



Bravo Six, Going Realtime. Transitioning Activision Data Pipeline to Streaming

© 2020 Activision Publishing, Inc.



Hello!

I am Yaroslav Tkachenko

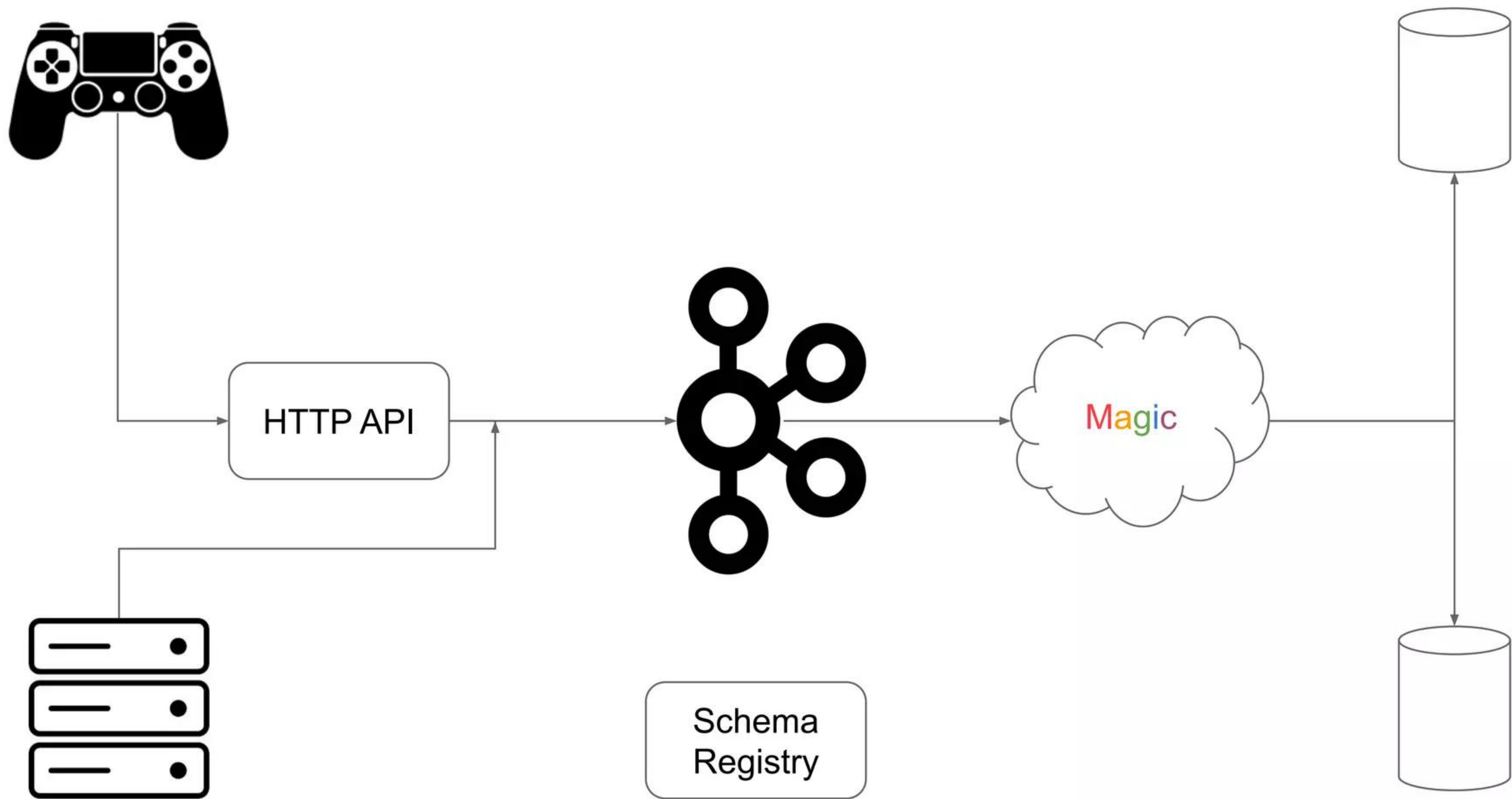
Software Architect at **Activision Data**.

