Streaming Pipelines in Kubernetes Using Apache Pulsar, Heron and BookKeeper

Karthik Ramasamy

Cofounder

Streamlio

Information Age

Event Data is key

Increasingly Connected World

Internet of Things
30 B connected devices by 2020

Connected Vehicles
Data transferred per vehicle per month
4 MB -> 5 GB

Health Care
153 Exabytes (2013) -> 2314 Exabytes (2020)

Digital Assistants (Predictive Analytics)
$2B (2012) -> $6.5B (2019) [1]
Siri/Cortana/Google Now

Machine Data
40% of digital universe by 2020

Augmented/Virtual Reality
$150B by 2020 [2]
Oculus/HoloLens/Magic Leap

Traditional Batch Processing

Challenges

- Introduces too much “decision latency”
- Responses are delivered “after the fact”
- Maximum value of the identified situation is lost

- Decisions are made on old and stale data

- Data at Rest

The New Era: Streaming Data/Fast Data

- Events are analyzed and processed in real-time as they arrive
- Decisions are timely, contextual and based on fresh data
- Decision latency is eliminated

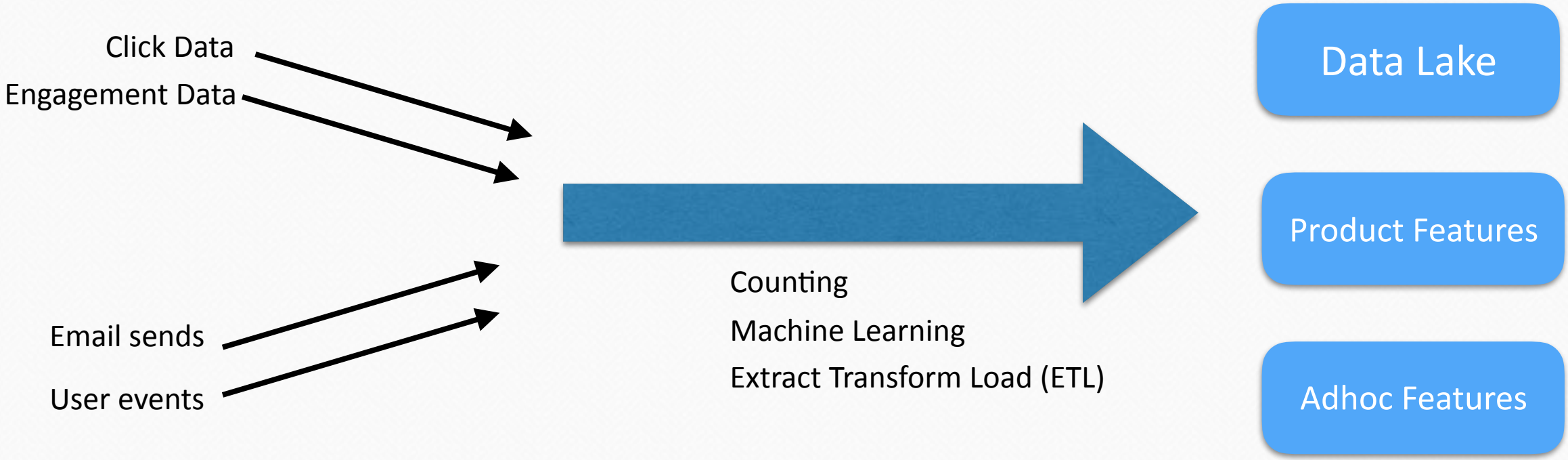
- Data in motion

Modern Data Pipelines  Streaming Data Pipelines

Streaming Data Pipeline

A **Streaming Data Pipeline** captures events for **analysis, process** as they **arrive** and produce **continuous** results

