

Building Scalable and Extendable Data Pipeline for Call of Duty Games: Lessons Learned



Yaroslav Tkachenko
Senior Data Engineer at Activision

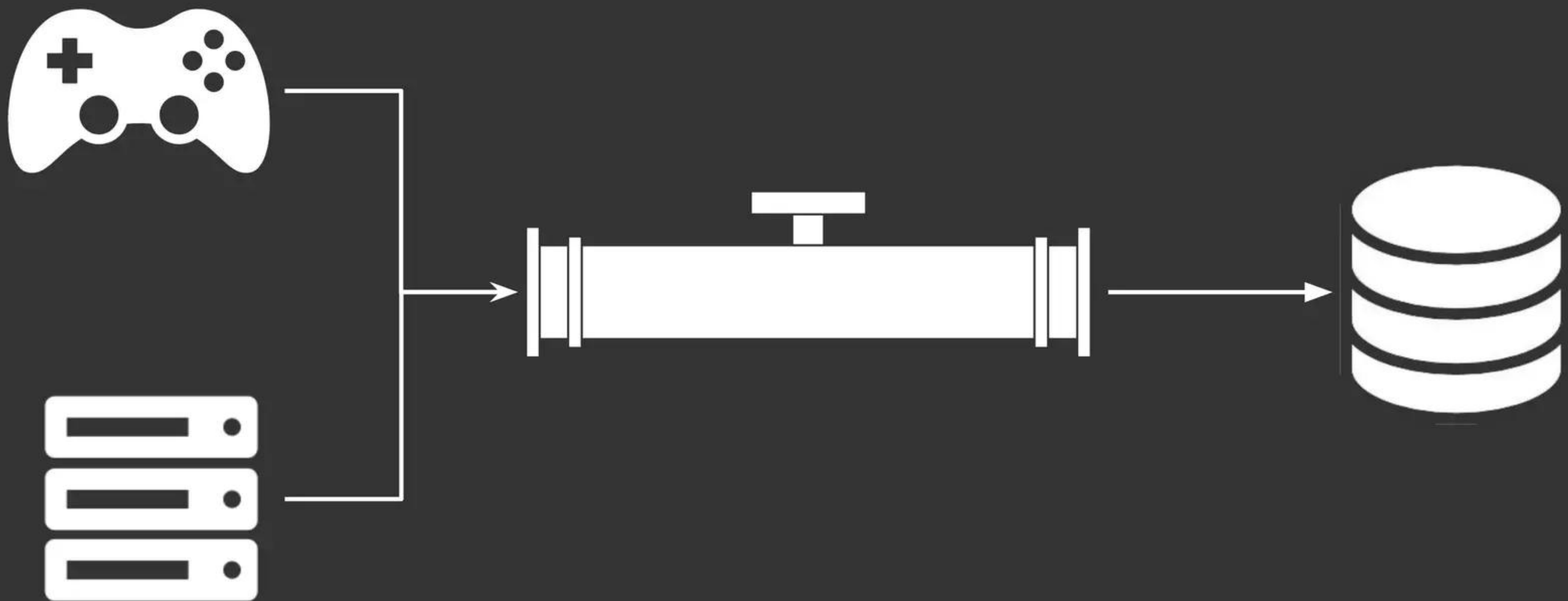


A promotional artwork for the video game Call of Duty: Black Ops III. The image features three soldiers in tactical gear against a dark, smoky background with a large orange and yellow hexagonal graphic. The central figure is a man with a beard and a tattoo on his forehead, holding a submachine gun. To his left is a Black soldier with a headlamp, and to his right is a woman with short blonde hair, also holding a submachine gun. The title 'CALL OF DUTY' is written in large white letters, with 'BLACK OPS' in smaller white letters below it. The Roman numeral 'III' is displayed in orange below the title.

CALL OF DUTY

BLACK OPS

III



1+
PB

Data lake size
(AWS S3)

Number of topics in the
biggest cluster
(Apache Kafka)

