Mockstagram Influencer Data Platform Design Document

1. Introduction & Overview

This document outlines the design for a distributed, streaming system to extract influencer data from the Mockstagram API. The system will capture each influencer's follower count once per minute, process this data, and store it in a database to power a dashboard showcasing influencer statistics, including current follower count, follower count timeline, and average follower count.

2. Core Requirements Addressed

a. **Data Granularity**: Capture follower count data with a granularity of one minute for each influencer.

b. **Average Follower Count**: Precompute the average follower count by maintaining a running *total_sum_of_follower* and *readings_count* for each influencer. This average should be updated with each new follower count.

c. **Scalability**: The system must scale horizontally. If N nodes can handle X influencers, 2N nodes should handle 2X influencers. This applies to both processing services and database choices.

d. **Load Distribution**: The load (influencer PKs to query) must be distributed evenly among processing nodes without hardcoding influencer lists to specific nodes. Kafka rebalancing addresses this.

e. **Streaming Solution**: A streaming architecture is preferred over batch processing.

f. **Data Storage**: Provide a database solution that allows for efficient querying of an influencer's current stats, timeline, and average**.**

3. Architecture Overview

The system employs a message queue-based architecture using Apache Kafka to decouple services and enable scalable, asynchronous processing. Services are designed to be stateless (except Bootstrap and Scheduler), allowing for flexible scaling and resilience.

The main components are:

- **Kafka Topics**: For managing tasks, raw data, and configurations.
- **Microservices**: Dedicated services for distinct tasks like scheduling, data fetching, data processing, and database interaction.
- **Database:** For storing the influencer data, system logs etc.
- **Observability:** For monitoring service and all infrastructure components.

4. **System Components - Kafka Topics**
    a. Registry Topic (active_influencers): Maintains the active list of influencer primary keys (PKs) that the system needs to track, allowing for dynamic updates to the list of influencers. It is configured with Kafka's log compaction feature, ensuring that only the latest status (e.g., active/inactive) for each PK is retained and providing a "last write wins" semantic for each influencer's registration status. Messages contain the influencer PK and any relevant metadata about their status.
    b. Influencer Fetch Tasks Topic (influencer_fetch_tasks): This single, persistent topic is used to distribute the task of fetching data for all active influencers every minute. Every minute, the Scheduler Service publishes messages to this topic, one for each active influencer PK, with each message formatted as:
    { "pk": <influencer_pk>, "target_minute_timestamp": "YYYY-MM-DDTHH:MM:00Z" }
    c. Result Topic (fetcher_results): Stores the raw API response (follower count, username, etc.) received from the Mockstagram API for each PK before it's processed and inserted into the database. After a successful API call data is published to this topic with messages such as:
    { "pk": <pk>, "username": <username>, "followerCount": <count>, "fetchTimestamp": <timestamp> }
    d. Dead Letter Queue (DLQ) (fetcher_dlq / result_handler_dlq): Collects any messages from the other topics that cannot be processed successfully by consumer services (after retries, if applicable). This allows for later inspection, debugging, and potential manual or automated re-processing of failed messages.

5. **System Components – Services**
    a. Bootstrap Service: This service populates the Registry Topic with the initial list of influencers (and, if the influencer list is dynamic, handles ongoing updates to the registry). It queries a definitive source for influencer PKs and publishes them to the Registry Topic.
    b. Scheduler Service: This service orchestrates the per-minute fetching cycle by consuming the Registry Topic to maintain an in-memory snapshot of all active influencer PKs. Every minute, it iterates through those PKs and publishes a task message for each to the Influencer Fetch Tasks Topic.
    c. Fetch Task Handler Service (API Caller): This service fetches follower count data from the Mockstagram API by consuming messages from the Influencer Fetch Tasks Topic. For each PK, it calls the external Mockstagram API asynchronously using Axios (configured with retry logic), and upon a successful response, publishes the result (including PK, username, follower count, and fetch timestamp) to the Result Topic. If API errors occur (after retry attempts), it sends those messages to the DLQ.
    d. Result Topic Handler Service (Data Processor & Storage): This service processes the raw data from API calls and persists it to the database. It consumes messages from the Result Topic and for each message, reads the current total_sum_of_follower and

