It's Time To Stop Using Lambda Architecture

Yaroslav Tkachenko

Batch

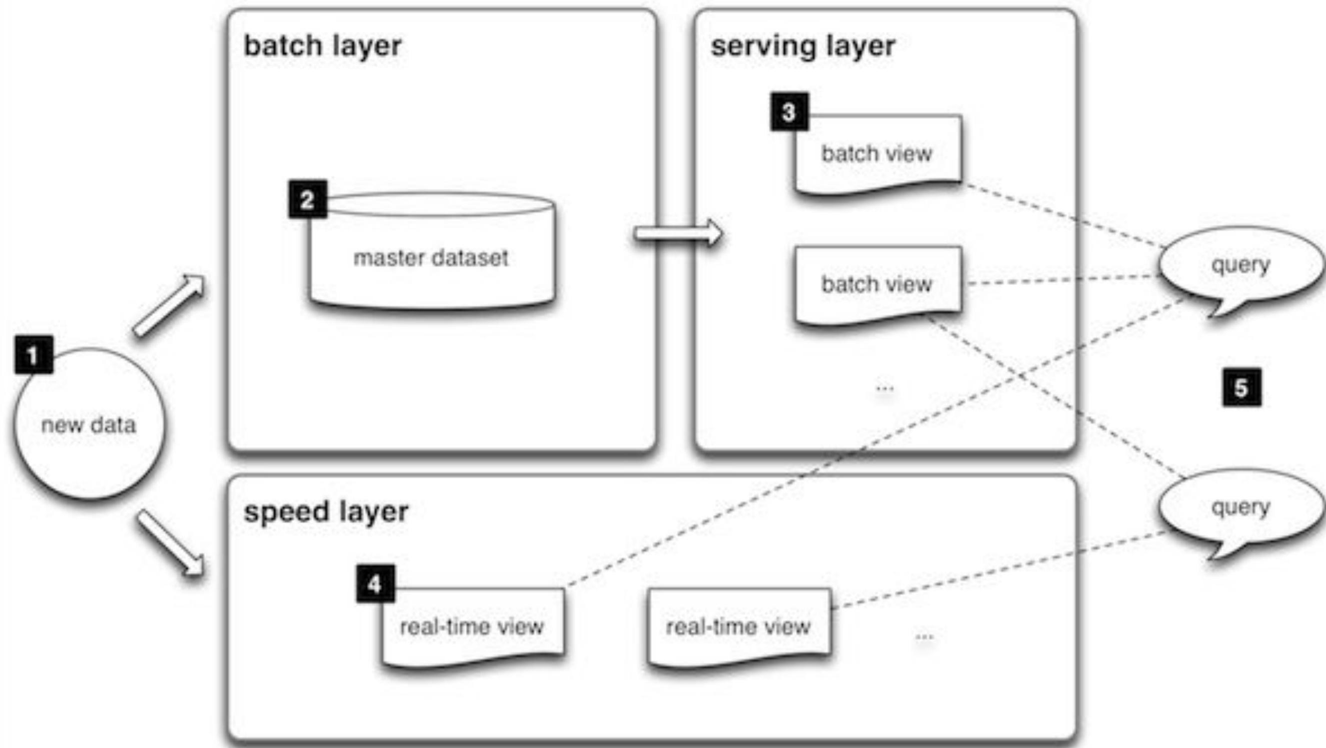
Streaming

👋 Hi, I’m Yaroslav

Staff Data Engineer @ Shopify

I like moving things from
Batch to Streaming. A lot.

