vibeiq • Senior Platform Engineer

Technical Assignment

VibeIQ Senior Backend Engineer Technical Challenge

Objective: Build a Scalable CSV-Based Item Loader

Design and implement an **ETL** service that processes a CSV file and upserts items into an existing Item Service using HTTP APIs. The service must transform CSV data into valid item objects, resolve roles, deduplicate and create families if needed, and safely interact with a Lambda-backed API.

To guide your design thinking, assume the service may need to handle:

- Up to 5 million item ingestions per day
- Peak throughput of 5,000 items per minute
- Maximum acceptable processing latency of 10 minutes per CSV

You don't need to implement a distributed system, but you should account for batching, rate limits, and API performance in your design decisions and documentation.

CSV Input Format

Each row in the CSV represents either a **family** or an **option**, with the following fields:

- `familyFederatedId` (string, required)
- `optionFederatedId` (string, optional)
- `title` (string)
- `details` (string)

An item family represents a product (e.g. Running Shoe). An item option represents a variant of that product that a customer can actually purchase (e.g. Blue Running Shoe). A family has many options.

Target Item Schema (Output Format)

Each item produced from the CSV should match this structure:

- `id` (string) - internal system ID (looked up via API; not provided in CSV)
- `name` (string) - mapped from `title`
- `description` (string) - mapped from `details`
- `federatedId` (string):
    - `optionFederatedId` if present
    - Otherwise, `familyFederatedId`

- `roles` (array of strings):
    - `"option"` if `optionFederatedId` is present
    - Otherwise, `"family"`

Loader Responsibilities

1. Parse the CSV and transform each row to match the target schema

2. Determine `role` and `federatedId` for each row:
    - If `optionFederatedId` is present → `"option"`, `federatedId = optionFederatedId`
    - If not → `"family"`, `federatedId` = `familyFederatedId`

3. Lookup existing items via federatedId using:
    - `GET /items/byFederatedIds?federatedIds=...`

4. Create or update items using:
    - `POST /items/batch` or `PUT /items/batch` (100 max per batch)

5. Ensure families exist:
    - If no item with the `familyFederatedId` exists and it's not present in another row as a `"family"`, create a `"family"` item for it with:
        - `federatedId` = `familyFederatedId`
        - `roles` = `["family"]`
        - A default name and description if one is not provided

---

Interface

Provide a clear and expressive REST API interface to invoke the loader.

Platform Constraints

- The Item Service runs on AWS Lambda
- There is a Lambda concurrency limit of 1000

Your loader should prevent overwhelming the downstream API.

Item Service APIs

You may assume access to the following:

- `GET /items/byFederatedIds?federatedIds=...`
- `POST /items` & `POST /items/batch`
- `PUT /items/:id` & `PUT /items/batch`

Deliverables

1. Design document explaining:
    - Architecture
    - Data transformation
    - Assumptions you made throughout the development process and/or questions you would raise to stakeholders to clarify ambiguity
    - Batching and concurrency strategy
2. GitHub repo with source code. Preferably in Typescript.
3. README with setup and run instructions.

Timebox

Please spend no more than **4-5 hours**. Prioritize architecture and clarity over feature completeness.