Pocket.cm Technical Assessment: AI Onboarding Agent

Scenario

Pocket.cm is a B2B SaaS platform. Our new customers often send us their employee or inventory lists in unstructured formats (PDF invoices, messy Excel sheets, Word docs) rather than clean CSVs.

We need a **FastAPI Microservice** that acts as an "AI Agent." It should accept these raw files, use a combination of standard libraries and AI/LLM logic to extract structured data, and use **Pydantic** to enforce strict validation rules before syncing to our internal API.

Core Objectives

Develop a Python/FastAPI application that handles the following pipeline:

1. **Ingest:** Securely upload files (PDF, DOCX, CSV, XLSX).
2. **Extract (The "AI" Part):** Convert raw document content into structured data.
3. **Validate (The Pydantic Part):** Use Pydantic models to enforce business rules and data types.
4. **Sync:** Send the clean data to a mock destination API asynchronously.

Technical Requirements

1. File Handling & Security

- **Endpoints:** Create a POST /upload endpoint that accepts file uploads.
- **Formats:** Support .csv, .xlsx, .pdf, .docx, and .json.
- **Security:** Implement validation to prevent malware uploads (e.g., check file magic numbers/MIME types, not just extensions) and prevent directory traversal attacks.

2. Intelligent Data Extraction

- *Note: Given your background in GenAI, we encourage using an LLM approach here for unstructured data.*
- **Structured Files (CSV/JSON):** Use standard libraries (pandas) to parse.
- **Unstructured Files (PDF/DOCX):** Implement an extraction layer. You may use libraries (like pdfplumber) OR an LLM integration (OpenAI API/LangChain/Local Model) to extract specific fields.

3. Pydantic Modeling & Validation (Critical)

- You **must** define a **Pydantic model** (e.g., class CustomerRecord(BaseModel):) to represent the target data schema.
- **Target Fields:**
    - customer_name (String)
    - email (String)

    - subscription_tier (String - Enum: "Basic", "Pro", "Enterprise")
    - signup_date (Date)
- **Validation Logic:** Implement **custom Pydantic validators** (@field_validator or @model_validator) to handle:
    - **Email Validation:** Ensure the email format is valid.
    - **Tier Normalization:** If the extracted tier is "Professional" or "Prem," automatically map it to "Pro." If it's unrecognized, default to "Basic."
    - **Date Parsing:** Ensure signup_date is converted to a standard YYYY-MM-DD format, handling potential variations in input (e.g., "Jan 1st, 2024").

4. Async API Integration

- Once data is validated against the Pydantic model, use aiohttp to POST the JSON serialization of the model (model.model_dump_json()) to a mock external endpoint.
- **Mock Endpoint:** You can use a service like Webhook.site or implement a dummy endpoint within your own app that just logs the receipt.
- **Resilience:** Implement a retry mechanism with exponential backoff for failed network requests.

5. Rate Limiting

- Use slowapi (or similar middleware) to limit the upload endpoint to **5 requests per minute** per IP.

Deliverables

1. **Code Repository:** A clean Git repository containing the source code.
2. **Docker Support:** A Dockerfile and docker-compose.yml to spin up the service easily.
3. **Documentation:** A README.md explaining:
    - How to run the app.
    - **Pydantic Implementation:** Briefly explain how you structured your models and validators.
    - Design decisions (specifically: why you chose your extraction strategy).

Grading Matrix (100 Points)

Category	Criteria	Points
Architecture	Clean FastAPI structure, Async implementation, and dependency management.	25

Pydantic Proficiency	Correct use of Models, Types, and **Custom Validators** to enforce business logic.	25
Functionality	File parsing works; Rate limiting is active; API integration handles retries.	25
AI/Data Engineering	Quality of the extraction logic. Does it handle messy PDF data gracefully?	15
Security & Quality	Secure file handling and readable code.	10

Bonus Points (Optional)

- **Pydantic Settings:** Use pydantic-settings to manage environment variables (API keys, DB URLs).
- **Unit Tests:** pytest coverage specifically for the **Pydantic validation logic**.
- **Structured LLM Output:** If using an LLM for extraction, use a library like instructor or LangChain's Pydantic parser to force the LLM to output JSON matching your Pydantic schema directly.