**DesIgnIng PredIctPulse Backend**

Let's design a remote monitoring service like PredictPulse, which is the first cloud-based analytics service for data center infrastructure to predict the failure of power components. Similar Services: Kaiser

# **1. What is PredictPulse?**

The PredictPulse is a cloud-based monitoring and management service that collects and analyzes data from connected power infrastructure devices, providing us with the insight needed to make recommendations and take action on your behalf.

# **2. Requirements and Goals of the System**

Let’s start with building a simpler version of PredictPulse with the following requirements:

**Functional Requirements**

* System need to monitor connected UPS and power devices.
* Each device such as UPS and power devices need to send telemetry to the system.
* Customer get to see all the intuitive reports that are easy to read at-a-glance summarizing what's happened, where and when.
* Customer can get real time status updates and diagnostics.
* System produce alerts and notifications, customer are notified that a ups or power device failure.
* System can do predictive analysis to handle power disruptions faster before they cause downtime.

**Non-functional Requirements**

* Our service needs to be highly available, highly scalable, loosely coupled, proactive, durable and resilient.
* Users should have a real-time status updates and diagnostics with minimum latency.

**Extended Requirements**

1. Real-time dashboards
2. Streaming analytics
3. Advanced security
4. Predictive analytics

# **3. CAPACITY ESTIMATION and CONSTRAINTS**

* Let’s assume we have 10M UPS and power devices.
* Let’s assume that all devices notify their current state every seconds 7X24.
* Once a power device failure happen, the system should be able to contact customer in real-time.

# **4. BASIC SYSTEM DESIGN and ALGORITHM**

Event-driven architectures are particularly well suited to event streams and through this in-stream processing enables businesses to fast decision making, ones where changing business solutions as they happen and make decisions based on all available current and historical data in real-time. Business logic within the application can now be applied to data in motion rather than needing to wait for the data to land somewhere and then do the analysis. This real-time analysis is good for issues like fraud detection, predictive analytics, tackling security threats on the fly, automating supply chains, and so on.

Since all ups and power devices are reporting their state with message every seconds, An individual message which is coming from devices doesn’t make much sense. We need to aggregate and perform certain operations on them to gain more insights. Also we need to keep messages for an extended period. For this reason We need to use event streaming instead enterprise messaging.

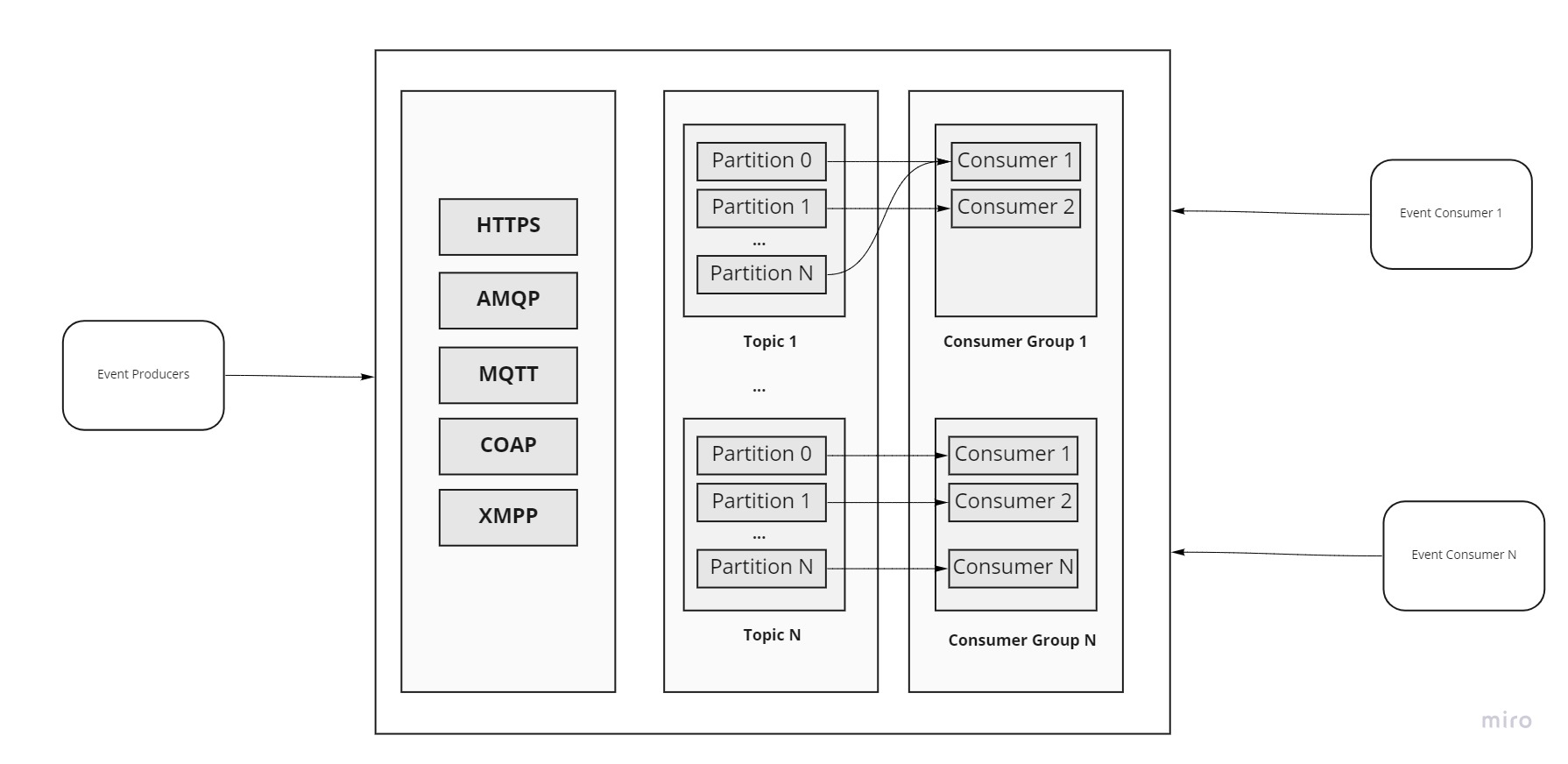
**Do we need to store events accross multiple data centers and geographic regions?** If the answer is yes, we need to use event streming platform.