data_point_count for that influencer from the database, calculates the new sum and count, updates or inserts the influencer's summary data (current follower count, new sum, new count, last updated timestamp), and inserts the new follower count and fetch timestamp into the historical timeline table. All of these database operations are executed atomically and in batches to ensure consistency and performance.

e. DLQ Handler Service: This service manages messages in the Dead Letter Queue(s) by providing mechanisms for logging, alerting, manual inspection, and (if needed) automated retry or discard strategies for any failed messages.
f. Dashboard Service: Exposes the influencer timeline data and aggregated data to external clients like web, mobile etc.

6. **Data Management & Storage**

   a. PostgreSQL with the TimescaleDB extension: This serves as the primary database for influencer data. First, enable the TimescaleDB extension to support time-series tables. Then create an influencer_summary table with columns for pk (primary key), username, current_follower_count, total_follower_sum, readings_count, and last_updated. Next, define a follower_timeline table to capture individual follower-count records over time, linking each row back to the influencer_summary via a foreign key. Convert follower_timeline into a hypertable on the timestamp column to leverage TimescaleDB's performant time-series features. For efficient querying, add an index on (pk, timestamp DESC) to support lookups by influencer and a separate index on (timestamp DESC) for queries over recent time windows. The primary key (influencer pk) is indexed by default.
   Example SQL:

```sql
-- Enable TimescaleDB extension
CREATE EXTENSION IF NOT EXISTS timescaledb;

-- Create influencer_summary table
CREATE TABLE IF NOT EXISTS influencer_summary (
    pk BIGINT PRIMARY KEY,
    username VARCHAR(255) NOT NULL,
    current_follower_count BIGINT NOT NULL,
    total_follower_sum BIGINT NOT NULL,
    readings_count BIGINT NOT NULL,
    last_updated TIMESTAMPTZ NOT NULL
);

-- Create follower_timeline hypertable
CREATE TABLE IF NOT EXISTS follower_timeline (
    pk BIGINT NOT NULL,
    timestamp TIMESTAMPTZ NOT NULL,
    follower_count BIGINT NOT NULL,
    CONSTRAINT fk_influencer
        FOREIGN KEY (pk)
        REFERENCES influencer_summary(pk)
        ON DELETE CASCADE
);

-- Convert follower_timeline to a hypertable
SELECT create_hypertable('follower_timeline', 'timestamp', if_not_exists =>
TRUE);

-- Create indexes
CREATE INDEX IF NOT EXISTS idx_follower_timeline_pk_timestamp
```

```
    ON follower_timeline (pk, timestamp DESC);

CREATE INDEX IF NOT EXISTS idx_follower_timeline_timestamp
    ON follower_timeline (timestamp DESC);
```

b. Used for aggregating and storing structured logs and distributed tracing data. By indexing JSON-formatted logs and trace spans, Elasticsearch enables fast, scalable search and analysis. Frontends like Kibana can be layered on top to explore logs and visualize trace data, making it easier to diagnose issues and monitor system performance.

