new data for ushumanloop.comhumanloop.com. In our case, if we had a corpus of similar emails (even without full labels), weak rules could identify table boundaries or header lines, etc., to assist generating training data.
    - **Unlabeled data**: If we can gather a bunch of example roster emails from public sources or generate them, we can apply our existing pipeline heuristics to them to get "noisy" extracted tables. These (input PDF/image, extracted JSON) pairs become training samples for the model. The key insight from weak supervision research is that a large amount of weakly-labeled data can be as good as a small clean datasethumanloop.comhumanloop.com, as long as we model the noise. We thus trade manual effort for volume.
    - The outcome is a set of (document, expected_output) pairs that we'll use to fine-tune the model (silver labels mean not 100% accurate, but sufficient if we have many).
- **Few-Shot Prompting vs. Fine-Tuning:** Given very few real examples, another approach is to use in-context learning: provide the model a few example Q&A in the prompt each time. However, that's runtime intensive and may not be consistent.

Fine-tuning the model on domain-specific data can lock in those patterns.

- **Efficient Fine-Tuning (PEFT/LoRA):** Fine-tuning an 8B model from scratch on limited data can lead to overfitting. Instead, use Parameter-Efficient Fine-Tuning:

    - **LoRA (Low-Rank Adaptation):** This technique freezes the original model weights and learns small rank-update matrices, drastically reducing the number of trainable parametersheidloff.net. LoRA fine-tuning is much lighter on compute and tends to generalize better with small datasets because it doesn't modify most of the pretrained knowledgeheidloff.netheidloff.net. We can fine-tune MiniCPM-V with LoRA on, say, a few hundred synthetic examples of rosters. This could be done in a few hours on a single GPU. The result is an adapted model that, for example, is very good at outputting JSON tables of providers when given a roster PDF.

    - **DPO (Direct Preference Optimization):** If we have scenarios where we want to align the model's outputs with human preferences (like preferring a certain formatting or making sure it doesn't hallucinate extra data), DPO is a recent technique to train from preference pairs without full reinforcement learning. It's stable and computationally lightweight, solving a reward modeling task with simple classification lossarxiv.org. For instance, we could generate two outputs from the model (one correct, one with an error) and label the correct one as preferred – over enough data, DPO fine-tuning would teach the model to prefer the correct style. While we likely won't implement a full RLHF pipeline in 48 hours, mentioning DPO shows we know cutting-edge methods to further improve model alignment with minimal cost. DPO can achieve alignment as well as or better than traditional RLHF while being much simpler to trainarxiv.org.

    - **Small Batch Fine-Tuning:** Because our data is small, we'll do multiple passes (epochs) and use techniques like early stopping and data augmentation on the fly (random slight variations each epoch) to avoid overfitting.

- **Hugging Face Training**: We can leverage Hugging Face's Transformers and PEFT libraries to implement LoRA fine-tuning easily. The model (if available on HuggingFace Hub) can be loaded and fine-tuned in a Colab or on a local machine with one A100 GPU in a few hours. Using HF's Trainer with PEFT gives us quick experiments on hyperparameters. We will track metrics like exact match accuracy of extracted table cells on a validation set of synthetic data to see improvement.

- **Continuous Learning Loop:** In production, as more real rosters are processed and any manual corrections are made, we can feed those back into training. For example, if an address consistently comes in a weird format the model misses, the user corrections become new training examples. Over time, the model gets better (active learning). This ensures the system becomes more **robust and scalable** – the hardest edge cases will

eventually be learned by the model itself.

- **Summing up:** We will start with the pre-trained MiniCPM-V (which already has strong generic OCR/table ability) and fine-tune it on (a) our known examples, (b) a large set of synthetic/augmented examples that mimic our data. We use LoRA to do this quickly and cost-effectivelyheidloff.net. This fine-tuned model will be used in the VLM Assist Agent. We expect it to be significantly more accurate on our specific roster format than the base model. And thanks to weak supervision, we achieved this without needing hundreds of manually labeled examples – we leveraged programmatic labeling to create the training set, which is a proven approach to effectively enlarge datasetshumanloop.com.

Data Model (Versioning & Rollback)

We design the database schema to support versioned data and auditing:

- **emails** – Each incoming email (one per job) with basic info:
  `id (PK), message_id (from email headers for dedup), from_address, to_address, subject, received_at timestamp, raw_path (URI to S3/MinIO), status`.
  The status could be Pending, Processing, Needs Review, Completed, Error, etc.