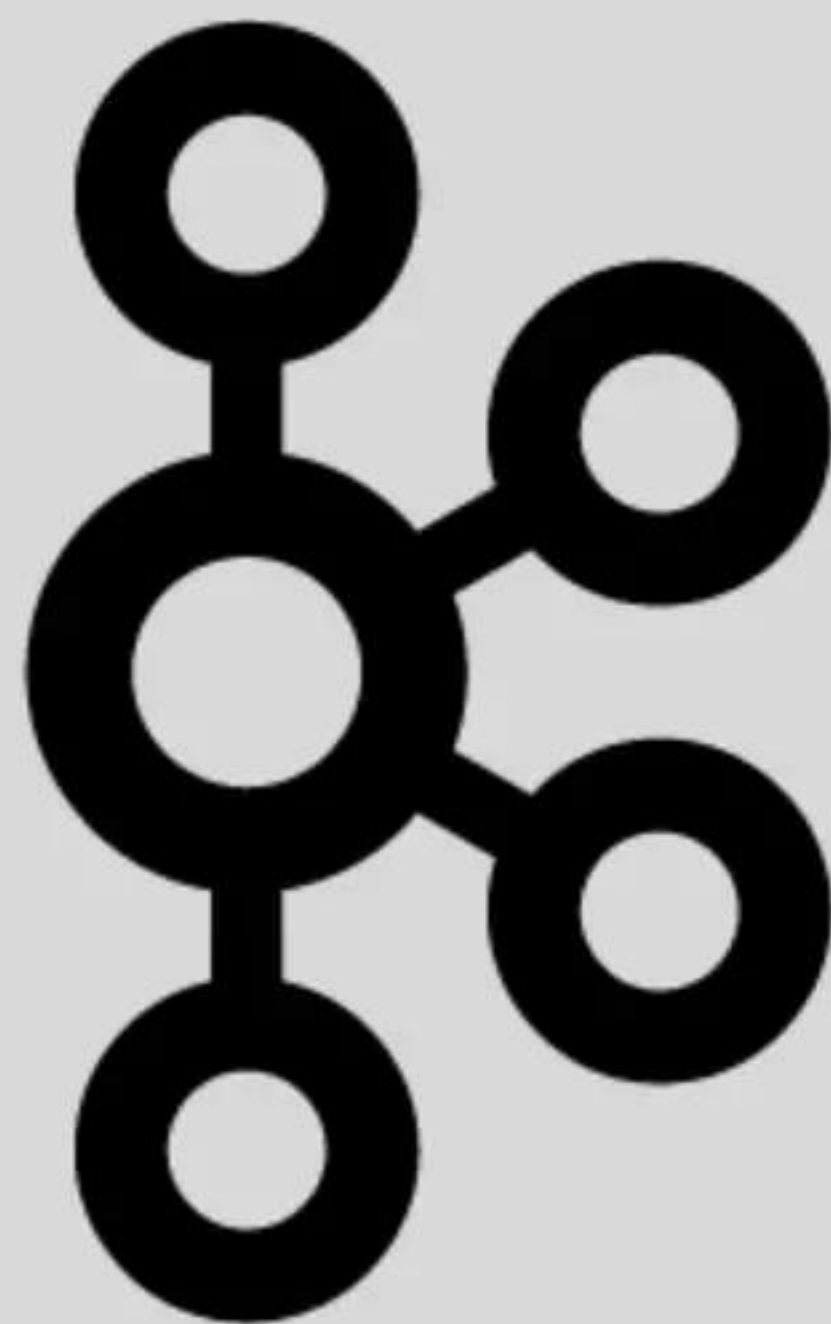
500+

10k - 100k+

Messages per second
(Apache Kafka)

