📄 Homework: Money Transfer Lifecycle Tracker

Timebox: 3 hours
Expectation: Partial solutions are expected. We care more about engineering judgment and reasoning than completeness or polish.

Background

You are building an **internal tool** to help engineers and support staff understand what is happening to **money transfers** in the system.

Transfers are initiated by your system, but **status updates arrive asynchronously** from downstream systems. These updates are not guaranteed to arrive once, in order, or at all.

Transfer Events

Downstream systems send events like:

```
{
  "transfer_id": "tr_123",
  "event_id": "evt_1",
  "status": "initiated",
  "timestamp": "2024-01-05T12:00:00Z"
}
```

```
{
  "transfer_id": "tr_123",
  "event_id": "evt_3",
  "status": "processing",
  "timestamp": "2024-01-05T12:01:30Z"
}
```

```
{
  "transfer_id": "tr_123",
  "event_id": "evt_2",
  "status": "settled",
  "timestamp": "2024-01-05T12:03:00Z"
}
```

```
{
  "transfer_id": "tr_123",
  "event_id": "evt_4",
  "status": "failed",
  "reason": "insufficient_funds",
  "timestamp": "2024-01-05T12:02:00Z"
}
```

Valid statuses

- `initiated`
- `processing`
- `settled` (terminal)
- `failed` (terminal)

Constraints

- Events may arrive **out of order**
- The same event may be sent **multiple times**
- Some events may **never arrive**
- `transfer_id` is correct and stable

Your Task

Build a small system that allows internal users and services to **understand what happened to a transfer and what state it is in right now**. It should include:

- a backend for handling events
- an API for transfer state
- a simple UI

These components are described further in the next sections.

Backend Requirements

Your backend should:

- Accept transfer events via an API
- Handle **duplicate events** safely (idempotency)
- Store enough information to support debugging and investigation
- Determine the **current status of each transfer** based on *all events received so far*, even if they arrived out of order
- Detect and surface **suspicious or inconsistent situations**, such as:
    - A transfer marked `failed` after being `settled`
    - Conflicting terminal states
    - Missing expected transitions

Second Consumer

There is an internal service, **Transfer Orchestrator** (which you do not own/need to implement), that triggers downstream workflows based on transfer statuses. For example:

- When a transfer becomes `settled`, it sends a "success" notification.
- When a transfer becomes `failed`, it schedules a retry (or alerts a human).
- It polls/queries transfer status frequently and should not need to parse raw events.
- **Do not** implement the orchestrator. You only need to supply an API for the orchestrator to call, so that it can be decoupled from transfer event handling.

Requirement: Provide an API that the Transfer Orchestrator can call to retrieve transfer status.

Minimum API expectation (you can choose REST or GraphQL):

- Fetch the status of a single transfer by ID (e.g., `GET /transfers/:id`)
- List transfers with status and "warning" indicators (useful for both UI and automation)

You may choose the language, framework, and storage.
An in-memory store is acceptable.

Internal UI Requirements

Build a simple internal-facing UI that:

Transfer list

Displays:

- Transfer ID
- Current status
- Whether the transfer is terminal
- Last update time
- A warning indicator if something looks suspicious

Transfer detail view

- Ordered timeline of all events for the transfer
- Current status and any detected issues
- Enough context for a human to understand what happened

No authentication or visual polish is required.

Design Notes (Short Write-Up / Video Explanation)

Briefly explain:

- How you handle duplicate and out-of-order events
- How you determine the current transfer status
- How you designed the API for the second consumer (contract, payload shape, tradeoffs)
- Key tradeoffs you made or considered

A README or short document is sufficient.

System Design Questions (Text Only)

Please answer the following in your write-up / video. Short, thoughtful answers are sufficient. No code is needed.

1. Scaling & Evolution

If this system needed to handle **millions of transfers per day**, what would you change first?
What parts of your current design would you want to keep?

2. Data Correction & Recovery

Imagine a bug caused incorrect transfer statuses to be shown for several hours, but the underlying events are correct.

How would you:

- Detect the issue?
- Correct existing data safely?
- Reduce the chance of this happening again?

3. Correctness vs Freshness

Some consumers care about **near-real-time updates**, while others care more about **correctness and stability**.

How would your design balance these needs?
What tradeoffs would change if real money safety were critical?

Optional (Not Required)

If you have extra time, you may choose to implement **one** of:

- A way to manually mark a transfer as resolved (with a note)
- A way to recompute a transfer's current status from its event history
- A simple rule to flag "stuck" transfers

These are optional and not required for a positive evaluation.

What We're Looking For

- Clear modeling of events and transfer status
- Thoughtful handling of real-world failure modes
- Reasonable scope and tradeoffs
- A UI that helps someone actually debug an issue
- Clear communication of decisions
- Note: you are welcome to use AI tools to speed up your implementation.

Submission Instructions

Please send:

1. **Code** as either:
    - a link to a GitHub repo (preferred), **or**
    - a zipped folder of the project
2. A short **README** including:
    - how to run backend + frontend
    - any assumptions / shortcuts
    - answers to the System Design questions mentioned above
3. *(Optional but helpful)* a **2–5 minute walkthrough video** (Loom or similar) showing:
    - transfer list + detail view
    - an example of out-of-order / duplicate events
    - how the second consumer would call the status API