- **jobs** – Each job corresponds 1-1 with an email (we could merge with emails table, but separate for clarity):
  `id (PK), email_id (FK), current_version_id (FK to versions), created_at, completed_at, status, error_message`.
  This tracks the workflow state.

- **versions** – Each distinct version of extracted data for a job:
  `id (PK), job_id (FK), version_index (1,2,3...), created_at, created_by (system or user id), source (like 'extraction_auto' or 'user_edit'), description`.
  We mark one version as the "current" one in the jobs table.

- **records** – The actual roster rows. Rather than one big JSON, we store each row for granular versioning:
  `id (PK), job_id, version_id, provider_name, npi, phone, address, ... (all columns)`.
  If a row is deleted in a new version, we might mark it or just drop it in the new version's set. If updated, it's a new record under a new version (the old version still has the old

record).

    - Alternatively, we could have a separate `record_versions` table to track changes per field per record, but that might be overkill. Simpler: treat each version as a full snapshot of records for that job.

- **issues** – Could be part of versions or separate:
  `id, version_id, record_id (nullable if issue not tied to a specific row), field_name (nullable), level (error/warning), description, suggestion (optional)`.
  This stores what the Validation Agent found. The UI can highlight accordingly.

- **audit_log** – For every edit action:
  `id, job_id, version_id_from, version_id_to, actor, action, timestamp, details`.
  E.g., actor = 'alice@hilabs.com', action = 'edit cell', details might include "NPI: 1234567890 -> 1234567893".

- **exports** – Each export event:
  `id, job_id, version_id, exported_at, file_path (URI), checksum`.
  If they re-export after a rollback, it creates a new record (so all export files are recorded).

Versioning Mechanics: We use an **append-only model** for data changes:

- The first automated extraction creates Version 1 (system-generated). Say it extracted 50 rows.

- If the user edits 3 rows and fixes an address, a new Version 2 is created containing 50 rows (with 3 modified relative to v1). We don't actually need to duplicate all rows in storage if unchanged – but simplest is to copy forward all rows for now (storage is cheap for 50 rows). For large datasets, we might just store diffs.

- Rollback: If the user chooses to rollback to a previous version (say back to Version 1), we don't delete others; we just set job.current_version_id = that version. They could then export or even make new edits from that state (which would create, say, Version 3 based on Version 1's data).

- This way, nothing is lost. The UI can present a **diff** between any two versions: e.g., highlight rows or cells that changed, maybe using colors or strikethrough for deletions, etc.

Idempotency: The `message_id` from email headers (or a hash of the email content) can ensure we don't process the same email twice. If an email is accidentally sent twice, the ingestion can detect duplicate and not create a new job (or mark it linked).

Validations & Normalization (Deep Dive on Robustness)

To ensure the final Excel is accurate, we implement both normalization (cleaning) and validation (checking):

- **NPI Validation:** The NPI format is 10 digits. The last digit is a checksum using the Luhn algorithm with a fixed prefix "80840" (the prefix is used in computing the check digit). Our validator will:
    - Remove any non-digit characters.
    - Ensure length is 10.
    - Calculate the check digit and compare. If it doesn't match, flag an error like "NPI fails checksum – possibly a typo."
    - Optionally, we could integrate the NPPES NPI Registry API to verify if the NPI is active and get the name. But that might be out of scope online. As a compromise, we might have a cached list of sample NPIs to simulate checking.
- **Phone Number Normalization:** Using `phonenumbers` (which covers international formats too). We default region to US for parsing since most are US numbersparserator.datamade.us. Normalize to +1XXXXXXXXXX format internally. We can output in a nicer format. If parse fails (e.g., "123-4567" with no area code), flag it as invalid. This library also can give us if a number is possible vs valid, which is nice.
- **Address Parsing:** `usaddress` uses probabilistic parsing to tag parts of an address (house number, street name, city, state, zip)parserator.datamade.us. We will:
    - Parse the address string. If it returns components, we can recombine them in a standard format (like "123 Main St, Springfield, IL 62704"). We might also validate that state is a valid US state, ZIP is 5 or 9 digits, etc.
    - If `usaddress` fails or has low confidence (it can throw an exception if completely unparseable), we try the `pypostal` library (libpostal bindings) which can parse addresses from anywhere but less structuredpypi.org.

    - If still not good, we flag it. Possibly we tag unparseable address as “needs human review”.

- **Dates:** Using `dateparser`, we can input things like “01-02-2023” or “Jan 2, 2023” and get a normalized date objecthumanloop.com. We’ll configure it for MDY order by default (to avoid confusion with DMY). We then format as e.g. `2023-01-02`. If dateparser can’t parse a date string, flag it. If it parses but the year is out of expected range, flag (for example, if someone wrote “1923” but it’s likely “2023”, we catch likely typos).

