Screening Round

Next Steps – AI Full Stack Developer Application

Thank you for reaching out to Humaein for the role of AI Full Stack Developer.
To qualify for the first interview, as part of our screening process, please complete the following**:**

1. Complete the two take-home case studies attached.
2. Share two separate GitHub links — one for each case study
3. Respond to contact@humaein.com with – a. Your latest resume, b. What excites you about Humaein; c. Why you are the best fit for us?

We look forward to reviewing your submission. — Team Humaein

Profile of 1st interviewer - https://www.linkedin.com/in/drnadeemahmed/ Website - https://humaein.com

Case Study #1: Claim Resubmission Ingestion Pipeline

Scenario:
You're tasked with building a **robust pipeline** for a healthcare data engineering team. This system will **ingest insurance claim data** from multiple external Electronic Medical Records (EMR) systems. These EMR systems vary widely in how they format and label their data. Your goal is to standardize this input, analyze it, and **flag claims eligible for resubmission** based on defined and inferred business rules.

Objectives:
Your pipeline should:

1. **Ingest and normalize** data from multiple EMR sources (structured differently).
2. **Unify the schema** into a common format.
3. **Determine eligibility** for claim resubmission based on a combination of deterministic and inferable logic.
4. **Produce a clean output** for downstream automation.
5. Include **code comments, exception handling**, and **simple metrics or logging**.

Data Sources

You will receive data in various formats:
Source 1: `emr_alpha.csv` (Flat CSV format)

```
Source 1
c
CopyEdit
claim_id,patient_id,procedure_code,denial_reason,submitted_at,status
A123,P001,99213,Missing modifier,2025-07-01,denied
A124,P002,99214,Incorrect NPI,2025-07-10,denied
A125,,99215,Authorization expired,2025-07-05,denied
A126,P003,99381,None,2025-07-15,approved
A127,P004,99401,Prior auth required,2025-07-20,denied
```

Source 2: `emr_beta.json`
(Nested JSON format with inconsistent keys)

```
Source 2
json
CopyEdit
[
  {
    "id": "B987",
    "member": "P010",
    "code": "99213",
    "error_msg": "Incorrect provider type",
    "date": "2025-07-03T00:00:00",
    "status": "denied"
  },
  {
    "id": "B988",
    "member": "P011",
    "code": "99214",
    "error_msg": "Missing modifier",
    "date": "2025-07-09T00:00:00",
    "status": "denied"
  },
  {
    "id": "B989",
    "member": "P012",
    "code": "99215",
    "error_msg": null,
    "date": "2025-07-10T00:00:00",
    "status": "approved"
  },
  {
    "id": "B990",
    "member": null,
    "code": "99401",
    "error_msg": "incorrect procedure",
    "date": "2025-07-01T00:00:00",
    "status": "denied"
  }
]
```

Required Steps

Step 1: Schema Normalization

Transform each record into the following unified schema:

```
json
CopyEdit
{
  "claim_id": "string",
  "patient_id": "string",
  "procedure_code": "string",
  "denial_reason": "string or null",
  "status": "approved/denied",
  "submitted_at": "ISO date",
  "source_system": "alpha or beta"
}
```

Notes:

- Normalize dates.
- Null values must be handled explicitly.
- Ensure consistent casing and formatting.
- Add a `source_system` field based on file origin.

Step 2: Resubmission Eligibility Logic

A claim should be **flagged for resubmission** if **all** the following are true:

1. **Status is `denied`**
2. **Patient ID is not null**
3. The claim was submitted **more than 7 days ago** (assume today is `2025-07-30`)
4. The **denial reason** is either:
   - A known retryable reason (see below),
   - OR inferred as retryable via LLM/heuristic classifier if ambiguous

Retryable Denial Reasons: "Missing modifier"**;** "Incorrect NPI"**;** "Prior auth required"
Known Non-Retryable: "Authorization expired"**;** "Incorrect provider type"
Ambiguous Examples (LLM Classification or Hardcoded Mapping): "incorrect procedure"**;** "form incomplete"**;** "not billable"**;** null

Step 3: Output

Produce a list of claims eligible for **automated resubmission**, including:

```
json
CopyEdit
{
  "claim_id": "A124",
  "resubmission_reason": "Incorrect NPI",
  "source_system": "alpha",
  "recommended_changes": "Review NPI number and resubmit"
}
```

You can use a **mocked or hardcoded classifier** for ambiguous denial reasons if needed.