Scaling the data pipeline even further



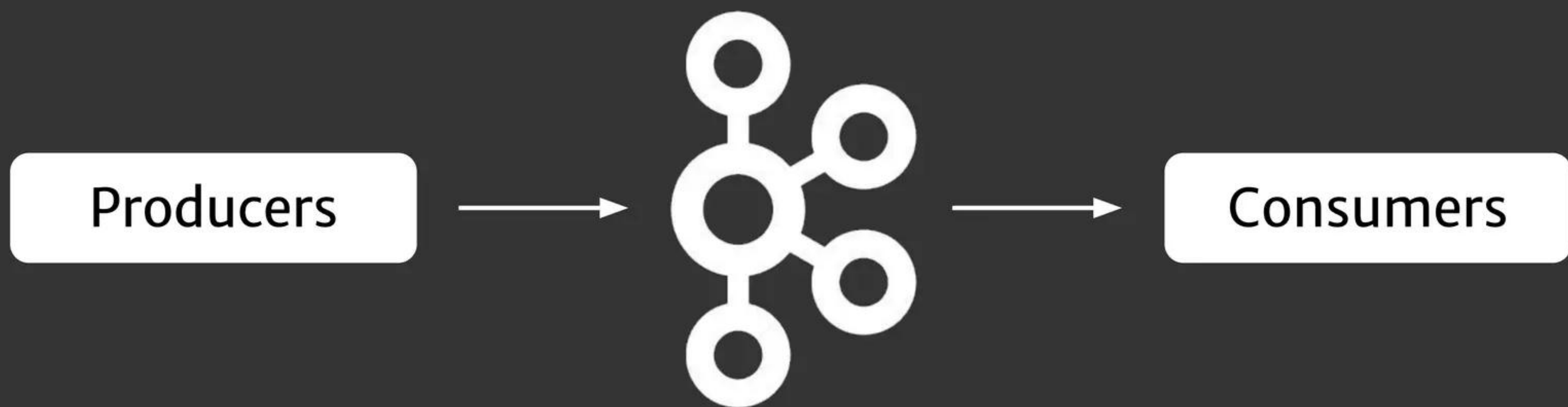


kafka

Kafka topics are