λ Architecture

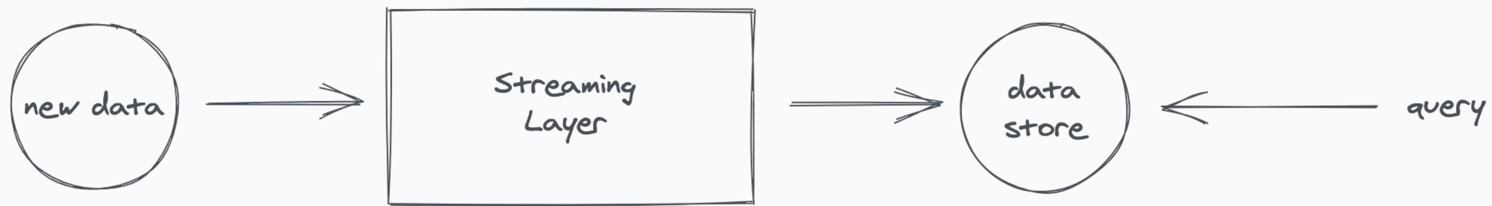


Classic Lambda Architecture © Databricks

Other Lambda Incarnations

- “Let’s run a batch job to **fix** the data”
- “Let’s run a batch job to **optimize** file size”
- “Let’s run a batch job to **reprocess** everything”

K Architecture



Classic Kappa Architecture

Kappa Concerns

- Data availability / retention
- Data consistency
- Handling late-arriving data
- Data reprocessing & backfill

Before we continue...

…why do I like streaming so much?

Latency

- It’s actually not the main goal, but it’s a *nice* one!
- You have no idea how much latency is OK

Handling Late-Arriving Data: Batch

- How much time should we wait?
- How much time is OK to reprocess?

Handling Late-Arriving Data: Streaming

- Stateless transformations and sinks with updates: **easy**
- Stateful transformations:
    - Small state: **easy**
    - Large state: *doable*
- Sinks without updates: *it depends*

Operations and Observability: Batch

- “Yeah, it occasionally fails, we just wait 6 hours for a retry run”
- “Oh, I disabled the wrong job and nobody noticed”
- “We don’t really have metrics for this job, but you can monitor it with this UI”

Operations and Observability: Streaming

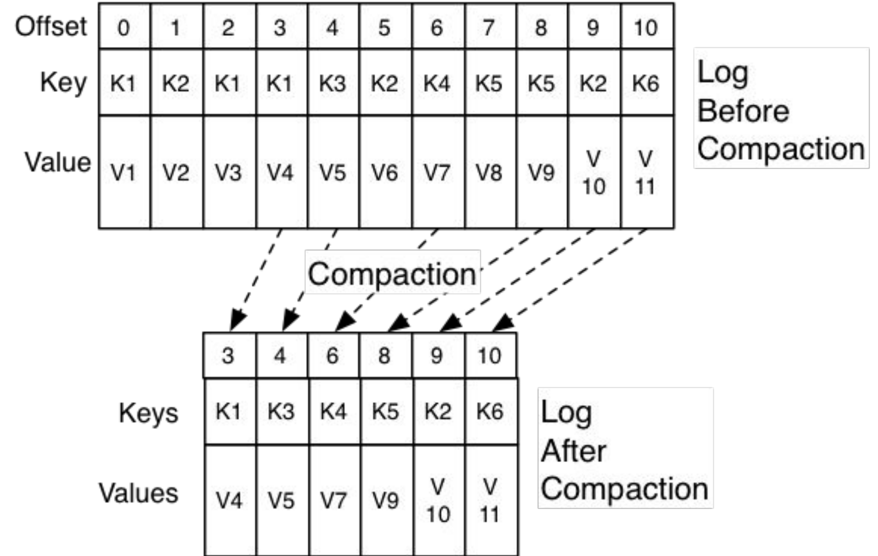
- Modern frameworks like Kafka Streams and Apache Flink can be deployed with orchestration systems like Kubernetes
- The same SLO and uptime expectations as with applications serving traffic
- You can **fully** embrace CI/CD, observability and other DevOps/SRE practices and mentality

