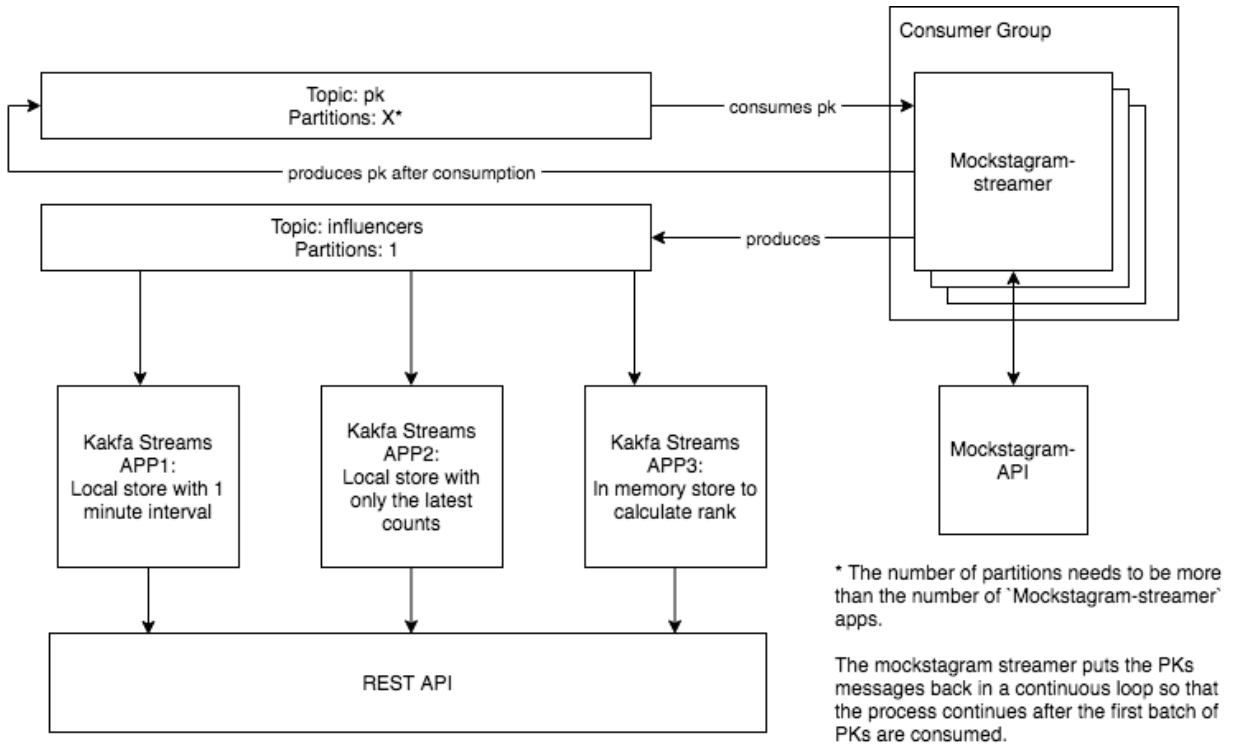
Architecture Diagram



Design Choices

1. **Data ingestion - Apache Kafka**
   One instance of the consumer is not enough to meet the 16k tps threshold to make sure that the updates intervals are sub minute. Horizontal scaling can be achieved by having a queue shared across instances that can spread and allocate work evenly.

   In the submission, I created 2 partitions and seeded the topic with a million :pk that will be consumed by an application that will call the mockstagram-api for the influencer data. When a :pk has been consumed, an API call will be sent and the :pk will be added back to the topic.

   The number of partitions can be increased if more instances are spun up. Kafka handles the work allocation by spreading it among the members in the consumer group.

2. **Data processing - Apache Kafka**

Since data is coming in all the time, we need a continuously updated view of the state. Kafka provides an aggregation and materialisation API for working with streams that will update as data comes in. It caches locally and we can easily query for results without re-processing the entire stream from the start.

For some more complicated queries, I manually processed the stream messages from the beginning and stored the result on memory. I have not figured out how to use the Streams API for ranking as every message that is consumed will create an update to the entire table which doesn't sound efficient.

I have also chosen to ignore `followerRatio` as that can be derived from the `followerCount` and `followingCount`. The client requesting the data can work to calculate the result if required.

Limitations

1. **Single application**
   To keep things simple, I merged all the parts into a single Spring Boot application. The components mainly communicate with each other through Kafka and they can be separated into individual apps.

2. **Processing speed**
   I'm using AsyncHttpClient for the requests. I have tried implementing using WorkerPools and Akka-based library parallec with no significant improvement. It is currently running at ~3k calls per second. The docker container I'm using also has a max connection limitation that stops us from hitting more aggressively.

3. **Averaging**
   Currently implementation is really slow as it averages all the latests stats from a stream. Can speed up by keeping the records in memory and calculating the average when requested.

4. **Surviving crashes**
   Currently everytime we consume a :pk, we add it back to the topic. However, crashes to the application might interrupt this process and we might lose a :pk in the mess. What could be done is to commit the offset on kafka only after it has been published again.