Streaming Data Pipeline



Use Cases

Product Safety

Observations

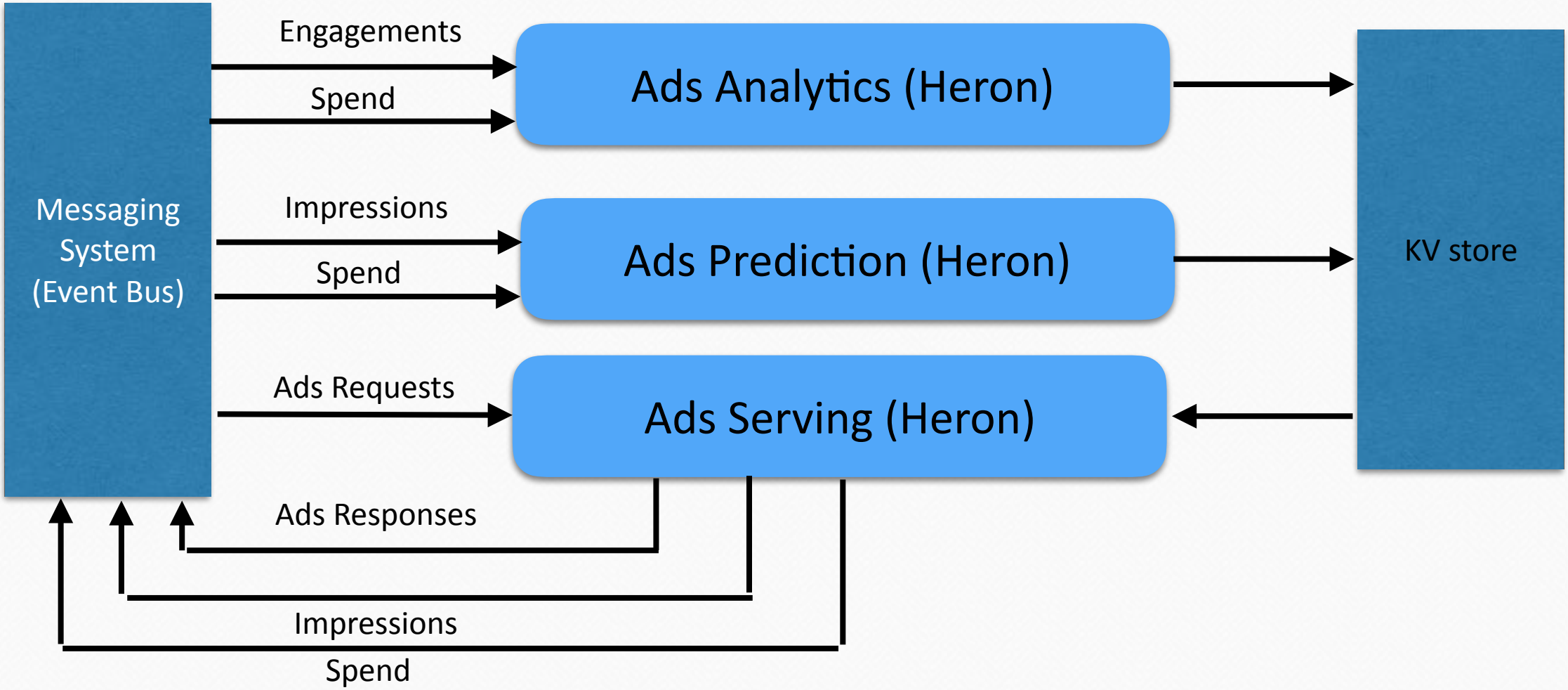
- Fight spammy content, engagements, and behaviors in Twitter
- Spam campaign comes in large batch
- Despite randomized tweaks, enough similarity among spammy entities are preserved

