Design and development of a web-based system facilitating the ingestion, storage and query of weather-related data produced by IoT sensor devices.

System Description

1. Every 10 seconds, IoT devices installed in meteorological stations in each city across the county measure properties such as temperature, humidity, and wind, and stream them to the backend.
2. Every morning, each meteorological station in each city collects all the data from the last 24 hours and sends them to the backend as a batch
3. A user uses daily the UI and submits a form with the weather forecast for tomorrow for each city regarding temperature, humidity, and wind
4. The backend should be able to receive all data and store them. It should also be able to provide the data to the UI upon request.

Methodology

Over 3 iterations we produced 3 architectural proposals mostly along the scalability characteristics of the end system.

We then implemented part of the simplest one as per requestor's ask.

1st Iteration: Minimum Viable Product

For MVP (or PoC) we'd need to support the basic functionality asked without addressing the scalability issues.

At minimum we need to have a web service able to listen for HTTP requests supporting the asked functionality.

To this end we proposed a web service with next REST endpoints (specification 1, 3):

- `/station supporting POSTs, PUTs, DELETEs so that the user can (through UI) add, edit and delete a meteorological station.`
- `/sensor supporting POSTs so that the user can add sensors (representing the actual IoT data sources)`
- `/ingest supporting POSTs where sensors will send their priodic (every 10 sec) measurements`
- `/forecast supporting POSTs and GETs where user can add and retrieve forecasts for various cities`
- `/weather supporting GETs where user can get the actual recorded weather (average value of all the recorded measurements for temperature, humidity, wind)`

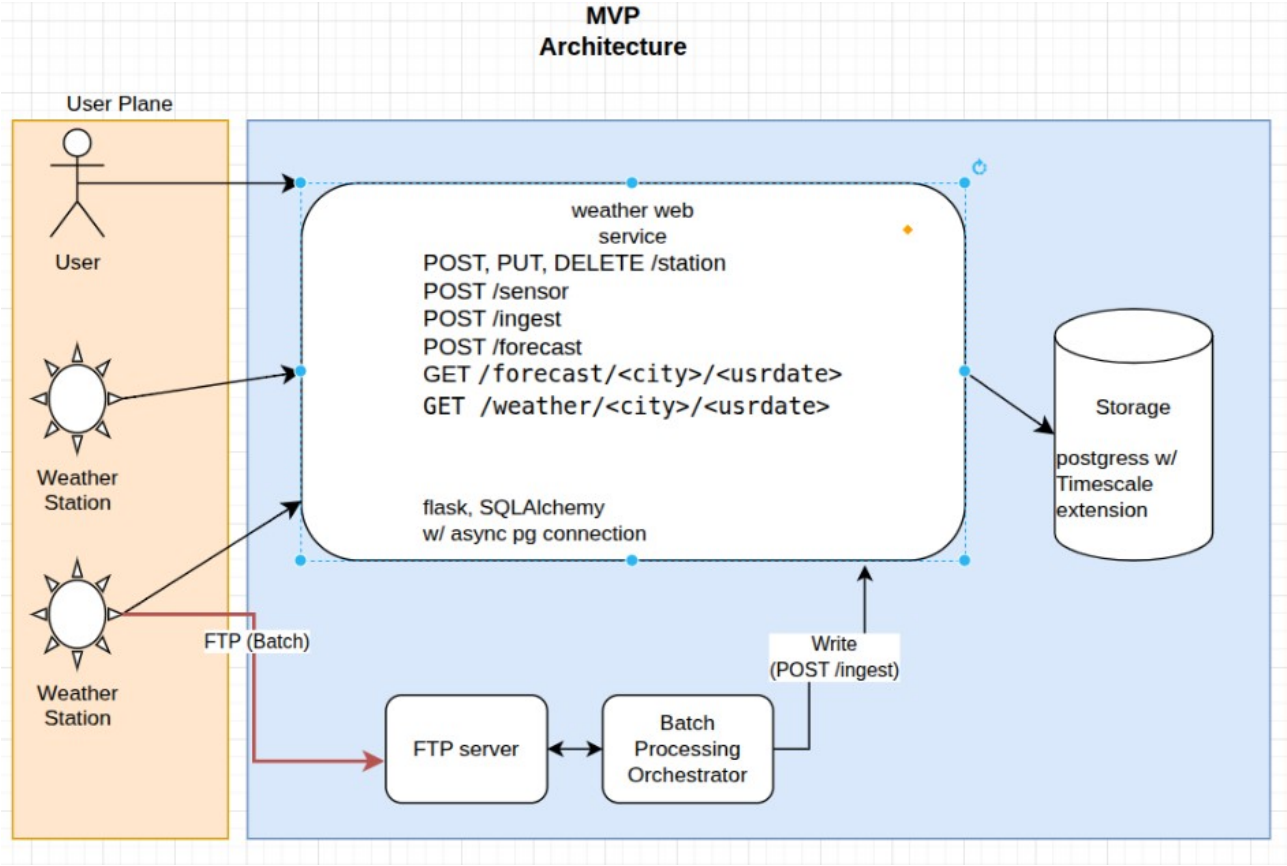
For batches (specification 2) we opted for an FTP server where the IoT devices can upload their batches every morning. A service (Batch Processing Orchestrator) will then read through the files and ping the measurements to /ingest endpoint at a low rate so as not to flood our system.