**What’s the right way to prioritize the data we want to capture in real time, and which data can wait? Some of devices need to prioritized?**

**How much bandwidth will our service consume to receive state updates from all devices?** If we get device name, temperature, battery life, timestamp and their location, it will be (3+4+4+4+16 => 31 bytes). If we receive this information every seconds from 10M devices, we will be getting 310MB per seconds. This will lead us to have 27TB of new telemetry every day.

**How can we efficiently implement Notification service?** We can either use websocket or push notifications.

**How can we communicate with UPS and power devices as a standard way?** The answer is ***o*neM2M** sets the standards for the Internet of Things, providing the industry with the foundations for scalable and interoperable systems.



# **5. Database Schema**

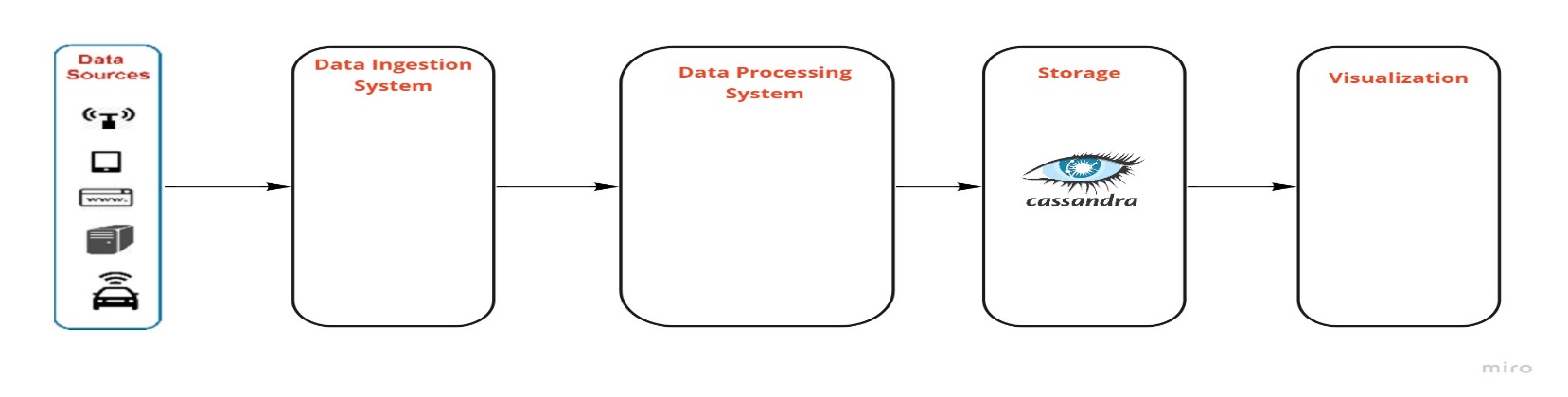
Cassandra is capable of; even with millions of devices that creating data all the time, the database is designed to ingest that much data as it is created.