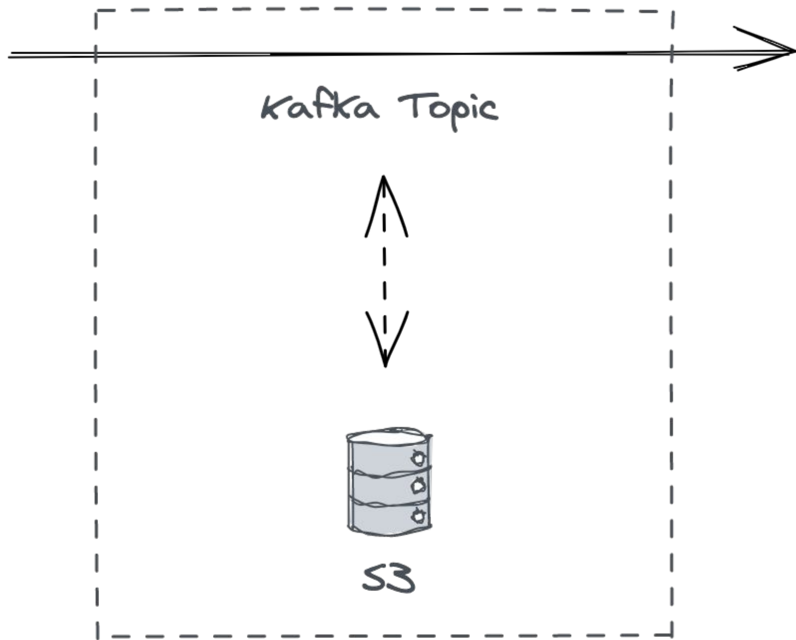
K Building Blocks

Core areas: The Log (Kafka), Streaming Framework (Flink), Sinks (e.g. Iceberg)

Kafka Topic Compaction

- Can be used if intermediate values (per key) are not important
- Enables infinite retention





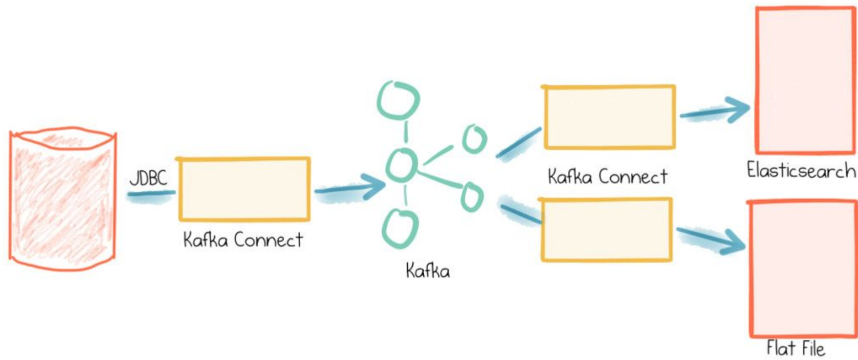
Kafka Tiered Storage

- WIP (KIP-405)
- Enables infinite retention for all topics
- “Topic Archive Pattern” have been used for years

Kafka Exactly-Once

- Introduced in 0.11, 4 years ago (!)
- Eliminates duplicates, ensures consistency

```
producer.initTransactions();
try {
    producer.beginTransaction();
    producer.send(record1);
    producer.send(record2);
    producer.commitTransaction();
} catch(ProducerFencedException e) {
    producer.close();
} catch(KafkaException e) {
    producer.abortTransaction();
}
```