You can find me at **@sap1ens** (pretty much everywhere).



Activision Data Pipeline

- Ingesting, processing and storing game telemetry data
- Providing **tabular**, **API** and **streaming** access to data





200k+ msg/s

Ingestion rate

5+ PB

Data lake size (AWS S3)

9 years

Age of the oldest game



Challenges

- Complex client-side & server-side game telemetry
- Long-living titles, hard to update or deprecate
- Various data formats, message schemas and envelopes
- Development data == production data
- Scalability, elasticity & cost



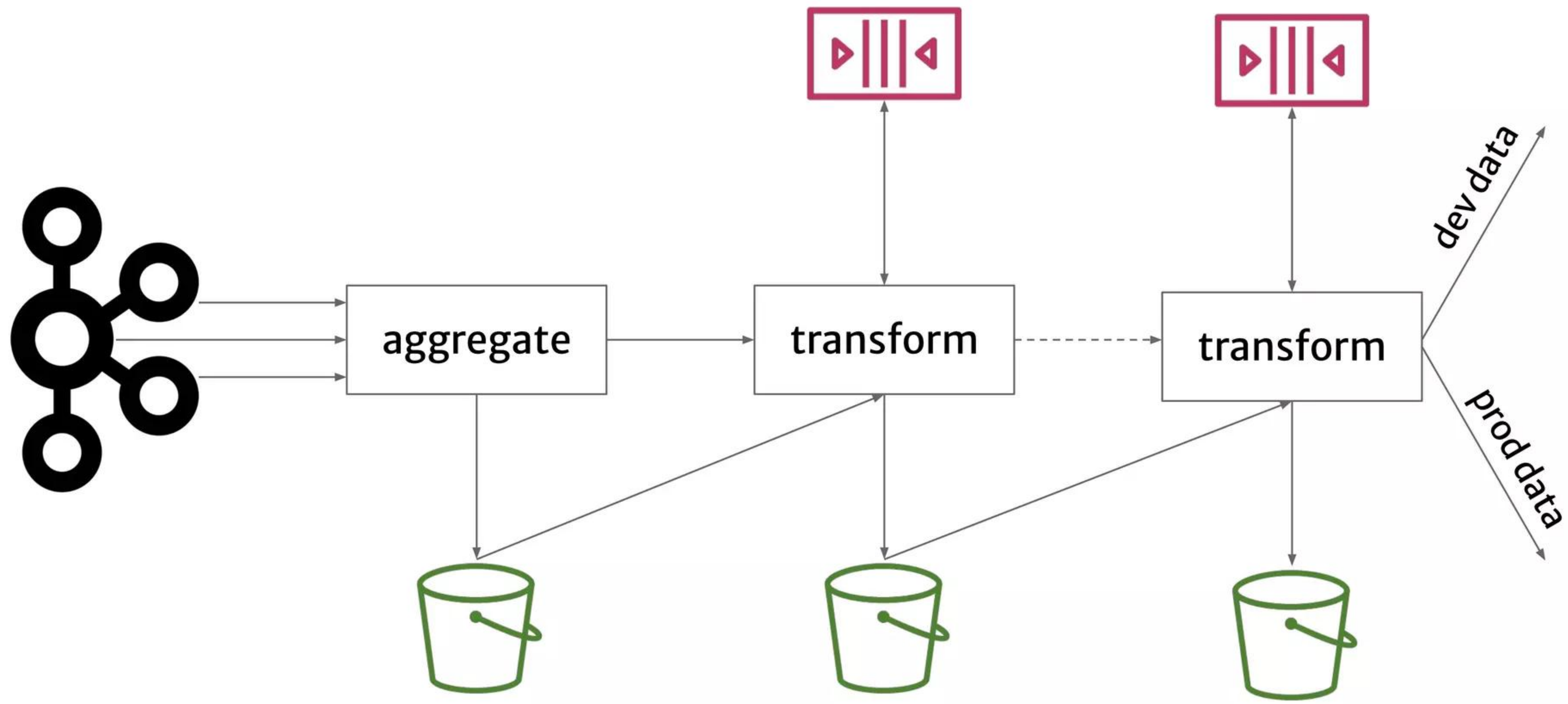
Established standards

- Kafka topic name conventions must be followed
- Payload schema must be uploaded to the Schema Registry
- Message envelope has a schema too (Protobuf), with a set of required fields