Requirement

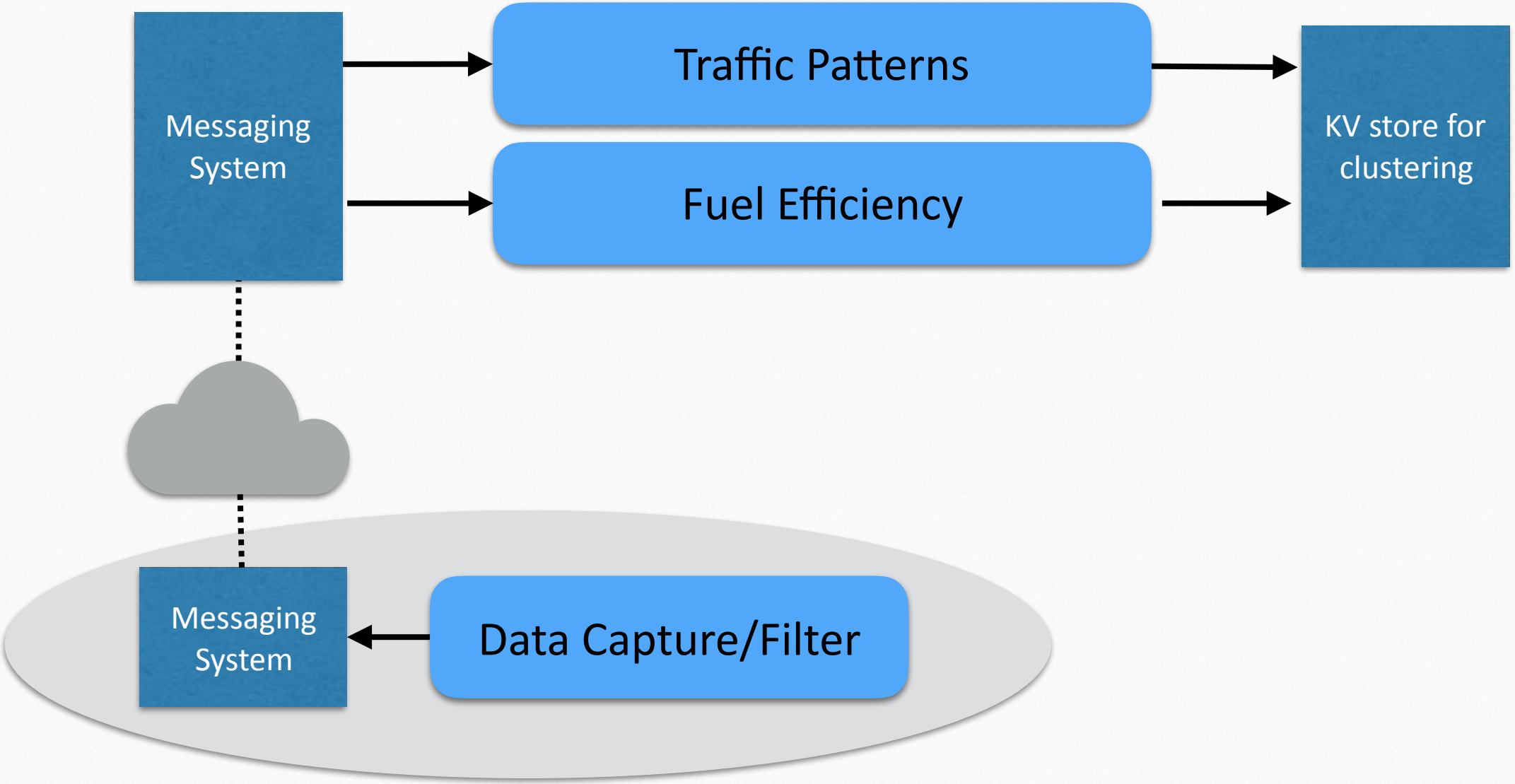
- Real time - a competition with spammers (i.e) “detect” vs “mutate”
- Generic - need to support all common feature representations