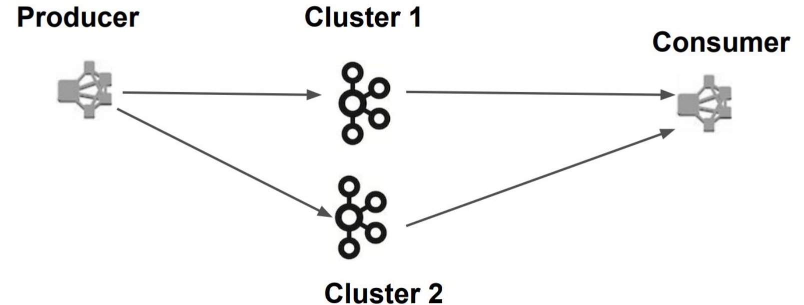


Data Integration

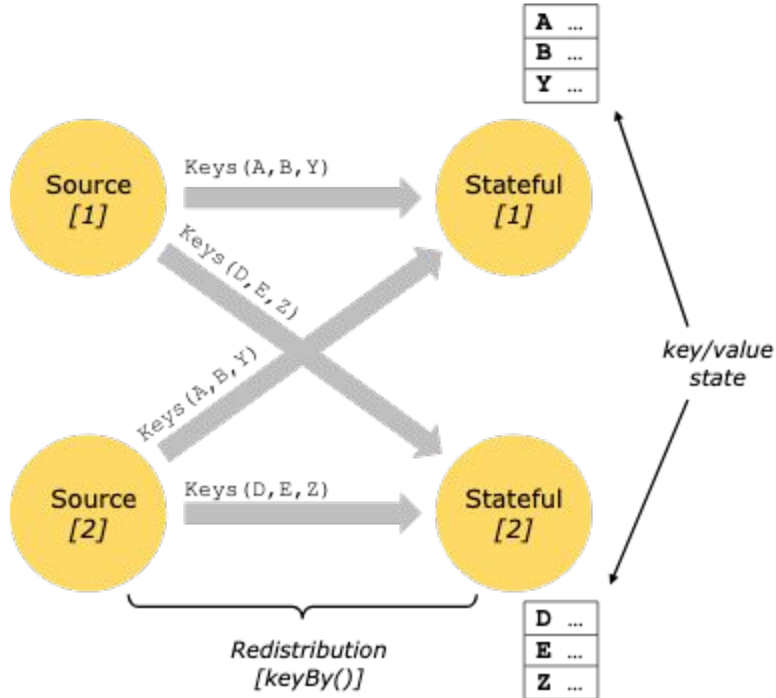
- Bringing all types of data to Kafka
- Solves “but I don’t have this in Kafka” question
- Kafka Connect works best with Kafka, but there are other options
- Just avoid building one-off integrations
- Makes sense for sinks too!

Dynamic Kafka Clusters

- Transient Kafka clusters can be brought on-demand for large reprocessing
- This requires protocol changes for producers and consumers
- Netflix has done it



Reliable **and** ***scalable*** **state as a part of your streaming engine is mandatory for any complex** ***κ*** **use-case**



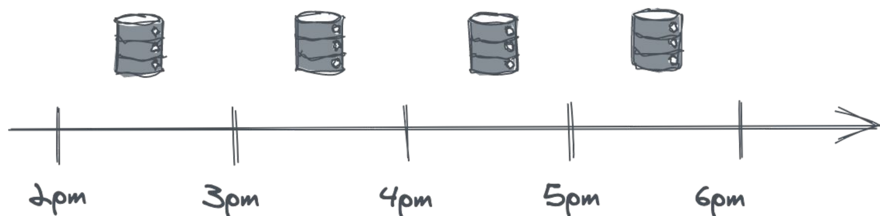
Flink State Concepts

- Keyed state is a *key* to scalable pipelines
- Checkpointing guarantees state fault-tolerance
- State is used as a building block in a lot of high-level components, you don’t always see it

Flink Exactly-Once

- Leverages state to support exactly-once semantics
- Has an advanced Kafka source/sink integration
- Custom sources and sinks can be created using standard patterns

```
abstract class TwoPhaseCommitSinkFunction
extends RichSinkFunction
implements CheckpointedFunction,
CheckpointListener
```



Flink State Management

- Custom state variables, timers and side outputs allow building very advanced workflows
- You can use state as a database
- Or you can even repopulate state when handling late-arriving messages

Flink State Processor API

- Don’t backfill your state by replaying from the source: update state directly
- Combines the power of batch and streaming by processing state with batch and then bootstrapping a streaming application

```
val listState  = savepoint
  .readListState(
    "my-state",
    "list-state",
    Types.INT
  )

// ...

Savepoint
  .create(new MemoryStateBackend(), 128)
  .withOperator("my-state", transformation)
  .write(savepointPath)
```

Some data stores are better suited to be used as data sinks for streaming pipelines than others

Supporting Updates/Upserts

Updates/Upserts can seriously simplify overall design: they can be used for data correction.