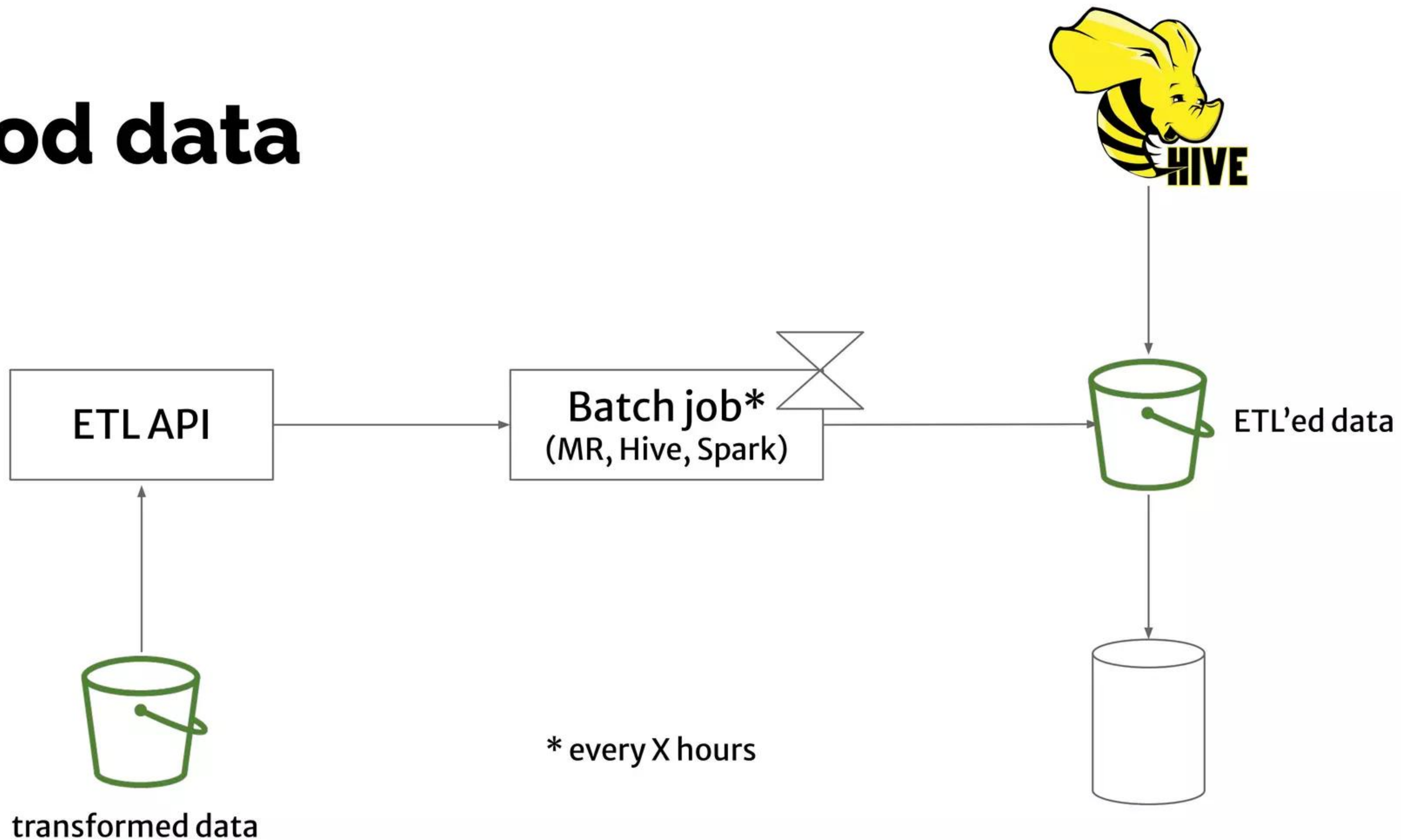


Old pipeline

Quick overview



Prod data





Old pipeline

Architecture Flaws

- Scalability solution as a workaround
- Painful to switch between dev & prod
- No streaming capabilities
- Adhoc integration

Bottlenecks

- Latency limitations
- MR glob length, memory is not infinite (ETL API), etc.
- Lots of manual configuration
- Lots of manual ETL



New pipeline

It gets better from here



Apache **Kafka**

- The **Streams API** allows an application to act as a **stream processor**, **consuming** an input stream from one or more topics and **producing** an output stream to one or more output topics, effectively **transforming** the input streams to output streams.
- The **Connector API** allows building and running reusable producers or consumers that **connect** Kafka topics to existing **applications** or **data systems**. For example, a connector to a relational database might capture every change to a table.