Product Safety - System Overview

Real Time Ads



Connected Cars

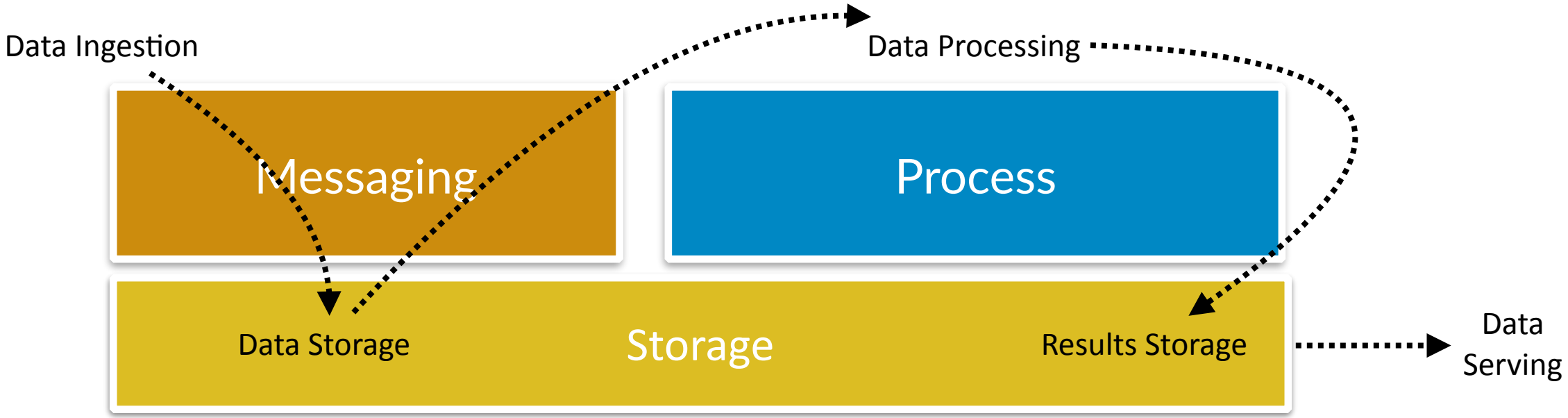


Streaming Use Cases

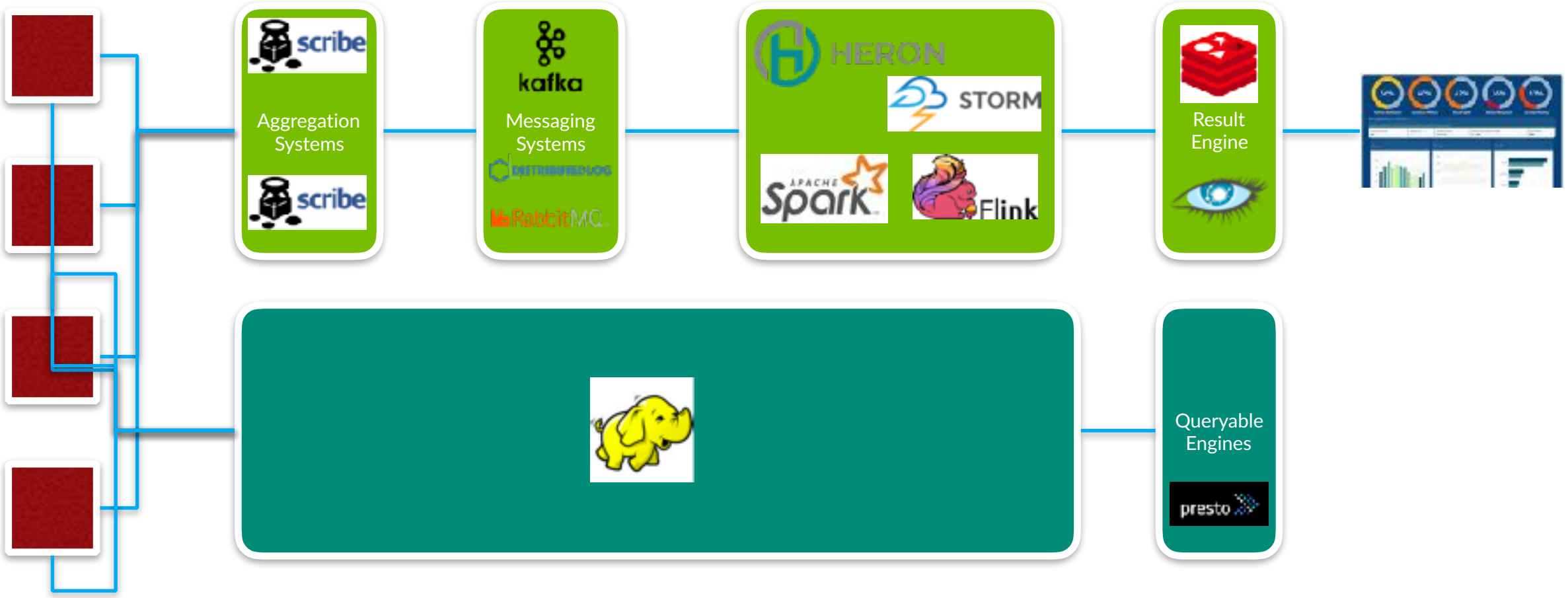
- Algorithmic trading
- Online fraud detection
- Geo fencing
- Proximity/location tracking
- Intrusion detection systems
- Traffic management
- Real time recommendations
- Churn detection
- Internet of things
- Social media/data analytics
- Gaming data feed
- IoT Edge Analytics