partitioned and replicated

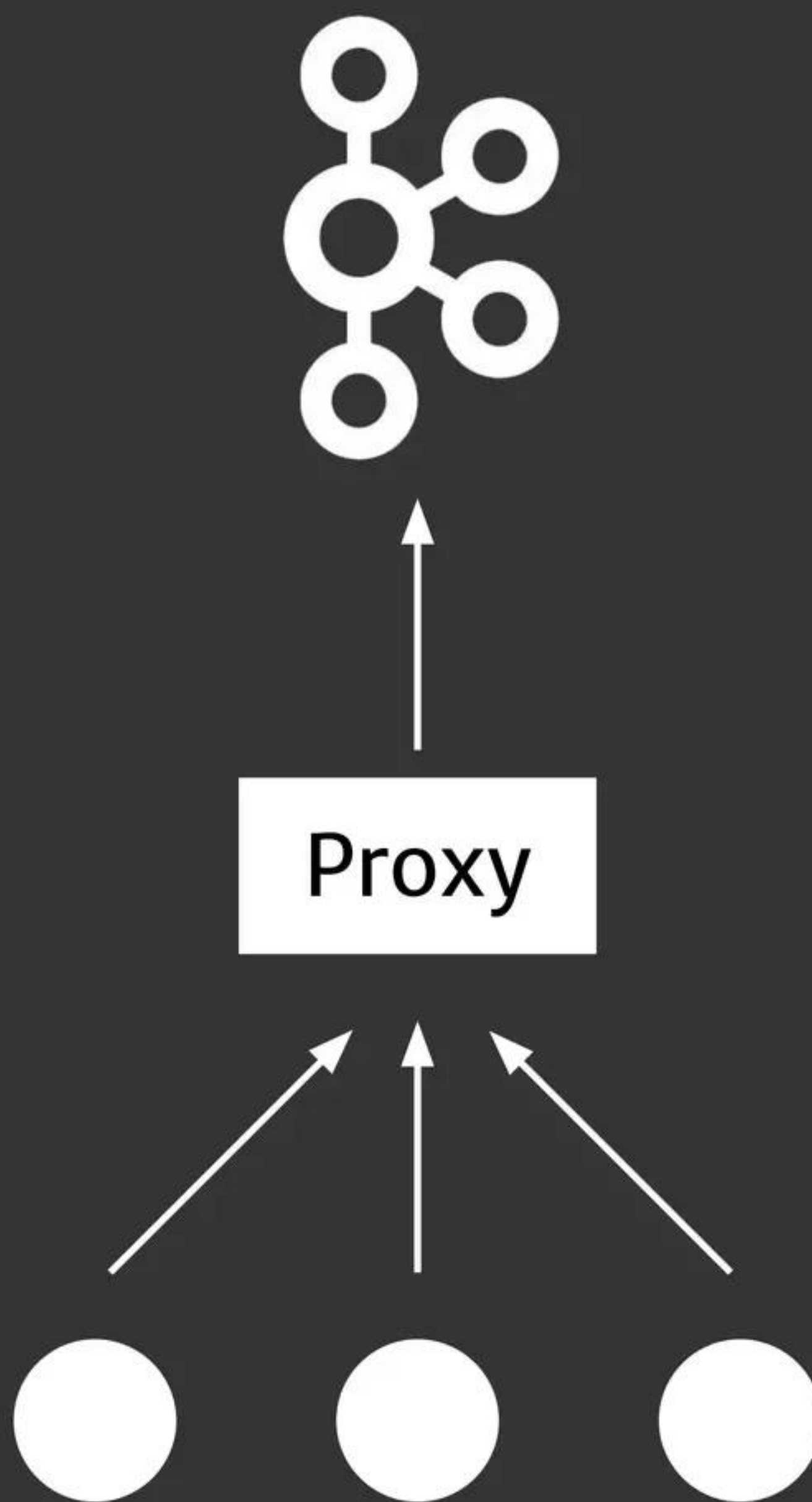
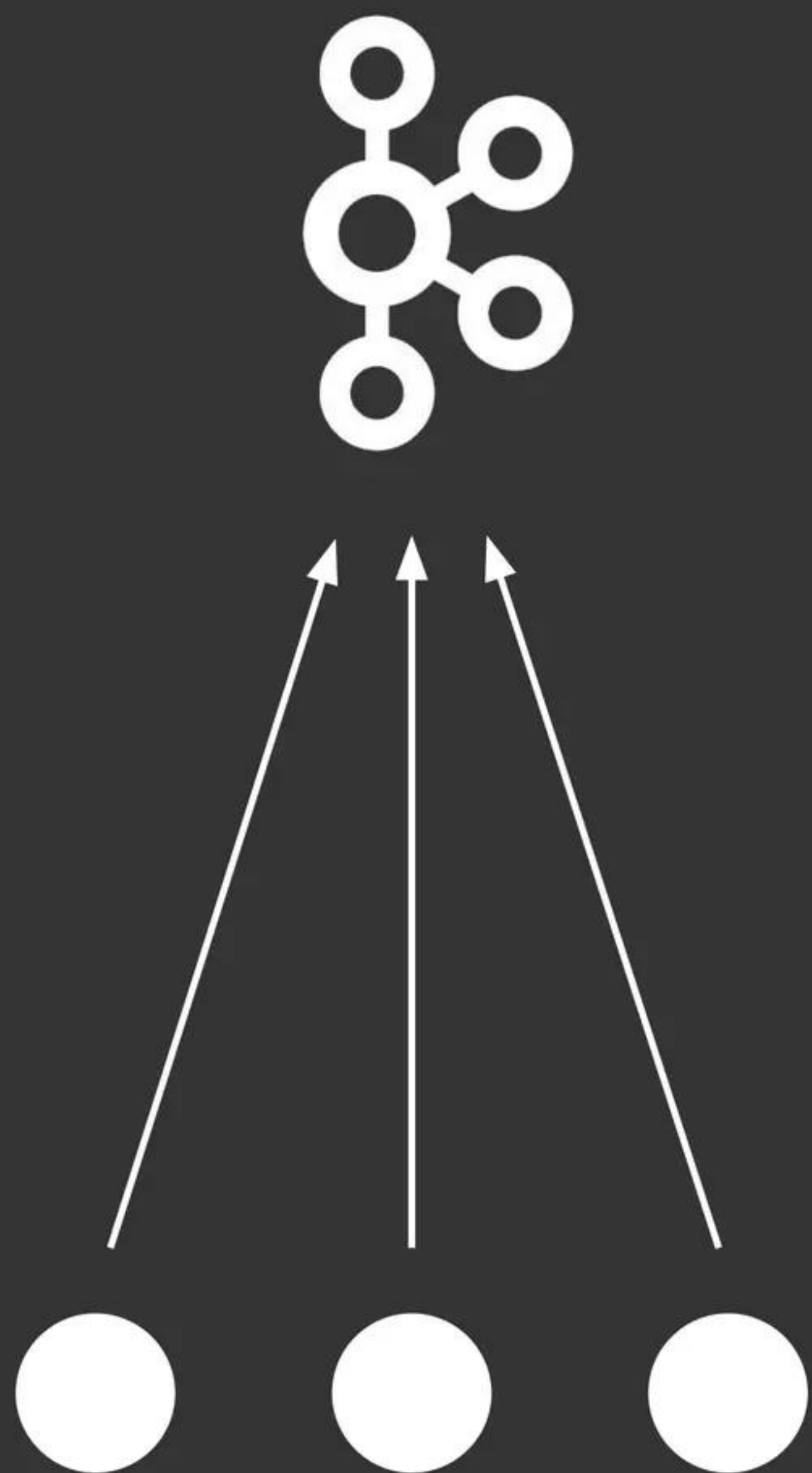