- Good
    - RDBMS, NoSQL (HBase, Cassandra, Elasticsearch), OLAP (Pinot), Compacted Kafka topics, “Lakehouse” object stores with Parquet*
- Problematic
    - OLAP (Druid, Clickhouse), Non-compacted Kafka topics, Regular object stores with Parquet*

“Lakehouse” Data Sinks

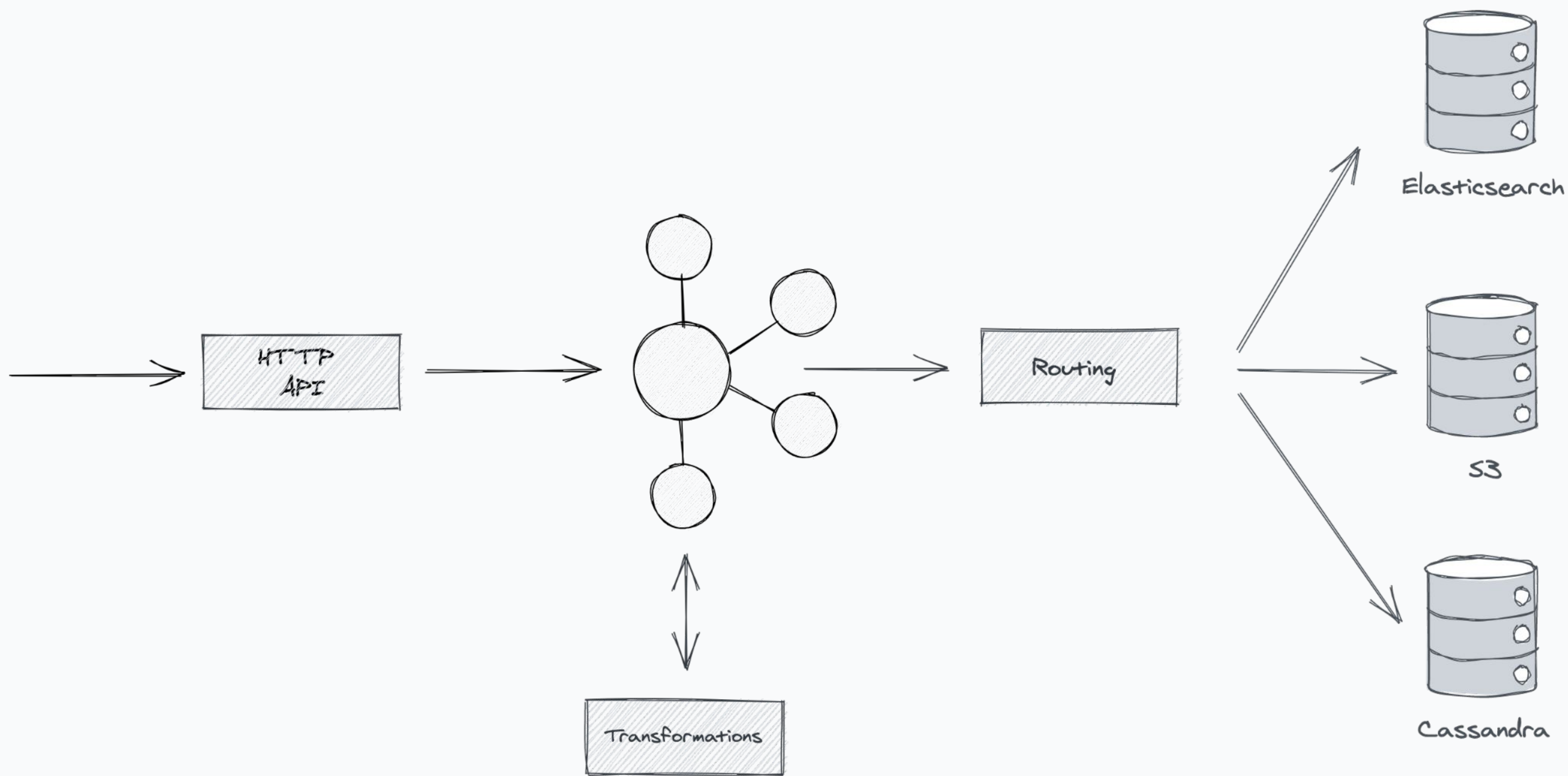
- Iceberg, Delta, Hudi
- Provide a transactional journal on top of the object store. Allow updates, compaction, even time travelling

```
FlinkSink.forRowData(input)
    .tableLoader(tableLoader)
    .overwrite(true)
    .hadoopConf(hadoopConf)
    .build()
```