Architectural Pattern

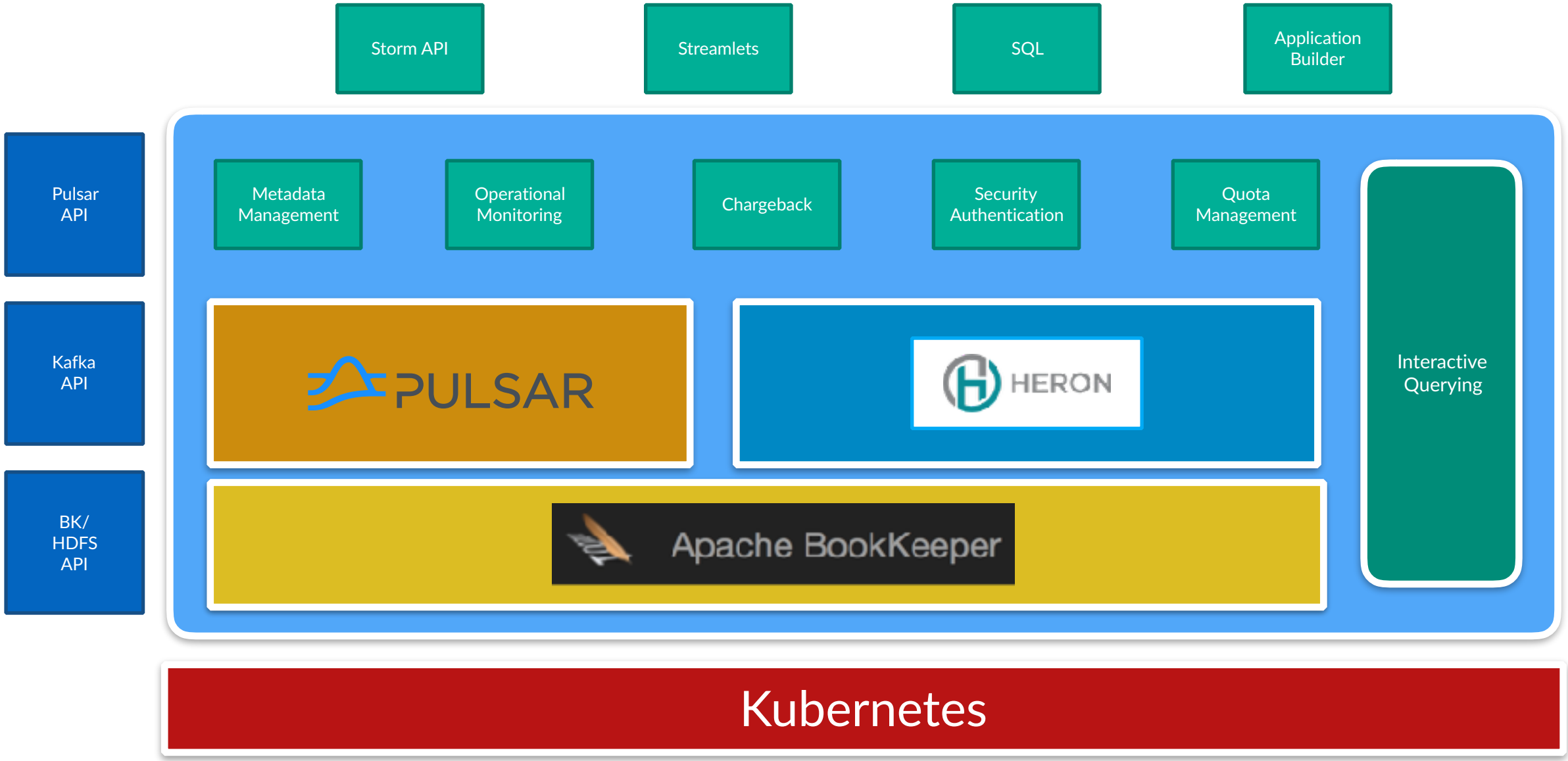
Recurring Pattern



State of the World



Towards Unification and Simplification



Apache Pulsar

Why Apache Pulsar?