Scaling the pipeline
in terms of **Volume**



Scaling producers

- Asynchronous / non-blocking writes (default)
- Compression and batching
- Sampling
- Throttling
- Acks? 0, 1, -1
- Standard Kafka producer tuning: batch.size, linger.ms, buffer.memory, etc.



Each approach has pros and cons

- Simple
- Low-latency connection
- Number of TCP connections per broker starts to look scary
- Really hard to do maintenance on Kafka clusters
- Flexible
- Possible to do basic enrichment
- Easier to manage Kafka clusters

Simple rule for high-performant producers? Just write to Kafka, nothing else¹.

1. Not even auth?

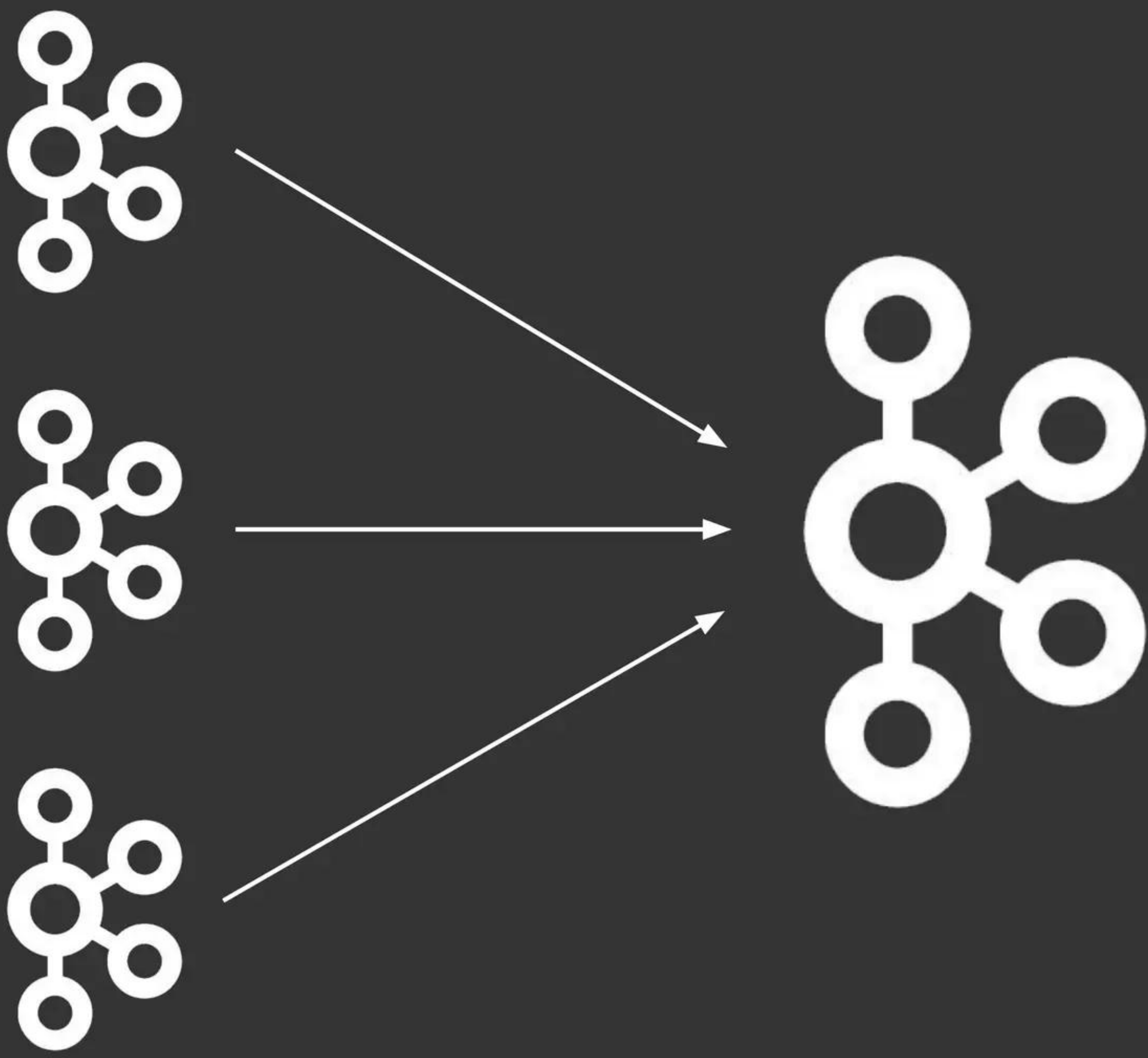
Scaling Kafka clusters

- Just add more nodes!
- Disk IO is extremely important
- Tuning *io.threads* and *network.threads*
- Retention
- For more: “[Optimizing Your Apache Kafka Deployment](#)” whitepaper from Confluent

It's not always about tuning. Sometimes we need more than one cluster.

Different workloads require different topologies.