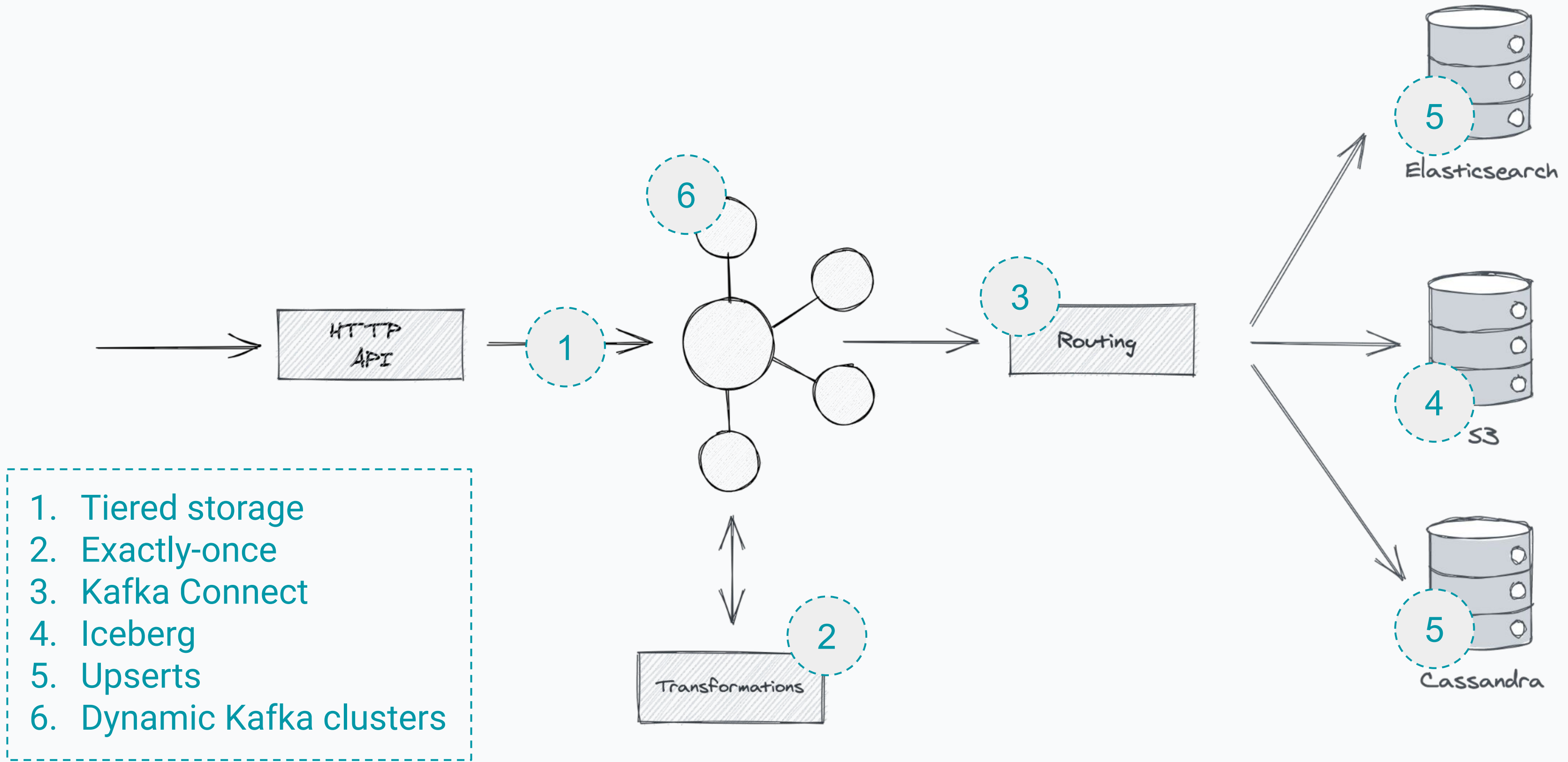
Addressing K Concerns

Addressing Concerns

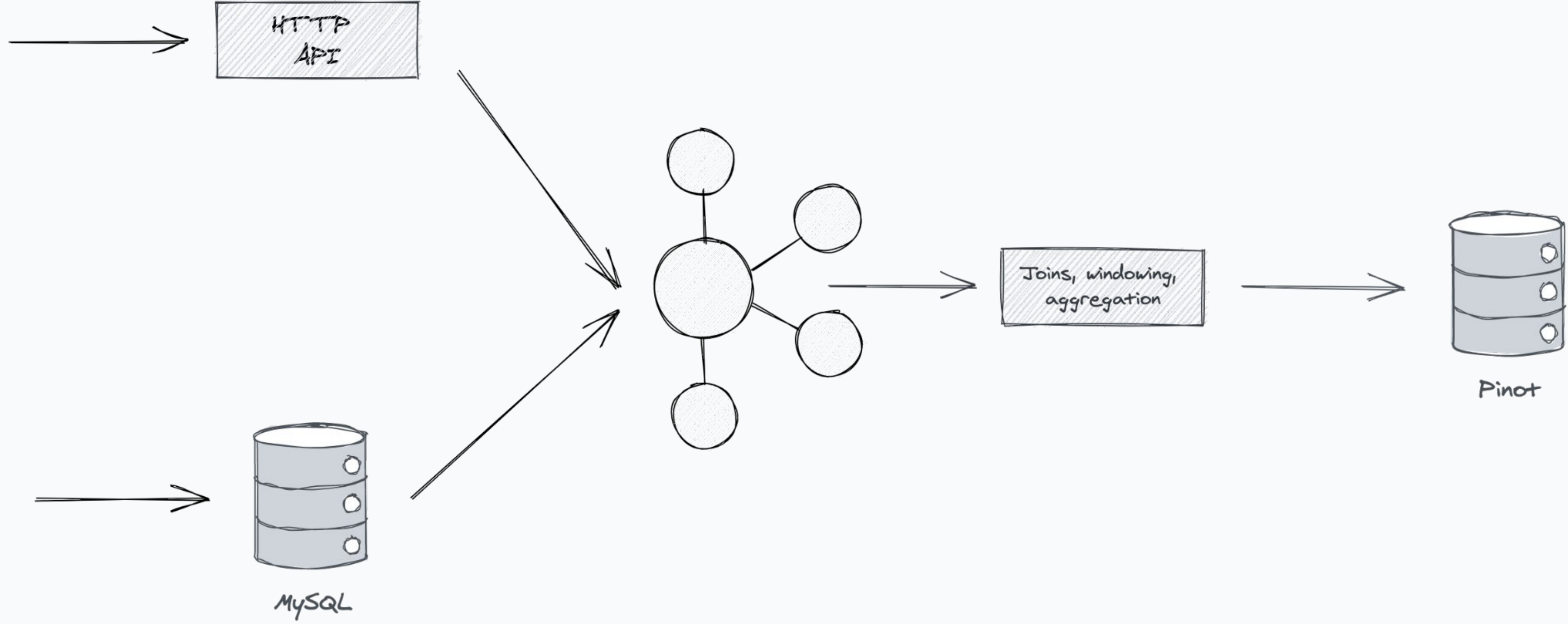
- Dava availability / retention
  - Data integration, compacted topics and tiered storage
- Data consistency
  - Exactly-once end-to-end delivery semantics
- Handling late-arriving data
  - State management, proper data sinks
- Data reprocessing & backfill
  - Dynamic Kafka clusters, Savepoints, State Processor API