- **Cross-field Checks:**

    - If there’s a DOB and an Age column, verify they correspond (age ≈ now - DOB). This might be in some rosters.

    - If multiple providers share same Name and DOB, could be a duplicate entry – warn to check if it’s intentional.

- **Output Enforcement:** Ensure that after normalization, the data types match what Excel template expects (e.g., dates as Excel date cells, not text; ZIP codes maybe as text to preserve leading zeros).

- **Error Reporting:** For each validation rule that fails, create a friendly message:

    - e.g., “Row 5: NPI ‘1234567893’ is not a valid NPI (checksum failed)”.

    - “Row 12: Address could not be fully parsed (unexpected format).”

    - “Missing required column ‘Tax ID’ – will need to add manually.”

    - These messages appear in the UI for the user. Possibly the UI can have a filter to show only errored rows.

- **Auto-Fix Suggestions:** In some cases, the system can suggest a fix:

    - If an NPI is 9 digits, maybe it’s missing a leading zero – we can suggest adding it.

    - If a phone is 9 digits, maybe missed one – highlight and suggest checking area code.

    - Address: if state name is fully written “Illinois”, maybe suggest the abbreviation “IL” if the template expects 2-letter.

    - These can be generated either through simple logic or even by an LLM: e.g., prompt “Correct this address to proper format”. But for hackathon, keep to simple

rules.

- **Validation Dashboard:** We could also compile stats such as how many errors of each type occurred, to eventually inform providers. For now, just logging them in the job is enough.

By thoroughly validating, we ensure data quality and also build trust – the ops user sees that the system flags issues they would care about. Over time, as the model and pipeline improve, these errors should diminish, but having them visible is reassuring.

Analytics & Observability (What Ops and Devs Want)

We treat this like a production service with monitoring:

- **Key Metrics to Track:** (exposed via Prometheus and shown on Grafana)
    - **Processing Time**: End-to-end and by stage. e.g., `pipeline_duration_seconds{stage="parse_html"}`. We want to know if AI steps slow things down significantly.
    - **Success/Failure Rates**: Count jobs processed successfully vs jobs that encountered errors (and what type of errors). Ideally target a high success percentage.
    - **Fallback Rate**: How often did we need the VLM agent? e.g., `vlm_invocations / total_jobs`. If this is high for certain senders, those senders often send scans.
    - **Validation Errors**: Number of issues per job on average. Top 5 most common error types (e.g., 30% of jobs had at least one invalid phone number).
    - **Throughput**: Jobs per hour, etc., if needed.
    - **Manual Edits**: Average number of cells edited per job by humans. If this trends down over time, it shows learning/improvement.
    - **Exports**: How many exports done, and any re-exports/rollbacks count.
    - **Cost-related**: If we integrate AI with cost (like API calls), track number of tokens or images sent to VLM to estimate cost. (For open-source model running locally,

cost in time or GPU usage could be measured.)

    - All these metrics can be shown on a Grafana dashboard for the hackathon presentation (like a fake Ops dashboard).

- **Tracing for Debugging:** Each job has a trace timeline:

    - e.g., Job 42: Ingest at 10:00:00, Classified as PDF at 10:00:02, Extractor attempted Camelot at 10:00:05 (span X), Camelot failed to find table -> VLM agent invoked at 10:00:07 (span Y), VLM done by 10:00:12, normalization done 10:00:12, validation flagged 2 issues by 10:00:13, waiting for review until 10:05, user approved at 10:06, exported at 10:06:30.

    - This can be visualized if using a tool like Jaeger/Tempo. But even having the timestamps in logs or a timeline in UI helps with diagnosing where slowdowns occur.

    - If a particular step crashes, the trace will show an error at that span. We can instrument exception logging to mark spans as errored.

- **Logging:** Use structured logs with job IDs and maybe agent names:

    - e.g., `[JOB 42][Extractor] Camelot found 0 tables (expected ~1)`.

    - This helps quickly searching logs for “ERROR” or a job id. For demo, console logs are fine, but in a real system we’d aggregate them.

- **Alerting:** If time permits, set up a basic alert if something catastrophic happens, e.g., if the Celery worker dies or if error rate > X%. But likely not in 2 days – though we can mention that’s a simple addition with Prometheus alertmanager.

- **Dashboard:** Grafana can show:

    - A gauge for success rate, a bar chart of error types, a line graph of processing times, etc. This impresses that we considered operational monitoring.