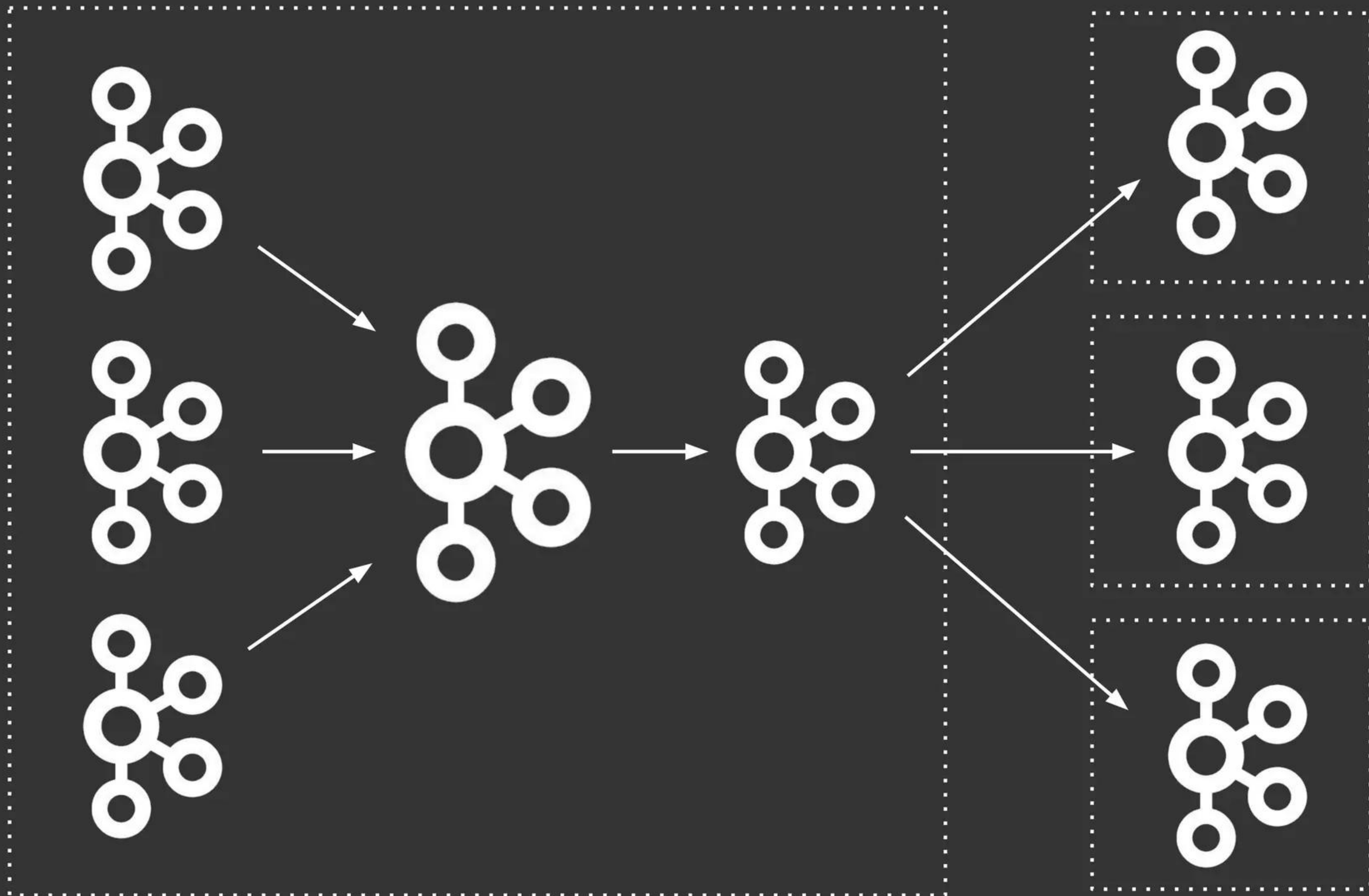


- Stream processing
- Short retention
- More partitions



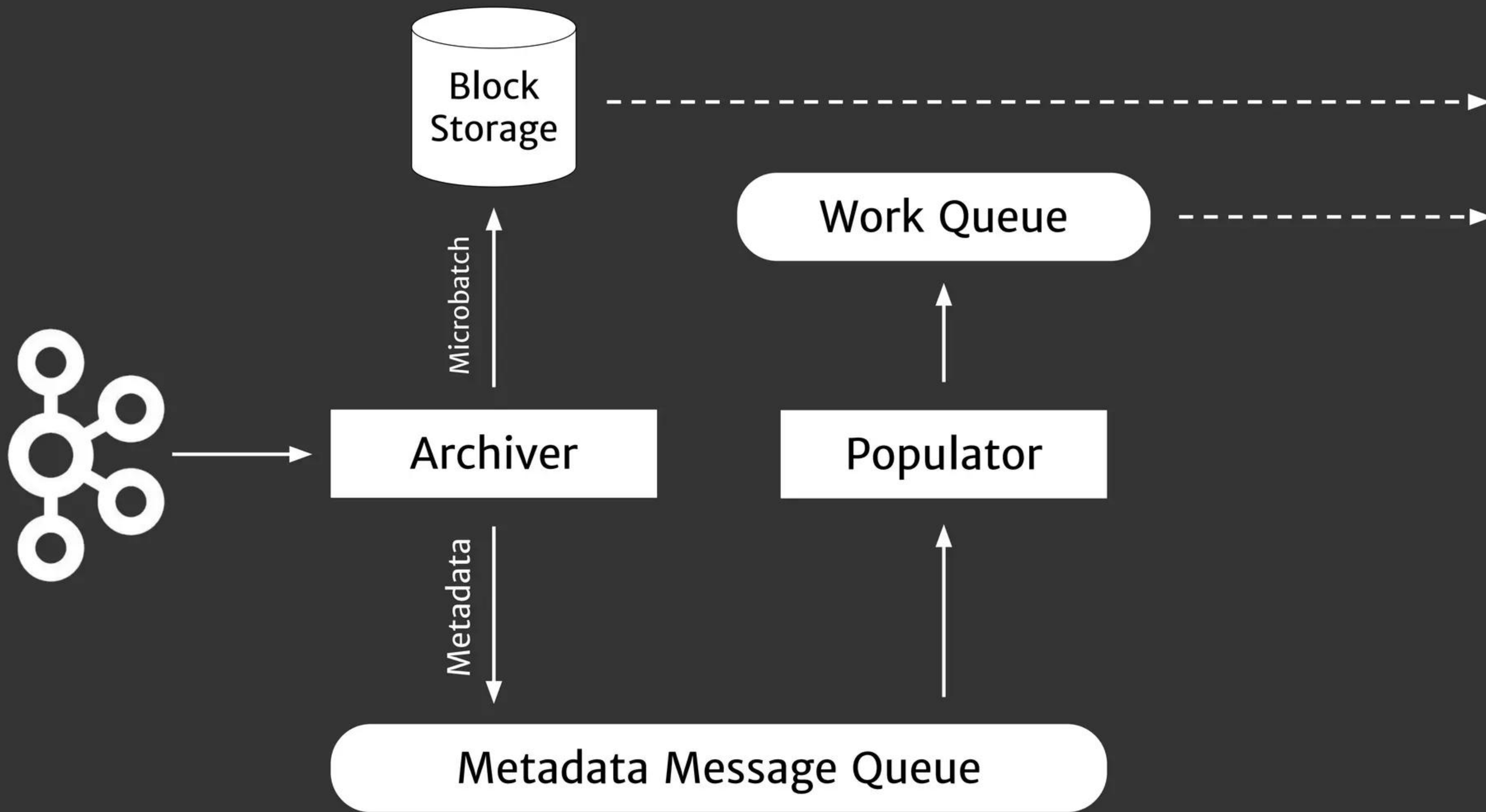
- Ingestion (HTTP Proxy)
- Long retention
- High SLA