7. Component Choices & Rationale

a. **Kafka as the Backbone:** Kafka is chosen because a durable, horizontally scalable queue is required to distribute tasks evenly via consumer-group rebalancing and to maintain a compacted view of active influencers. The influencer_fetch_tasks topic is initially configured with n partitions, allowing up to n Fetcher Service instances to process disjoint subsets of influencers each minute. As the set of active influencers grows, additional partitions can be added, and consumer groups will rebalance automatically without downtime. By keying messages with each influencer's primary key, all records for a single influencer land in the same partition, ensuring timeline order is preserved.

b. **PostgreSQL + TimescaleDB:** A PostgreSQL database extended with TimescaleDB provides both robust relational capabilities and powerful time-series functionality. TimescaleDB delivers ACID guarantees for summary tables (such as influencer_summary), native SQL support for complex queries (for example, window functions to compute rolling averages), built-in retention policies and compression for older data, and seamless integration with the existing Postgres ecosystem. Indexes include idx_follower_timeline(pk, timestamp DESC) for efficient lookups by influencer and idx_follower_timeline(timestamp DESC) for optimized access to recent time-series data.

c. **Microservices in Node.js:** Node.js is employed for its ability to handle I/O-bound, asynchronous workloads. Axios is used for HTTP requests with configurable retry logic (by default, it attempts three retries with exponential backoff before sending the message to the DLQ if failures persist). All services are stateless except for the Scheduler, which maintains an in-memory snapshot of the active registry. If the Scheduler restarts, it re-consumes the compacted active_influencers_registry topic from the beginning (thanks to log compaction), instantly rebuilds its in-memory set, and resumes the per-minute loop without losing any registry state. This approach ensures resilience and seamless autoscaling.

d. **Observability (Prometheus, Grafana, OpenTelemetry, Jaeger + Elasticsearch, Fluentd + Elasticsearch, Kibana):** Prometheus is chosen for its pull-based model and native integration with the Node.js ecosystem. Its pull-based model allows metrics such as CPU usage, memory consumption, application counters, and Kafka consumer lag, to be regularly scraped, with efficient time-series storage driving alerts and ad-hoc queries. Grafana leverages Prometheus to build customizable dashboards and configure alerting. Panels can be assembled for Kafka health (under-replicated partitions, producer/consumer throughput), Fetcher Service latency percentiles, data "freshness lag" etc., without any custom front-end development. OpenTelemetry provides a vendor-neutral tracing API that propagates W3C Trace Context headers end-to-end (from Scheduler → Fetcher → Result Handler → DB). Jaeger serves as the trace storage and visualization backend. For log, trace aggregation and long-term storage, Elasticsearch indexes structured JSON at scale, enabling queries like "find all Fetcher errors where HTTP status = 500 and traceId = X." Kibana sits on top of Elasticsearch to deliver an interactive log exploration interface that simplifies troubleshooting by correlating logs, metrics, and traces in one place. Fluentd standardizes log collection from Node.js applications and forwards buffered JSON entries to Elasticsearch. Its buffering ensures that log data isn't lost, even if Elasticsearch experiences downtime.

8. Scalability & Fault Tolerance

a. **Consumer Group Rebalancing**: Kafka consumer groups automatically distribute workload. When a new Fetcher Service instance comes online, Kafka rebalances partitions across the group. During that brief rebalance window (typically just a few seconds) message consumption pauses. Since each influencer has a one-minute processing window, rebalances are scheduled during off-peak periods, and additions to the influencer roster are planned in advance to avoid disrupting the main fetch cycle.

b. **Horizontal Scaling**: Both the Fetcher and Result Handler microservices are stateless and can be scaled out without extra coordination. The Bootstrap Service only runs at startup, so it doesn't impact runtime scaling. The Scheduler Service does maintain state (the active_influencers list), but if multiple Scheduler instances become necessary, the active_influencers set can be partitioned and each partition assigned to its own Scheduler instance.

c. **Database Scaling**: PostgreSQL running with the TimescaleDB extension can be scaled using standard techniques (for example, read replicas, vertical scaling, or sharding) to handle growing data volumes and query loads.