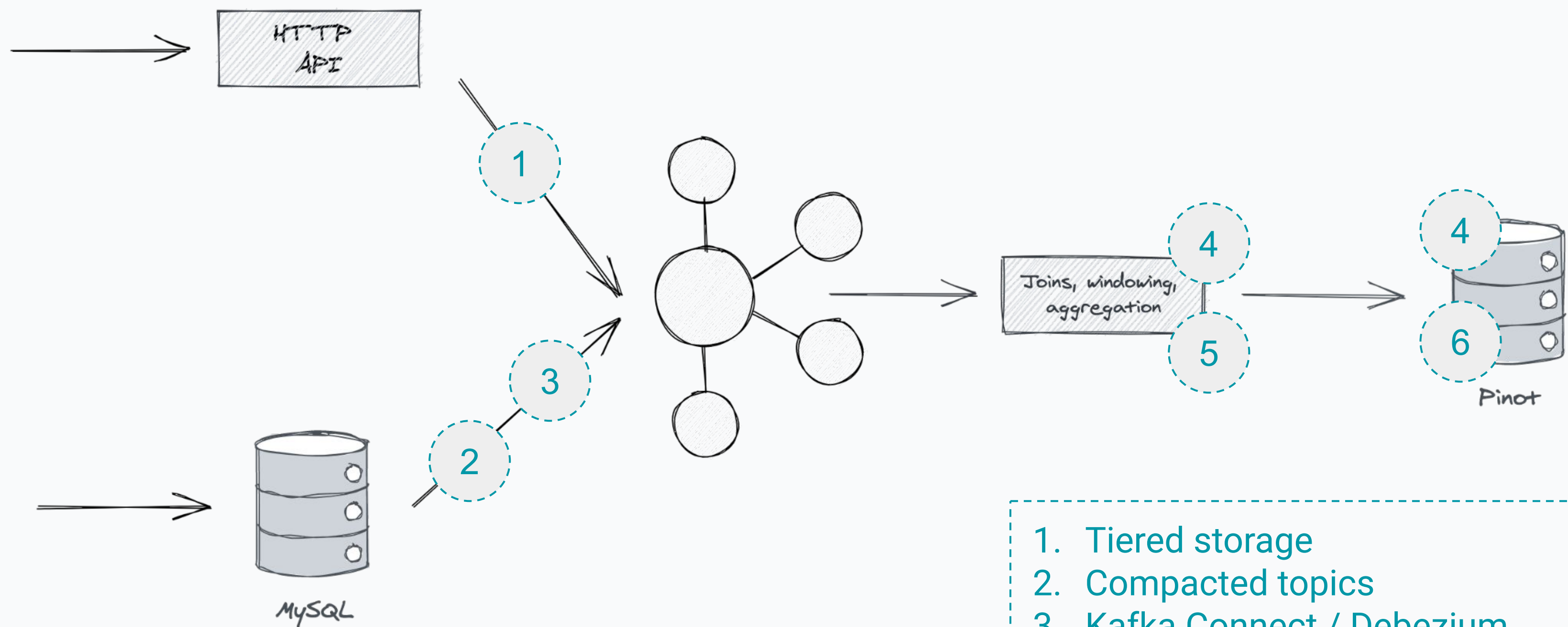
Use-case 1: **stateless** transformations, routing and integration



Use-case 1: **stateless** transformations, routing and integration



Use-case 2: **stateful** transformations, analytics



Use-case 2: **stateful** transformations, analytics

Questions?

@sap1ens