- Lots of consumers
- Medium retention
- ACL



Scaling consumers is
usually pretty trivial -
just increase the number of
partitions.

Unless... you can't. What
then?



Even if you can add more partitions

- Still can have bottlenecks within a partition (large messages)
- In case of reprocessing, it's really hard to quickly add A LOT of new partitions AND remove them after
- Also, number of partitions is not infinite

You can't be sure about
any improvements without
load testing.

Not only for a cluster,
but producers and
consumers too.

Scaling and extending
the pipeline in terms of
Games and Use-cases

We need to keep the number of topics and partitions low

- More topics means more operational burden
- Number of partitions in a fixed cluster is not infinite
- Autoscaling Kafka is impossible, scaling is hard

Topic naming convention

Producer

Unique game id
"CoD WW2 on PSN"

\$env.\$source.\$title.\$category-\$version

The diagram illustrates the topic naming convention. It shows a template string with four variables: \$env (red), \$source (blue), \$title (green), and \$category-\$version (grey). Two arrows point from labels above to the template: 'Producer' points to \$env, and 'Unique game id' (with the example 'CoD WW2 on PSN') points to \$title.

prod.glutton.1234.telemetry_match_event-v1

A proper solution has
been invented decades
ago.

Think about databases.

Messaging system IS a form of a database

Data topic = Database + Table.

= Namespace + Data type.

Compare this

prod.glutton.1234.telemetry_match_event-v1

dev.user_login_records.4321.all-v1

prod.marketplace.5678.purchase_event-v1

telemetry.matches

user.logins

marketplace.purchases

Each approach has pros and cons

- Topics that use metadata for their names are obviously easier to track and monitor (and even consume).
- As a consumer, I can consume exactly what I want, instead of consuming a single large topic and extracting required values.
- These dynamic fields can and *will* change. Producers (sources) and consumers *will* change.
- Very efficient utilization of topics and partitions.
- Finally, it's impossible to enforce any constraints with a topic name. And you can always end up with dev data in prod topic and vice versa.

After removing
necessary metadata
from the topic names
stream processing
becomes mandatory.

