Question 1 – Architecture for a Netflix Originals calendar

I’ve worked on near-real-time (NRT) systems like this in the past where it is essentially a “distributed database” problem with data coming in from multiple places all in different formats.

**Assumptions:**

* Upper bound on active movie\_ids: 600 current shows -> 2x magnitude 60,000
* Calendar UI is a web GUI
* Need to support year/month/day views
* Calendar entries are discreet dates (no multi-date events spanning days/ months/years)
* All historical data will be retained
* Data received from µSvc will only be filtered on is\_netflix\_original Boolean
* The calendar UI needs to support an upper bound of 10,000 users (current Netflix headcount of 3,500 \* ~3 since not all current employees will use the UI, but external users will likely also exist)
* µSvcs are JSON-over-HTTP(S)
* Ω µSvc uses auto-incrementing integer as movie\_id value

**Startup conditions:**

* Data will need to be loaded from external systems

**Backend Implementation:**

* PostgreSQL for datastore
  + Calendar table with forecast\_launch\_date, movie\_id, is\_netflix\_original
  + Events table for each µSvc from which data is collected
* If no webhooks, system is reduced to polling which doesn’t scale at all, since for each of the 60,000 potential shows Calendar would need to constantly be making a request to each µSvc to check for updates; not feasible
* The way I have done this in the past is to have each µSvc use sequence numbers (incremented when data is updated) and send out notifications via webhooks.
  + Each time a µSvc resource is inserted/updated, update the sequence number for the resource’s datastore (even if it is a soft-delete) and then log the change to an “updates” table (using a DB trigger) and fire notifications to webhook subscribers.
  + The initial startup conditions are relatively straightforward here: subscribe to the webhook, then start pulling resources by movie\_id from 1 and incrementing up until HTTP 404; then discard all received webhook notifications w/ a seq\_num < max fetched seq\_num.
  + The webhook interface to each µSvc would ideally include a heartbeat option that would include current\_time and most recent seq\_num, or this could be provided as a REST endpoint that would use ETag headers w/ HTTP HEAD requests for lightweight checking of the service.
  + In addition, the µSvc would have a /seq\_num/:id route for fetching missed seq\_num data should a webhook notification be missed for some reason.
* As data is received on our webhook listener, it gets put into the appropriate table for that µSvc, and on insert/update it would run some logic to combine this new data with existing data from other µSvc to produce the desired data for our UI.

**Frontend Support:**

* As data is received, a background daemon could periodically write calendar entries to static JSON files in a path such as /calendar/YYYY, /calendar/YYYY/month/MM, /calendar/YYYY/week/WW for all year/month/weeks found in pg for fast and easy service over HTTP
* Static data service is simple and scales OK, but lacks the eye-candy of real-time updates. To facilitate that, I have used REST handlers that fire internal events over a message bus (ZeroMQ), and those messages are then bridged through to websocket listeners (IIRC the bridge used eventmachine) to facilitate pushing updates out effectively in real time to connected web GUI clients.