Cassandra or key-value stores in general, always maintain a certain number of replicas to offer reliability. Also, in such data stores, deletes don’t get applied instantly, data is retained for certain days (to support undeleting) before getting removed from the system permanently.

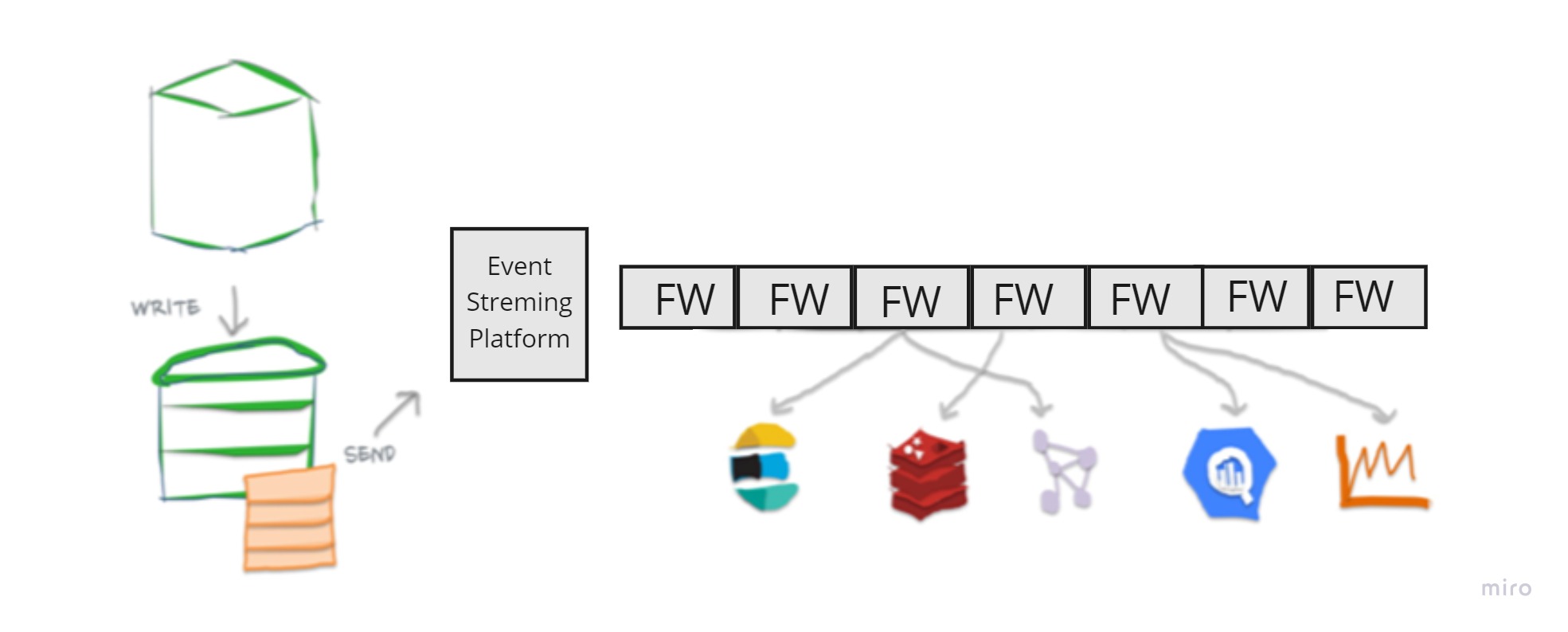
For time-series data, this is especially valuable as it means that there should be no loss of data in the list of transactions over time.

Cassandra is by nature good for heavy write workloads. In combination with Apache Spark and the like, Cassandra can be a strong 'backbone' for real-time analytics. And it scales linearly. So, if you anticipate growth of your real-time data, Cassandra definitely has the utmost advantage here.

There are two ways Cassandra achieves a fast speed: It makes quick decisions on where to store data using a hashing algorithm. It lets any node to make data storage decisions.



If we want to seperate production operations from analytics, Cassandra is best choicefor that: Cassandra introduced a [change data capture (CDC) feature](http://cassandra.apache.org/doc/3.11.3/operating/cdc.html) in 3.0 to expose its commit logs. Commit logs are write-ahead logs in Cassandra designed to provide durability in case of machine crashes. They are typically discarded upon flush. With CDC enabled, they are instead transferred to a local CDC directory upon flush, which is then readable by other processes on the Cassandra node. This allows us to use the same CDC mechanism as in our streaming pipeline. It decouples production operations from analytics, and thus does not require additional work from application engineers.