Stream processing becomes mandatory

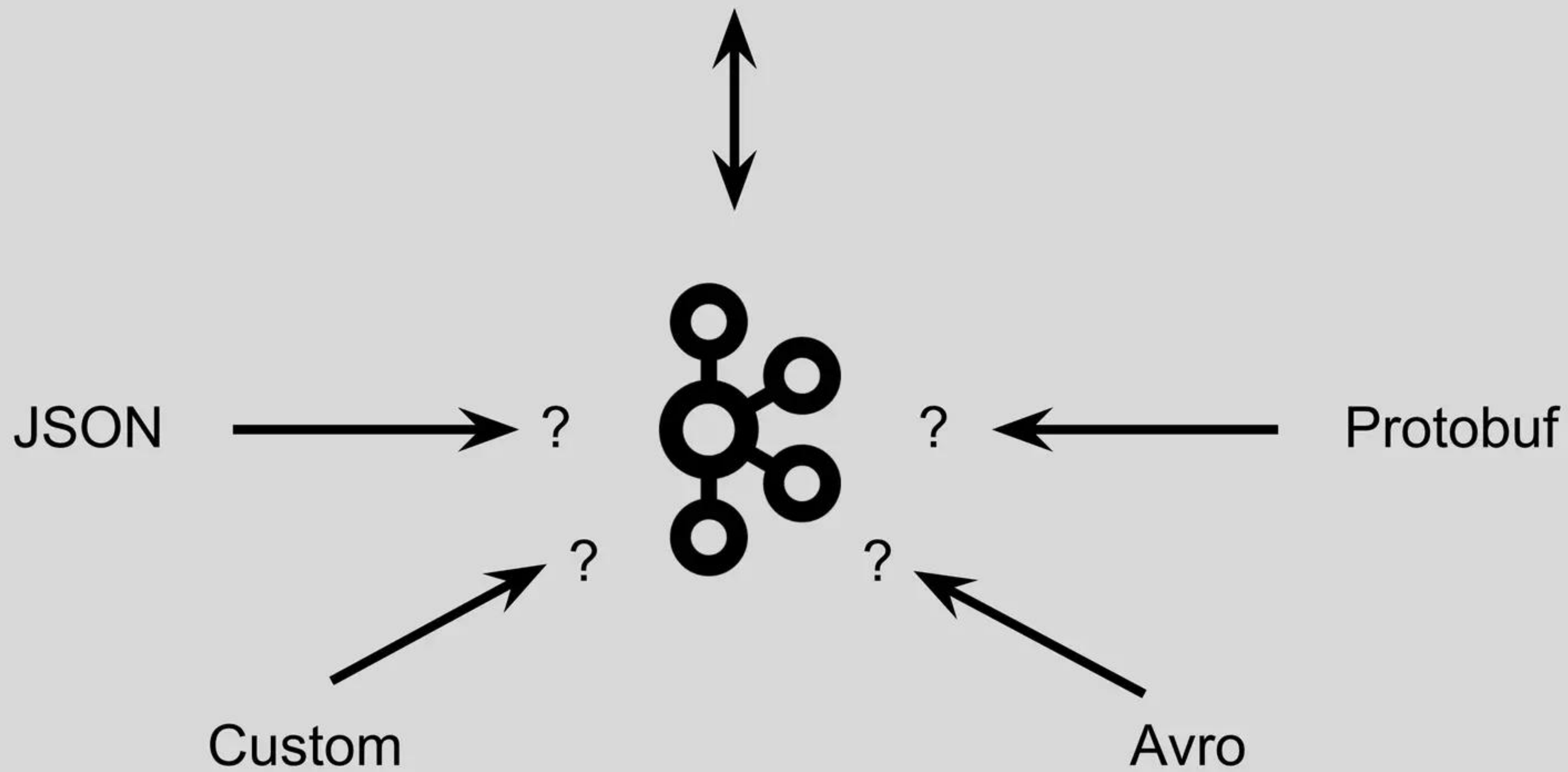
Measuring → Validating → Enriching → Filtering & routing

Having a single
message schema for a
topic is more than
just a nice-to-have.

Number of supported
message formats

8

Stream processor



Custom deserialization

```
// Application.java  
props.put("value.deserializer", "com.example.CustomDeserializer");
```

```
// CustomDeserializer.java  
public class CustomDeserializer implements Deserializer<??> {  
    @Override  
    public ??? deserialize(String topic, byte[] data) {  
        ???  
    }  
}
```

Message envelope anatomy



Unified message envelope

```
syntax = "proto2";
```

```
message MessageEnvelope {  
    optional bytes message_id = 1;  
    optional uint64 created_at = 2;  
    optional uint64 ingested_at = 3;  
    optional string source = 4;  
    optional uint64 title_id = 5;  
    optional string env = 6;  
    optional UserInfo resource_owner = 7;  
    optional SchemaInfo schema_info = 8;  
    optional string message_name = 9;  
    optional bytes message = 100;  
}
```

Schema Registry

- API to manage message schemas
- **Single** source of truth for all producers and consumers
- It should be **impossible** to send a message to the pipeline without registering its schema in the Schema Registry!
- Good Schema Registry supports **immutability**, **versioning** and basic **validation**
- Activision uses custom Schema Registry implemented with Python and Cassandra

Summary

- Kafka **tuning** and **best practices** matter
- Invest in good **SDKs** for producing and consuming data
- Unified **message envelope** and **topic names** make adding a new game almost effortless
- “Operational” **stream processing** makes it possible. Make sure you can support adhoc **filtering** and **routing** of data
- Topic names should express **data types**, not producer or consumer metadata
- **Schema Registry** is a must-have

Thanks !

@sap1ens