d. **Fault Tolerance**: Kafka's persistence and replication guarantee that messages aren't lost, and any stateless service can simply be restarted if it fails. Both the Fetcher and Result Handler use dead-letter queues: for example, fetcher_dlq and result_handler_dlq to capture messages that fail processing even after retries. If a fetch attempt fails by default after three retries, the message lands in fetcher_dlq; the DLQ handler can retry within the same one-minute window, providing an extra layer of resilience. Outside that window, messages are dropped to prevent stale data. Throughout, metrics record retries counts and failure rates so engineers can spot and remedy systemic issues or archive persistent errors.

e. **SLA Adherence (1-Minute Data Freshness):** Delivering fresh data every minute relies on the Scheduler Service keeping perfect time and maintaining enough Fetcher Service capacity. If the system starts to lag (measured by missed one-minute deadlines) it indicates under-provisioning or a processing bottleneck. In that case, enhanced monitoring, consumer auto-scaling, capacity planning, and targeted optimizations are employed to stay within the SLA.

9. Observability and Monitoring

Robust monitoring are essential for ensuring system health, performance, SLA compliance, and for diagnosing issues.

a. **Core Monitoring Principles:** Monitoring must encompass every layer of the stack like infrastructure, Kafka, microservices, the database, and the end-to-end pipeline. Focus on metrics that drive action, such as performance indicators, availability signals, and correctness checks. Implement consistent, structured logging across all services, and use distributed tracing (for example, with OpenTelemetry) to visualize request flows. For instance, when the Scheduler publishes a fetch task, it injects traceparent/tracestate headers; the Fetcher service then captures those headers via the OpenTelemetry SDK, ensuring that if a database write fails, the trace spans Scheduler → Fetcher → Result Handler → DB. Proactive alerting should be in place for critical conditions, and dashboards must be created to surface key metrics in real time.

b. **Monitoring Approaches by System Component:**

    i. **Kafka:** Track broker health, topic message rates, consumer lag (especially for influencer_fetch_tasks), producer metrics, and under-replicated partition (URP) counts. Use Kafka Exporter to push these metrics into Prometheus.

    ii. **Node.js Microservices**: Monitor instance health (CPU, memory, event-loop lag), application throughput, error rates, latency/processing times, active instance counts, and Kafka client metrics. Instrument each service with Prometheus's prom-client library to expose metrics over HTTP, generate structured JSON logs using Winston or Pino, and employ OpenTelemetry for distributed tracing, propagating context via Kafka message headers (e.g., W3C Trace Context) and exporting traces to Jaeger (with Elasticsearch as the backend).

   iii. **PostgreSQL + TimescaleDB**: Observe system resource usage, query performance, throughput, connection counts, overall database health, and TimescaleDB-specific metrics. Leverage tools like pg_stat_activity, pg_stat_statements, TimescaleDB's built-in functions, and a Prometheus exporter for Postgres.
   iv. **End-to-End Pipeline & SLA Tracking**: Measure overall data freshness, end-to-end task latency, DLQ size, and system throughput to ensure SLA compliance. Map these metrics to dedicated dashboards such as Kafka Health, Fetcher Service, Result Handler, and System SLA dashboards to simplify at-a-glance assessments.

c. **Implementing Monitoring:** Centralize all logs by forwarding structured JSON from each service into Elasticsearch, then visualize and explore those logs in Kibana. Use Fluentd or Logstash to collect logs reliably (buffering them if Elasticsearch becomes temporarily unavailable), forming an EFK/ELK stack. For metrics, configure Prometheus to scrape and store endpoints exposed by every component, and build dashboards in Grafana for real-time visualization. Finally, set up Alertmanager and integrate with your preferred notification channels so that any critical issues are raised immediately.

10. Dashboard Service & Client UI

This service fetches influencer data to serve frontend clients. For example, a request to
GET /influencers/{pk}/timeline?start=<>&end=<>
returns both time-series data and the influencer's summary. Because the summary is recalculated on each fetch, there's no need for a full table scan. Indexes and the TimescaleDB extension ensure efficient retrieval of timeline data. Additionally, a Redis-backed distributed LRU cache can store frequently accessed results to reduce database pressure.