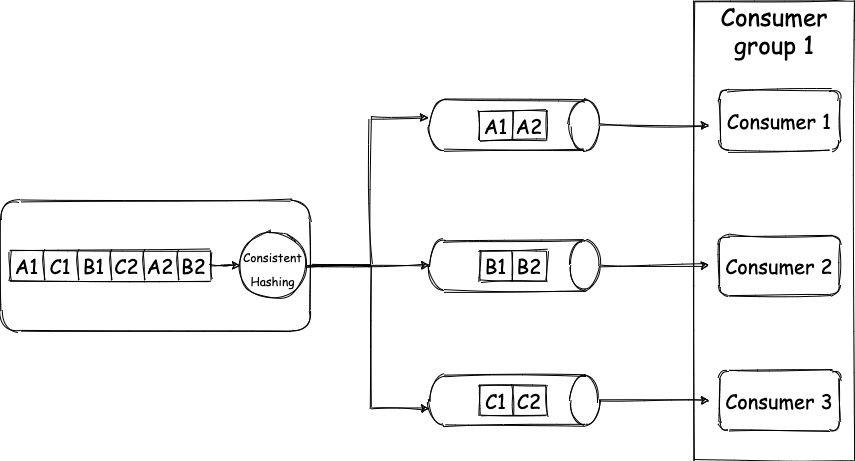


# **6. DATA PARTITIONING**

**Sharding based on topics:** Since no one topic could ever get too big or aspire to accommodate too many reads or writes, that would be just unfortunate. But if we need this option we can divide our topic to partition. This way, the work of storing messages, writing new messages, reading existing messages, reading existing messages, all that kind stuff can be split among nodes. Having broken a topic up into partitions, we need a way of deciding which messages to write to which partition? We need to know which piece a given message goes to. If a message has no key, If that key is null, it’s empty then the messages that we want to write will be distributed round robin among the topic’s partition. If the message does have a key then we use that key to figure out which partition to put message in.

This approach has a couple of issues:

Now we have seen that a topic can be partitioned and multiple consumers can consumer from the same topic, then how do we maintain the order of messages on the consumer-end one might ask. Good question. One partition can’t be read by multiple consumers in the same consumer group. This is enabled by the consumer group only, only one consumer in the group gets to read from a single partition. Let explain.



So our producer produces 6 messages. Each message is a key-value pair, for key “A” value is “1”, for “C” value is “1”, for “B” value is “1”, for “C” value is “2” ….. “B” value is “2”. Our topic has 3 partitions, and due to consistent hashing messages with the same key always go to the same partition, so all the messages with “A” as the key will get grouped together and the same for B and C. Now as each partition has only one consumer, they get messages in order only. So the consumer will receive A1 before A2 and B1 before B2, and thus the order is maintained

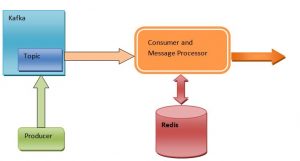
To recover from these situations either we have to use consistent hashing.

# **7. Fault Tolerance and Replication**

**What if a Notification server dies?** We would need replicas of these servers, so that if the primary dies the secondary can take control. Databases are not the obvious choice to capture millions of data items per second. ESP is a distributed system purpose-built to ingest millions of data records per second. This data then stored in fault-tolerant storage to keep them safe from data losses.

# **8. CACHING**

First of all, Event Streaming Platform may only guarantees message ordering within a partition, not across partitions. For example, the ability to partition data by key and one consumer per partition. There are cases where we want to have flexibility, or cases where we cannot follow these design patterns. One of the solutions in these cases is to let the consumer re-order the messages during processing. We will use a caching solution with Redis below.



Redis is a high performance in-memory cache with persistence options. We will use it as a cache and staging area for the messages. The message processor temporarily puts the messages in Redis as needed. When the message processor gets all the related messages it needs based on business or design logic, it re-assembles them in order, processes them and generates the output as it intends to do. Since the messages will only be stored in Redis temporarily and will be removed from the cache after they are re-ordered, the possibility that Redis runs out of memory is remote.

If you use 1 stream -> N consumers, you are load balancing to N consumers, however in that case, messages about the same logical item may be consumed out of order, because a given consumer may process message 3 faster than another consumer is processing message 4.

So basically event streaming partitions are more similar to using N different Redis keys. While Redis consumer groups are a server-side load balancing system of messages from a given stream to N different consumers.

So, the Consumer Groups feature in Redis does not serve to the same purpose of event streaming platform.