Durability
Data replicated and synced to disk

Ordering
Guaranteed ordering

Delivery Guarantees
At least once, at most once and effectively once

Geo-replication
Out of box support for geographically distributed applications

Multi-tenancy
A single cluster can support many tenants and use cases

Low Latency
Low publish latency of 5ms at 99pct

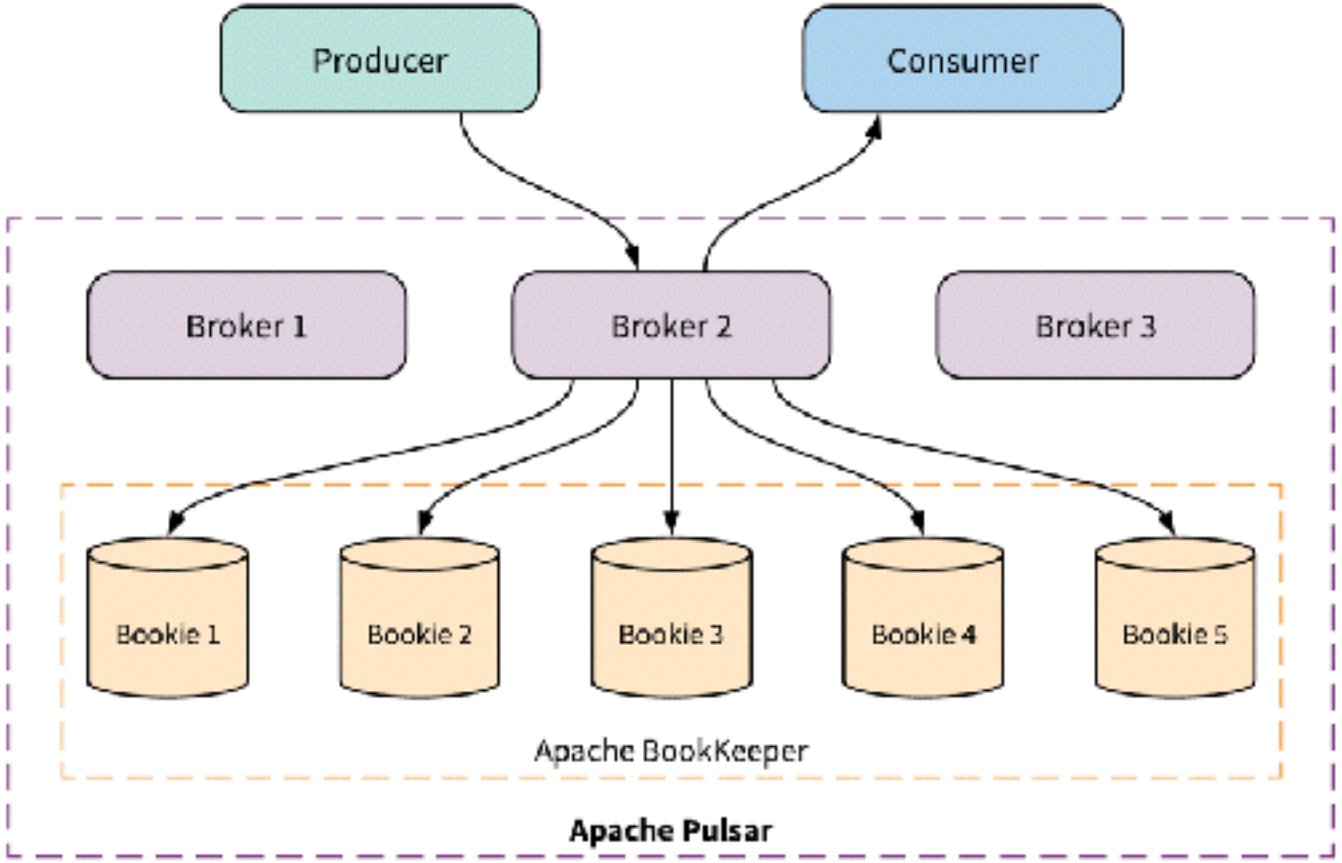
Unified messaging model
Support both Topic & Queue semantic in a single model