Final Deliverables

1. **Working script** (or Jupyter notebook or pipeline script) that performs all steps.
2. Print or save output to `resubmission_candidates.json`
3. Basic **logging or metrics**, e.g.: Total claims processed; How many from each source; How many flagged for resubmission; How many excluded (and why)
4. Handle malformed or missing data gracefully.

Bonus Stretch Goals (Optional but impressive):

1. **Modularize the pipeline** using functions or classes.
2. Add a **FastAPI endpoint** to upload new datasets and return resubmission candidates.
3. Simulate a **Dagster or Prefect pipeline** for orchestration.
4. Mock an LLM **"classifier" function** for ambiguous denial reason inference.
5. Export failed records to a **separate rejection log file**.

Evaluation Criteria

- Data wrangling and schema mapping skills
- Clear, testable logic for eligibility
- Robust handling of inconsistent data
- Modular and maintainable code
- Communication (comments, structure, logging)
- Ability to handle ambiguity and incomplete input

Case Study #2: Alternate General Challenge: Cross-Platform Action Agent Using LLM + Browser Automation

Scenario:
Modern web-based services have highly variable UIs — but users often want to **automate similar workflows** across them (e.g., sending an email, uploading a document, submitting a form). You're designing a prototype system that can perform a **single generic user goal** (like sending an email) across **multiple service providers** with different UI layouts.
You can use tools such as:

- **LLM Agents** for instruction following / reasoning (mocked or real)
- **Browser automation tools** such as `nano-browser`, `playwright`, `selenium`, `puppeteer`, or a custom wrapper
- (Optional) RAG, DOM-tree parsing, or screenshot-to-instruction bridging

Challenge Objective:
Design and implement a **simple agent-enabled automation** that can execute a **single goal-oriented task** across **multiple web UIs**.
Task Options (choose one or propose your own):

1. **Send an email** using both **Gmail Web** and **Outlook Web**.
2. **Upload a document** to both **Google Drive** and **Dropbox**.
3. **Schedule a meeting** via both **Google Calendar** and **Outlook Calendar**.
4. **Submit a support ticket** on two different platforms (e.g., Zendesk and Freshdesk).
5. **Post a message** on two different forums or communities (e.g., Reddit and Discourse).

Required Features:

1. Accept a **natural language instruction** (e.g., "Send an email to Alice about the meeting at 2pm").
2. Use an **LLM or mocked reasoning function** to:
   - Interpret the instruction
   - Identify form fields or buttons in the target UI
   - Generate DOM interaction steps
3. Use **browser automation** to:
   - Launch the browser
   - Navigate to the service (Gmail, Outlook, etc.)
   - Authenticate (mock or real)
   - Execute the UI actions
4. Support **at least two providers** with differing DOM structures.
5. Abstract the provider-specific logic behind a **unified interface**.

Stretch Goals (Optional but encouraged):

- Automatically recover from minor DOM structure changes using heuristics or vision-based methods
- Use **DOM-tree + LLM prompt chaining** to match fields dynamically
- Allow the agent to analyze **screenshots** or **HTML** to reason about UI
- Log each step (e.g., "Clicked Compose", "Filled subject field")
- Build a **"Generic UI Agent"** class that works across services with minimal per-provider config
- Add a CLI or FastAPI endpoint that takes commands like:

```bash
CopyEdit
python agent.py "send email to joe@example.com saying 'Hello from my automation system'"
```

Evaluation Criteria:

- Agent architecture and generalization ability
- DOM parsing and automation reliability
- Use of LLM for UI intent inference
- Abstraction and modularity of per-provider logic
- Logging and recoverability
- Thoughtfulness in handling edge cases (e.g., auth, field not found)

Inspiration / Framing Hints:
You can imagine this as building a **"universal web action agent"** — a system that could plug into multiple frontends and understand how to complete a user task even across unfamiliar interfaces.
If building the real agent is too complex for the timebox, candidates can **focus on a mocked or conceptual implementation** that shows clear design reasoning and basic browser interaction.