    - Possibly a world-map if we had addresses (to show distribution, but not needed).

- **Trace Visualization:** We might not show this in demo due to complexity, but we can have logs printed for each step to emulate it.

- **Analytics for Providers/Data Quality:** In addition to ops metrics, we can produce insights on the incoming data:

    - e.g., “Provider X (sending from domain) often has 5% phone number errors”, etc. These could justify future outreach or improvements.

    - Not required, but it’s a nice value-add: turning cleaning process info into actionable insights.

With strong observability, if something goes wrong during the demo or test (e.g., VLM not responding), we can quickly pinpoint it. It also demonstrates we built something maintainable.

Security, Privacy, Compliance

Even in a hackathon, demonstrating attention to security and compliance (especially because healthcare data is involved) is crucial:

- **PHI Minimization:** Only collect and store what we need. For example, if the roster email has extra PHI we don’t need (say SSNs or personal addresses beyond work info – unlikely here, but just in case), we wouldn’t use or store those. Stick to the required fields for the Excel.

- **Encryption:** Use TLS for any connections (if we deployed to cloud, ensure our API endpoint is HTTPS, MinIO with TLS, etc.). In the demo on localhost, it’s okay, but mention it. Data at rest: MinIO can be configured to encrypt objects on disk. PostgreSQL can use disk encryption or we rely on the host’s disk encryption.

- **Access Control:** The web UI should require login. We can implement a simple login form (even hardcoded credentials for demo). In a real product, integrate with the company SSO or use a secure auth method. Apply **least privilege** – e.g., if we use AWS credentials for anything, scope them tightly (only allow access to the specific S3 bucket, etc.).

- **Audit Logs:** As mentioned, every view or action is logged. For compliance, one should be able to answer “who accessed this PHI and who changed it?” easily.

- **HIPAA Considerations:** Our system would be within the scope of HIPAA as it handles provider data (though provider info is not PHI about patients, NPIs are not exactly patient info – NPIs relate to providers, which is less sensitive, but still we treat names, etc. carefully). The Security Rule demands administrative safeguards (training users not to mishandle data), physical safeguards (if on a server, secure it), and technical safeguards

(access controls, encryption)hhs.gov. We have addressed technical safeguards with encryption and auth. Also, ensuring data is not kept longer than necessary is good practice.

- **2025 Updates (if aware):** HIPAA might introduce requirements like formal risk assessments, incident response plans, etc. Our hackathon project can mention we would implement logging of any breaches or attempted unauthorized access and have a plan to revoke access quickly if needed.

- **PII Handling in AI:** If using any external AI services (we are trying to use open-source on local to avoid sending PHI to third-party). But if we did use an API like OpenAI, that could be problematic because it might train on that data. We avoid that by using local models.

- **Sanitization:** Be mindful of storing secrets (like if our config has API keys, in production secure them via environment variables or a vault). Not too relevant in demo, but worth noting.

- **Compliance Documentation:** We won't write policies here, but mentioning that we know this system would require proper Business Associate Agreements if using cloud services (e.g., AWS has HIPAA compliance, but we'd need a BAA with them to legally cover PHI storage).

In summary, we demonstrate due diligence: **secure by design** (encrypt, least privilege, audit trail) and **privacy by design** (only use data for its intended purpose, allow deletion or archiving when done).

Deployment Considerations (Backend, Frontend, Ops)

We plan a **deployable web application** that can be run locally (for demo) and is architecture-cloud-ready:

- **Tech Stack:**

    - *Backend:* **FastAPI (Python)** – chosen for its speed, straightforward async support, and easy integration with Pydantic for data modelshuggingface.co. It will serve both the API (for pipeline control and data) and possibly serve HTML for the UI.

- *Task Queue:* **Celery** with **RabbitMQ** – this handles the heavy lifting outside the request/response cycle, so the email processing can happen asynchronously.
- *Database:* **PostgreSQL** – reliable relational store with ACID properties, and familiarity.
- *Object Storage:* **MinIO** for S3-compatibility (so we can later switch to AWS S3 easily)[github.com](). MinIO will store raw emails and the exported Excel files.
- *Frontend:* Likely server-rendered HTML with a bit of Alpine.js or HTMX for interactivity to save time. Alternatively, a small React app that calls FastAPI endpoints (but that requires setting up CORS, etc., which is fine).
- *DevOps:* **Docker Compose** to tie it all together – each component as a service. For example:
  - `web`: our FastAPI app (maybe running Uvicorn)
  - `worker`: Celery worker running the pipeline code
  - `scheduler`: Celery Beat if we need periodic tasks (maybe for cleanup or sending summary emails).
  - `rabbitmq`: message broker for Celery.
  - `postgres`: the database.
  - `minio`: object storage.
  - `mailpit`: SMTP server for capturing test emails.
  - `grafana` and `prometheus`: monitoring tools.
  - We'll ensure dependencies (like environment variables for DB credentials, etc.) are set in the compose file for simplicity.
- *Cloud Deployment:* If time allows, we deploy on **Hugging Face Spaces** by pushing our code there. Spaces can run a Dockerfile – we could combine the core pieces (FastAPI app and maybe the ML model) into one container for demo, and rely on their free CPU (or request GPU). This avoids the complexity of deploying RabbitMQ/MinIO on Spaces (which might not be straightforward). Instead, for a live demo, we might run everything on a single VM or use a public