High throughput
Can reach 1.8 M messages/s in a single partition

Highly scalable
Can support millions of topics

Pulsar Architecture

Separation of Storage and Serving



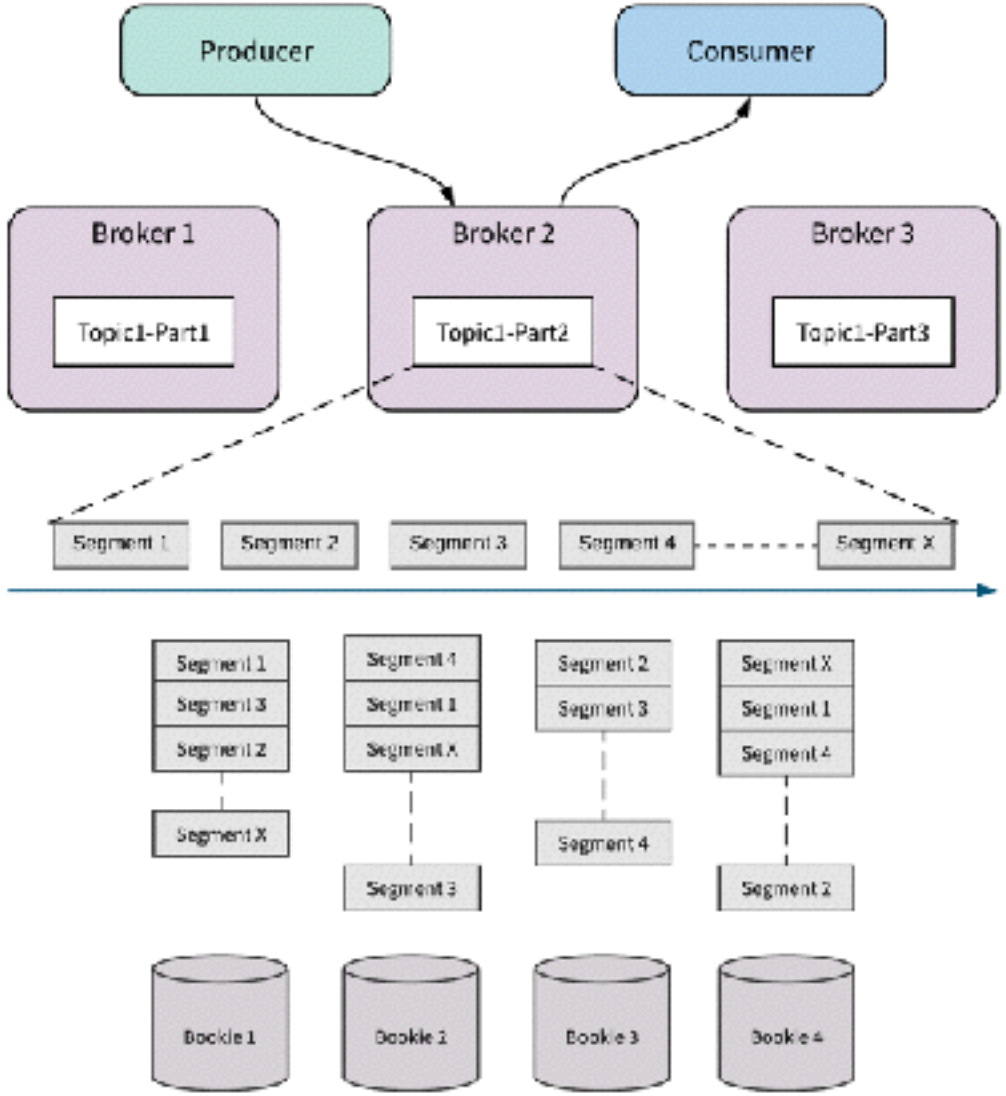
SERVING