~10 seconds

End-to-end streaming latency

6-24 hours → 5-10 mins

Tabular data available for querying

90% cheaper

Per user/byte



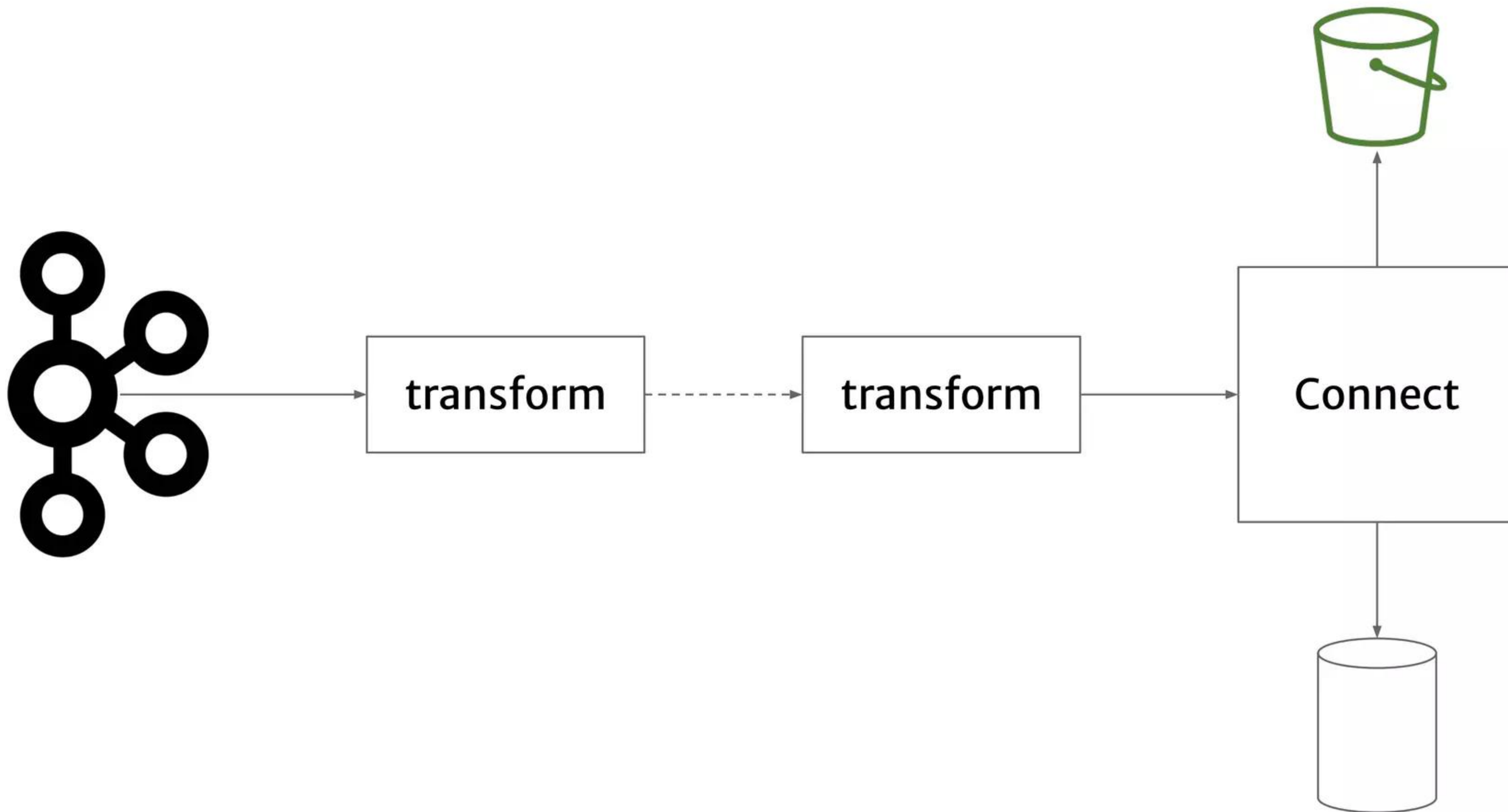
Guiding principles

Kafka Streams

- One transformation step = one service*
 - Not entirely true anymore, we've combined some steps to optimize cost and reduce unnecessary IO
- Stateless if possible
- Rich routing
- Auto-scaling & self-healing
- LOTS of tooling

Kafka Connect

- Handle integration - AWS S3, Cassandra, Elasticsearch, etc.