The batch measurements will for the most part be already known to the system so most of them will be rejected by the DB (see next section).
The measurements will be persisted in a Database. We thought about resorting to some sort of NoSQL DB as the data are essentially timeseries and some NoSQL Dbs are specifically geared towards such use cases. Yet it is evident from specs that the data are inter related and we'd have to use these relations to support queries later on (e.g. average all measurements for a specific city).

After some research we saw that a classic and tested Relational DB product, namely Postgresql, does have an extension (Timescale) specifically geared towards supporting timeseries data efficiently (by partioning the data on the time column). Being a relational DB we could also have full SQL support. Thus we opted for that specific implementation.

Implementation of MVP

We implemented this system using Python flask framework with SQLAlchemy ORM.
To ensure minimum latency we made all the DB transactions async.
The DB, as described above, was Postgresql with Timescale extension.



2nd Iteration: Increased scalability

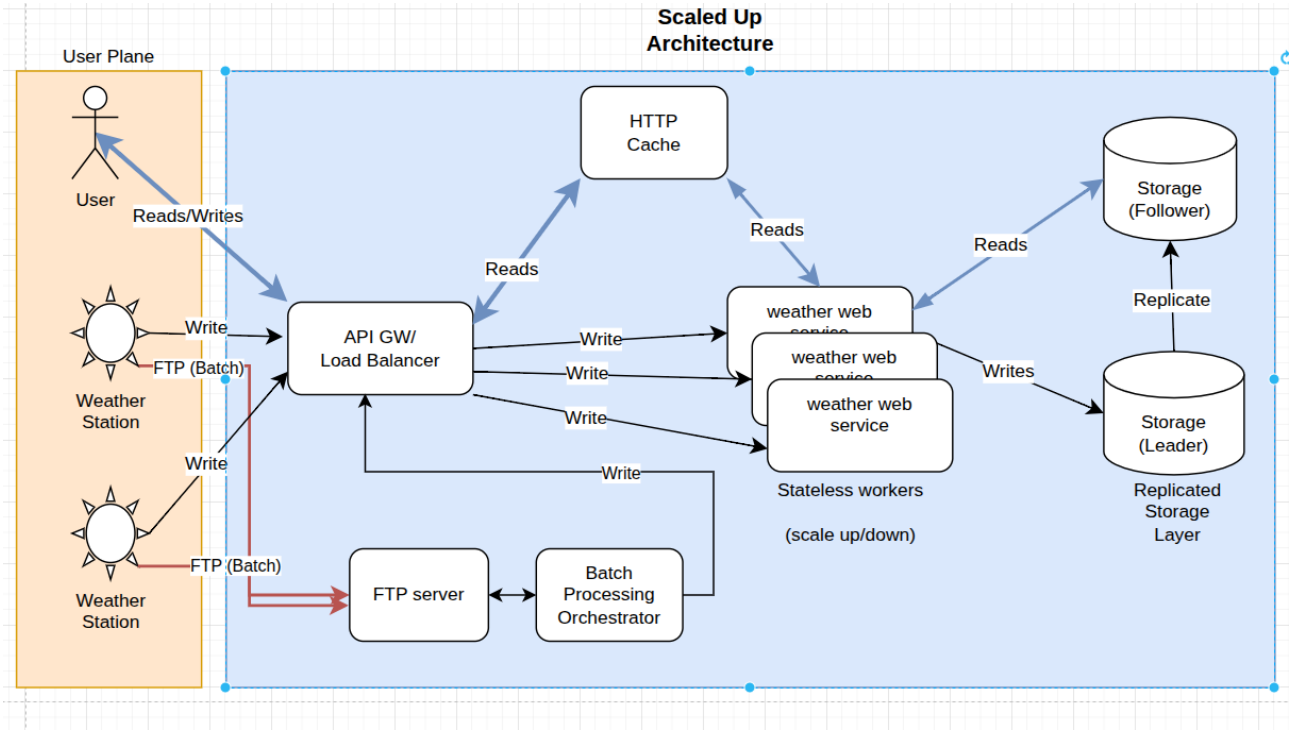
To increase scalability we propose fronting the web service with a Load Balancer. (We might as well pack an API gateway with it as it will help with caching).

LB will be able to disperse write requests (/ingest endpoint) to a pool of web services that can be scaled horizontaly. Note that to enable that we must ensure that the web services remain stateless.

Reads can be directed by API GW to a middleware HTTP cache service. In case of cache misses the service will just forward the request to a web service (/weather endpoint) which will send the query to the DB.

Batch Processing Orchestrator will hit the /ingest endpoints keeping a low rate as before.

At the DB layer, for high availability and decreased latency, we can replicate the DB in a Leader/Follower fashion so as to separate the reads (done from the follower) from the writes (on Leader).



3rd Iteration

Aiming further we would propose an event driven architecture where essesntially all writes are published in a message broker. Afterwards, in a fully asynchronous fashion, they are consumed by the web service (which actually now is just a backend worker service).