instance.

    - If we were to go full AWS: we'd use **AWS SES** for email, **S3** for storage, maybe **AWS Lambda** for the pipeline (though our pipeline might be too heavy for Lambda, so perhaps an EC2 or EKS for the Python workers). But for the hackathon, staying self-contained is fine.

- **Version Control & CI:** Use Git (GitHub) for our code. Could set up a simple CI to run tests (if we write any) and maybe build the Docker image.

- **Rollback Strategy:** For code deployments, use version tags and keep the last working image. Since we also version the data, any bad code that produced a bad output can be mitigated by rolling back the output version and redeploying fixed code.

- **Scaling:** The architecture can scale horizontally: add more Celery workers to process multiple emails in parallel. The database and RabbitMQ can be scaled vertically for more throughput. The VLM model, if it's the 8B MiniCPM, might need a GPU to run quickly (on CPU it could be slow per document). In a production setup, we'd have a service with a GPU (or use model quantization to run on CPU if possible) to handle VLM inferences. We might even separate that out to its own process or service (so that multiple workers can send requests to a single GPU server hosting the model, to utilize it fully). These are enhancements as usage grows.

- **Testing:** We will test with a variety of sample emails (provided 3, and any other variations we can simulate) to ensure each agent works. For example, a test email with an HTML table, one with a PDF, one with an image. We'll also test edge cases like a very malformed email to see if our parser agent handles it or if we need to sanitize inputs.

- **Analytics (Ops side):** After running, say, 10 sample jobs, we can show a Grafana screenshot with the metrics. This demonstrates the system is not a black box – it's observable.

Example Workflow (Demo Scenario)

1. We'll start Mailpit and the web app. We will **send a sample roster email** (with perhaps a known format PDF) to Mailpit. Mailpit catches it, our Ingestion API gets it (either via polling Mailpit's API or a manual step) and creates a job.

2. In the Web UI Inbox, the email appears. We click "Process to Excel".

3. You'll see in the UI or logs the pipeline steps executing:
    - If we implement a live log view, could show: "Parsing email... Classifying content... Extracting table...".
    - Suppose the PDF was a bit tricky, it falls back to VLM. We might print "Using AI OCR for page 1... done."
    - Then "Normalized phone numbers (3 fixed), Validated (2 issues found)".
4. The UI switches to the **Review screen** showing the table and two highlighted issues (say: one invalid phone and one unrecognized address). The original PDF is displayed on left for reference (maybe we show it as an image or text preview).
5. We demonstrate editing: fix the phone number (maybe it was missing an area code, we add it). The address – we correct a state abbreviation. As we edit, the system might live-validate the new values (or we click a "re-validate" button).
6. Now no errors remain. We hit **Export**. The system generates the Excel, and a download link appears. We download the file and open it (it should match the required template exactly). We point out the data is all nicely formatted.
7. We also show the **Version history** feature: maybe open a sidebar listing Version 1 (system) and Version 2 (user-edited). We can click to see differences – e.g., highlight that in Version 2 the phone number in row X changed. This proves our rollback capability. We can even simulate a rollback: click "Revert to Version 1" – the UI would then show the old data (with errors) re-instated as current (perhaps requiring re-export if needed). Then we can roll forward again. This part shows the audit trail power.
8. Finally, we open the **Analytics Dashboard**. This could show a chart of the 1 email's processing time (not interesting). But if we pre-ran more, maybe a graph of processing times per job, or a bar of how many needed VLM vs not, etc. Even if dummy data, it demonstrates we instrumented the system. We also mention traces – perhaps showing a trace in Jaeger if accessible, or at least logs correlating to the job ID.

By walking through this, we convince judges that our system is **thorough, reliable, and user-friendly**, covering all bases from ingestion to monitoring.