- Only sink connectors
- Invest in configuration, deployments, monitoring



Why Kafka Streams?





Our internal **protocol**

Kafka Message Headers



Schema guid
Other metadata,
mostly for routing

Kafka Message Key



Null (99%)

Kafka Message Value



Serialized **Avro**



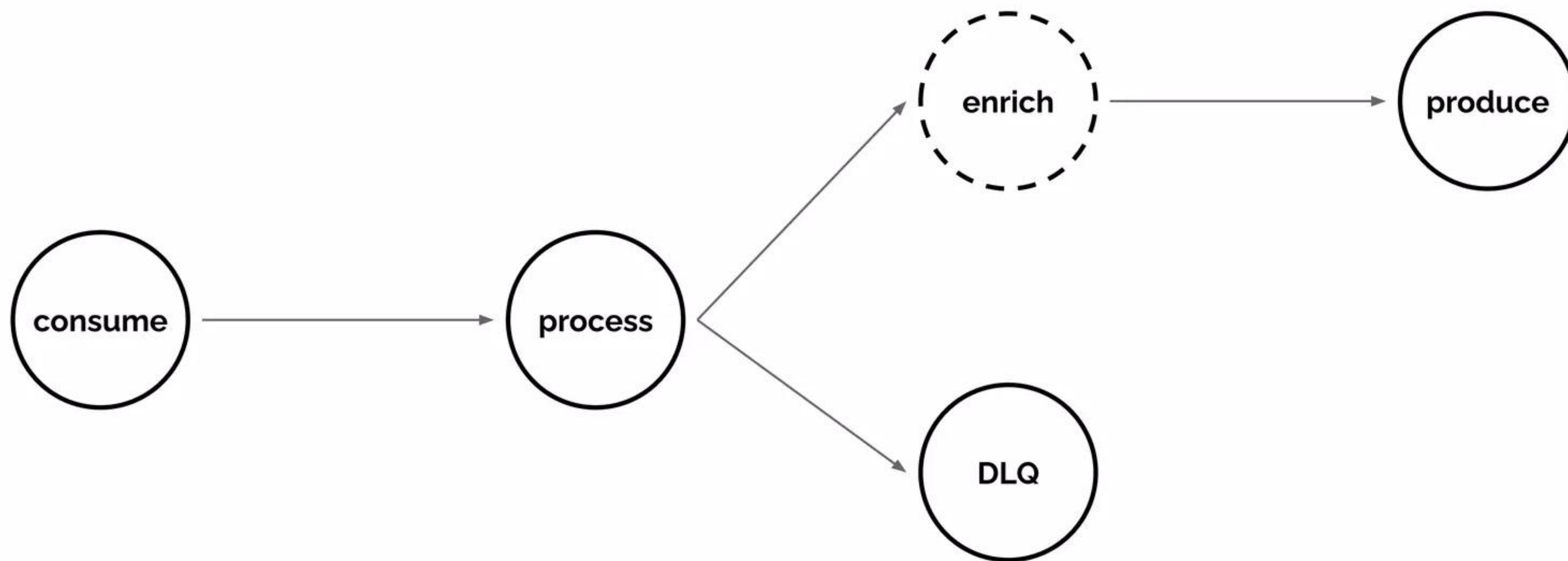
Schema management

- Schemas are generated & uploaded automatically if needed. Schema hash is used as id
- Make schemas immutable and cache them aggressively. You have to use them for *every single record*!





