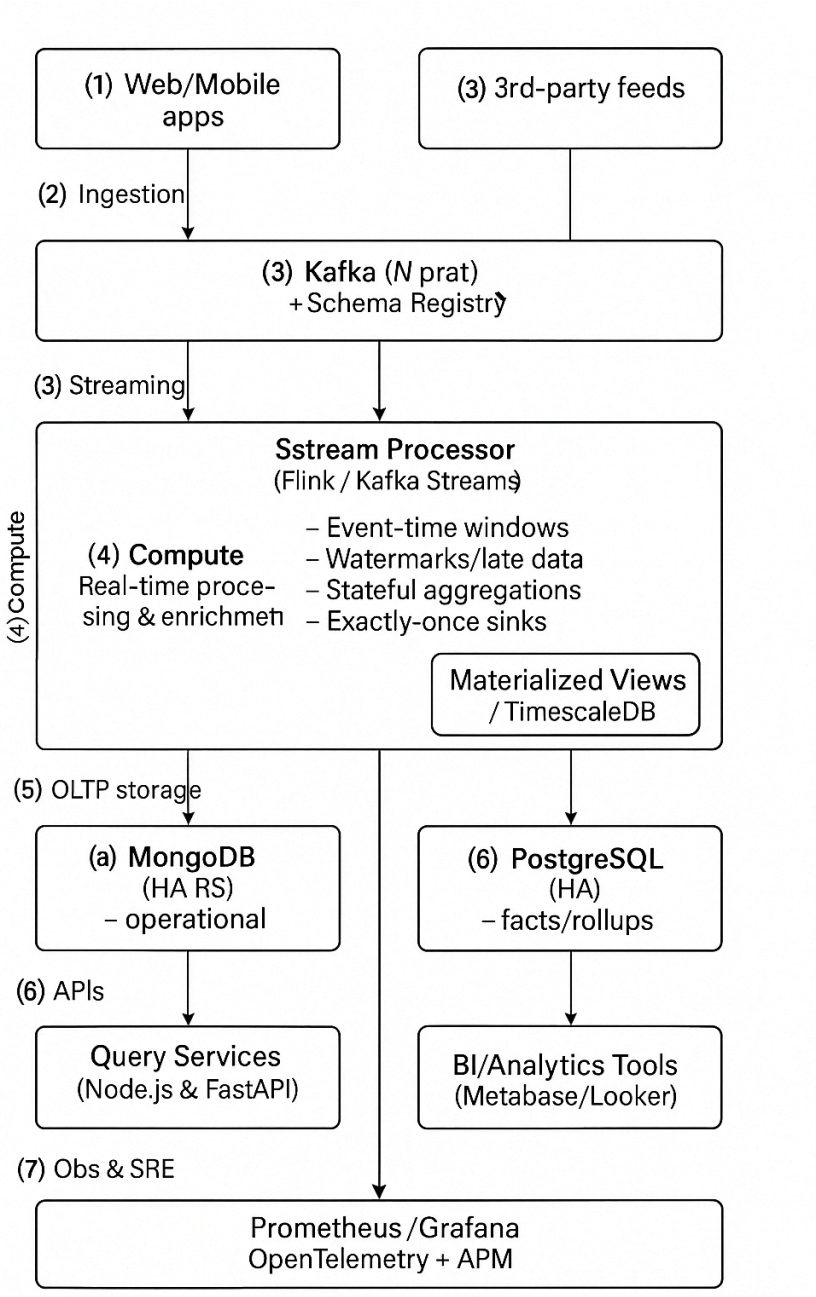
Question 5: System Architecture Design

Problem Statement:

You need to design a scalable architecture for a new feature that involves real-time data processing and analytics. The system will ingest data from multiple sources, process it in real-time, and store the results in both MongoDB and PostgreSQL for different types of querying. The architecture must support high availability, fault tolerance, and scalability. Provide a high-level architecture design with key components and justify your choices.

Solution -



1) Ingestion layer

- **API Gateways + Ingest Microservices (Node.js/FastAPI)** with backpressure and async I/O.
- **Why:** isolates source-specific logic (auth, throttling, validation) and emits clean, versioned events.

2) Durable event backbone

- **Kafka** with **N partitions** per topic, **replication factor ≥ 3**, and **Confluent/Redpanda** compatibility.
- **Schema Registry** (Avro/Protobuf) to enforce contracts and enable evolution.
- **Why:** high-throughput, replayability, consumer groups for horizontal scaling.

3) Real-time compute

- **Apache Flink** (or Kafka Streams if you prefer lighter ops).
- Features you'll use:
  - **Event-time processing**, **watermarks** to handle late/out-of-order data.
  - **Stateful** operators with **RocksDB state backend** (Flink) for large state.
  - **Checkpoints** + **savepoints** for exactly-once and safe upgrades.
- **Why:** precise time semantics, strong state & fault tolerance, exactly-once sinks.

4) Dual-write sinks (exactly-once)

- **MongoDB sink**: upserts for entity/lookup views (e.g., latest state per key).
  - Keys: compound index `{ tenantId, entityId }`, TTL indexes for ephemeral views if needed.
- **PostgreSQL sink**: append-only **fact table** for events + **aggregates/rollups**.
  - Use **TimescaleDB** (or native partitioning) on `time, tenantId`.
  - **Materialized views** (REFRESH CONCURRENTLY) for hot queries; or **continuous aggregates** in TimescaleDB.
- **Why:** operational reads in Mongo (low-latency key lookups), analytical SQL in Postgres (joins, windows).

5) Query services

- **Node.js** for high-QPS operational APIs (Mongo).
- **FastAPI** for analytical endpoints (Postgres), async SQLAlchemy + read replicas.
- **Why:** separation of concerns and independent scaling knobs.

6) Observability & SRE

- **OpenTelemetry** tracing from gateways → Kafka → Flink → sinks → APIs.
- **Prometheus/Grafana** for metrics (lag per partition, processing time, checkpoint duration).
- **Alerting** on consumer lag, error rates, checkpoint failures, disk/CPU saturation