Typical Kafka Streams service **topology**





```
1 KStream[] streams = builder
2   .stream(Pattern.compile(applicationConfig.getTopics()))
3   .transform(MetadataEnricher::new)
4   .transform(() -> new InputMetricsHandler(applicationMetrics))
5   .transform(ResultExtractor::new)
6   .transform(() -> new OutputMetricsHandler(applicationMetrics))
7   .branch(
8       (key, value) -> value instanceof RecordSucceeded,
9       (key, value) -> value instanceof RecordFailed,
10      (key, value) -> value instanceof RecordSkipped
11  );
12
13 // RecordSucceeded
14 streams[0].map((key, value) -> KeyValue.pair(key, ((RecordSucceeded)
value).getGenericRecord()))
15   .transform(SchemaGuidEnricher<String, GenericRecord>::new)
16   .to(new SinkTopicNameExtractor());
17
18 // RecordFailed
19 streams[1].process(dlqFailureResultHandlerSupplier);
```



Routing & configuration

Before:

<env>.<producer>.<title>.<category>-<protocol>

e.g.

prod.service-a.1234.match_summary-v1

“raw” data, no transformations



Routing & configuration

Now:





Routing & configuration

prod.rdp.mw.ingested



prod.rdp.mw.parsed

microservice

```
prodMwServiceA:
  stream:
    headers:
      env: prod
      game: mw
      source: service-a
      exclude: <thingX>
  action:
    type: parse
    protocol: proto2
```



Routing & configuration

prod.rdp.mw.ingested



prod.rdp.mw.parsed

microservice

Streams can be skipped, split, merged, sampled, etc.

```
prodMwServiceA:
  stream:
    headers:
      env: prod
      game: mw
      source: service-a
      exclude: <thingX>
    action:
      type: parse
      protocol: proto2
```



Dynamic Routing*

- Centralized, declarative configuration
- Self-serve APIs and UIs
- Every change is automatically applied to all running services within seconds



Infra & Tools

- One-click Kafka deployment (Jenkins, Ansible)
- Kafka broker EBS auto-scaling
- Versioned & deployable Kafka topic configuration
- Built tooling for:
 - Data reprocessing and DLQ resubmission
 - Offset migration between consumer groups
 - Message inspection
 - ...



Auto-scaling & self-healing

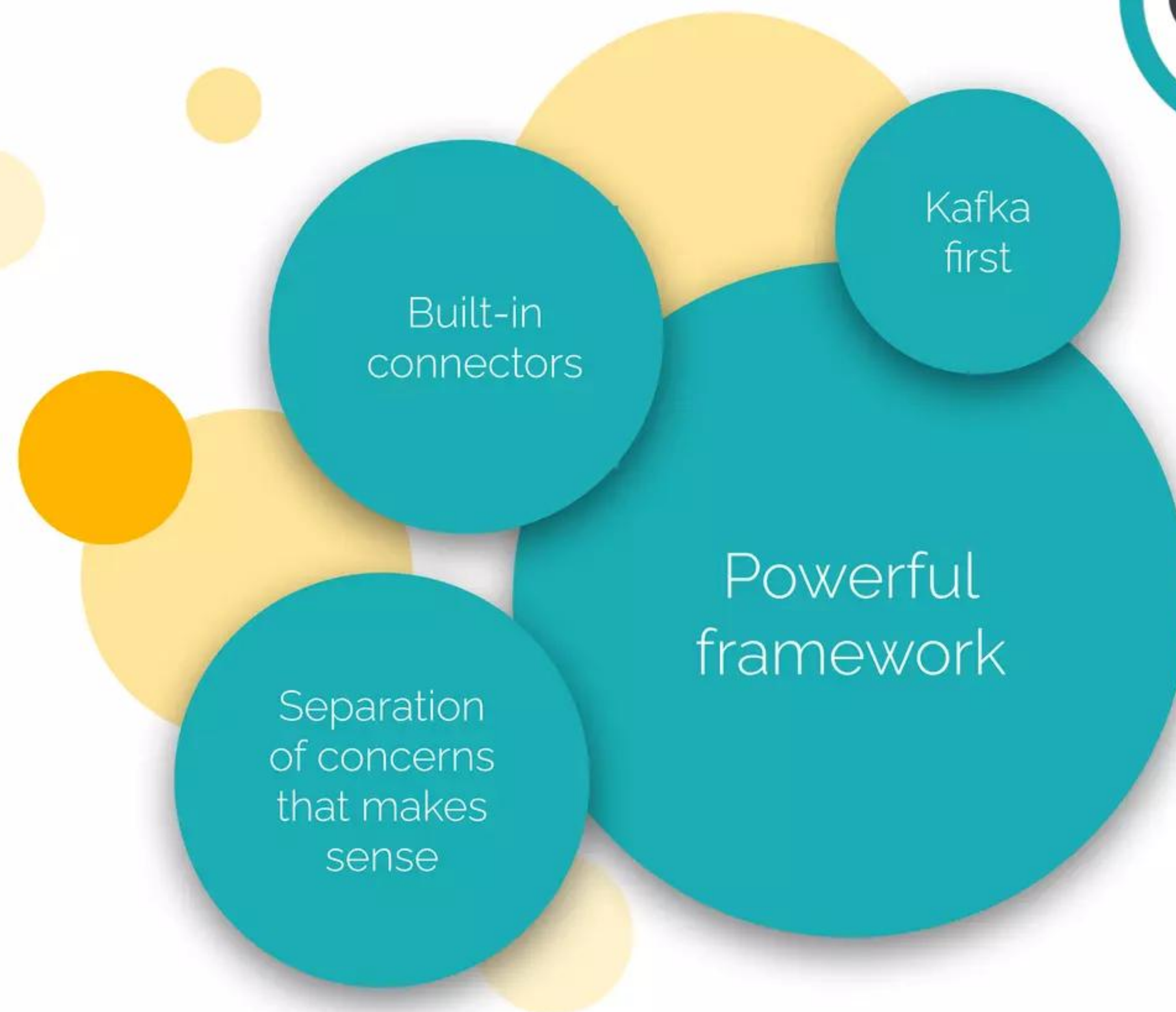
Scaling

- Every application submits **<app_name>.lag** metric in milliseconds
- ECS Step Scaling: add/remove X more instances every Y minutes
- Add an extra policy for rapid scaling

Healing

- Heartbeat endpoint monitors **streams.state()** result
- ECS healthcheck replaces unhealthy instances
- Stateful applications need more time to bootstrap

Why Kafka Connect?





Kafka Connect

- Multiple smaller clusters > one big cluster
- Connectors configuration [lives in git, uses Jsonnet](#).
Deployment script leverages REST API
- Custom Converter, thanks to [KIP-440](#)
- ❤️ [lensesio/kafka-connect-ui](#)
- Collecting & using tons of metrics available over JMX



C* Connector

- Implemented from scratch, inspired by JDBC connector
- Started with porting over existing C* integration code
- Took us a few days (!) to wrap it up
- Generalizing is hard
- Very performant, usually just a few tasks are running



ES Connector

- Using open-source [kafka-connect-elasticsearch](#)
- Leveraging SMTs to:
 - Partition single topic into multiple indexes
 - Enrich with a timestamp
- Currently very low-volume



S3 Connector

- Started with forking open-source [kafka-connect-s3](#)
- Added custom Avro and Parquet formats
- Added a new flexible partitioner
- Optimized connector for at-least-once delivery
 - Generate less files on S3, reduce TPS
 - Avoid file overrides with non-deterministic upload triggers
- Running hundreds of tasks



Dev data is **prod** data

- Scale is different, but the pipeline is the same
- Running as a separate set of services to reduce latency, low latency is a requirement
- Different approach to alerting

Otherwise, it's the same!



Use Case: RADS

Flatten my data!



```
{
  "headers": {
    "field1": "value1",
  },
  "data": {
    "match": {
      "field2": "value2"
    },
    "players": [
      {"field3": "value3",
       "field4": "value4"},
      {"field3": "value3",
       "field4": "value4"}
    ]
  }
}
```



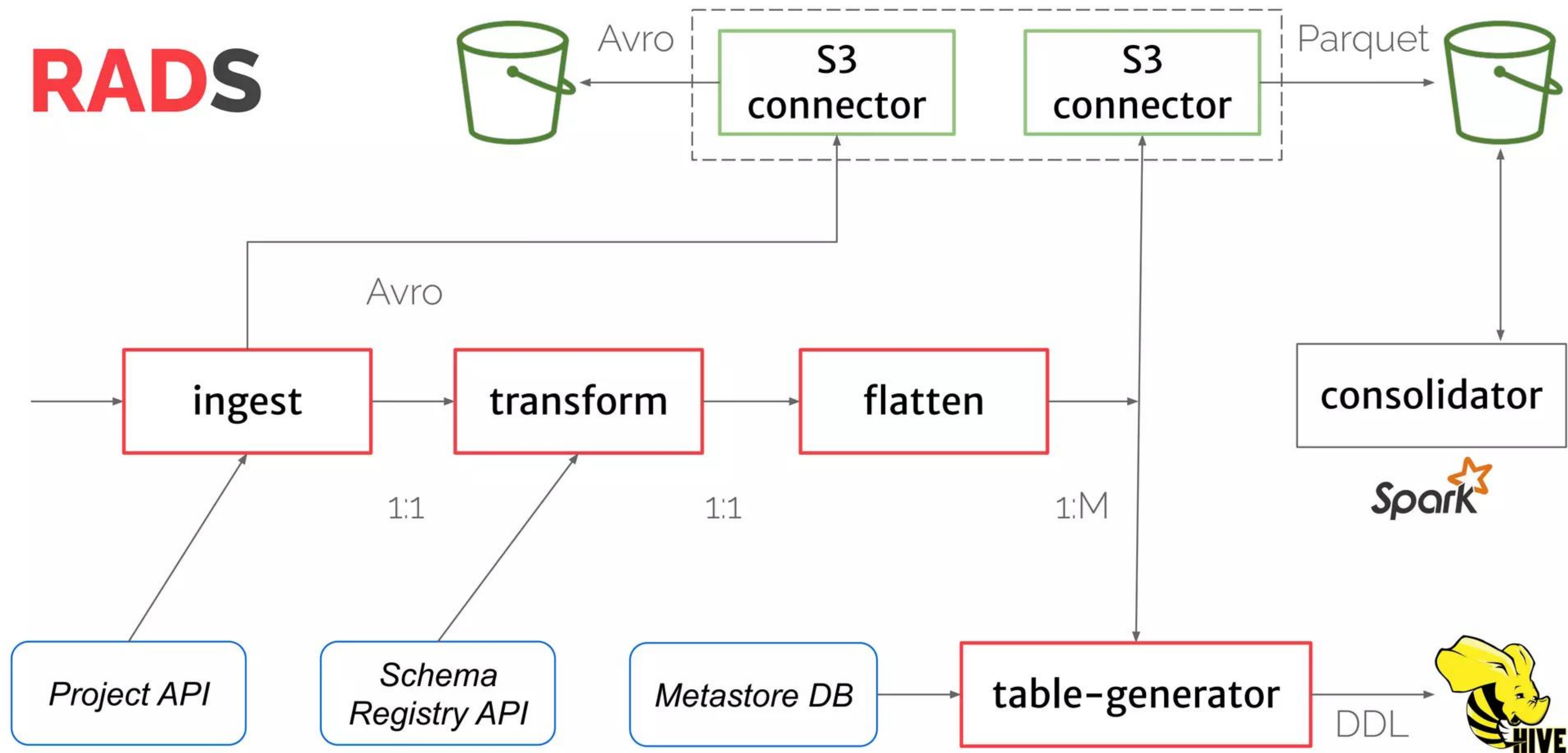
fact_data

message_id	context_headers_field1_s	data_match_field2_s
...	...	...
...	...	...

fact_data_players

message_id	index	context_headers_field1_s	data_players_field3_i	...
...	...	...	...	...
...	...	...	...	...

RADS





Why is **RADS** *rad*?

- Has enough automation and generic configuration to automatically create Hive databases, tables, add new columns and partitions for a **brand new game** with no* human intervention.
- As a data producer you just need to start sending data in the right format to the right Kafka topic, **that's it!**
- We get **realtime** ("hot") and **historical** ("cold") data in the same place!



Thanks!

Any questions?

@sap1ens