Brokers can be added independently
Traffic can be shifted quickly across brokers

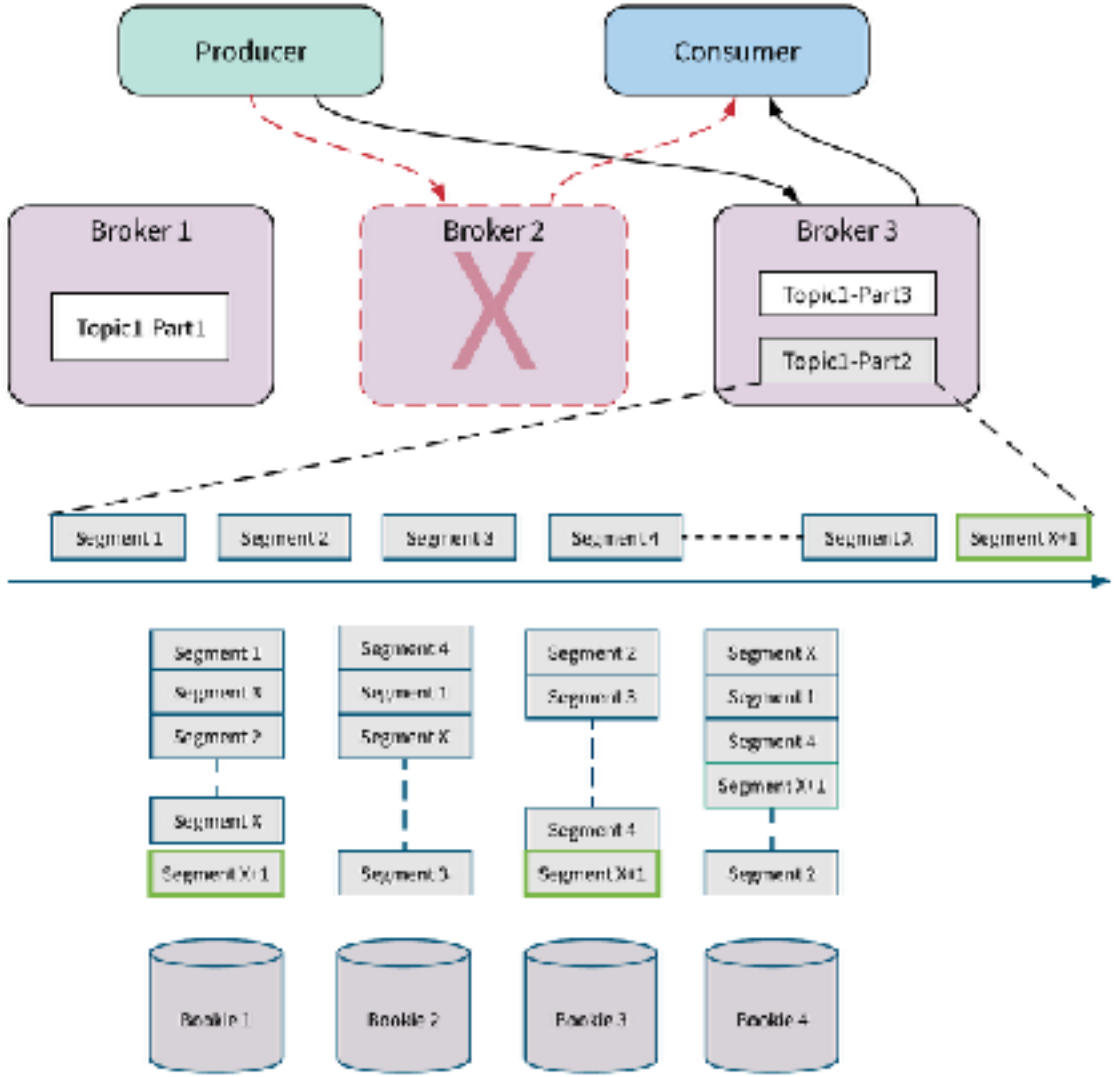
STORAGE

Bookies can be added independently
New bookies will ramp up traffic quickly

Segment Centric Architecture

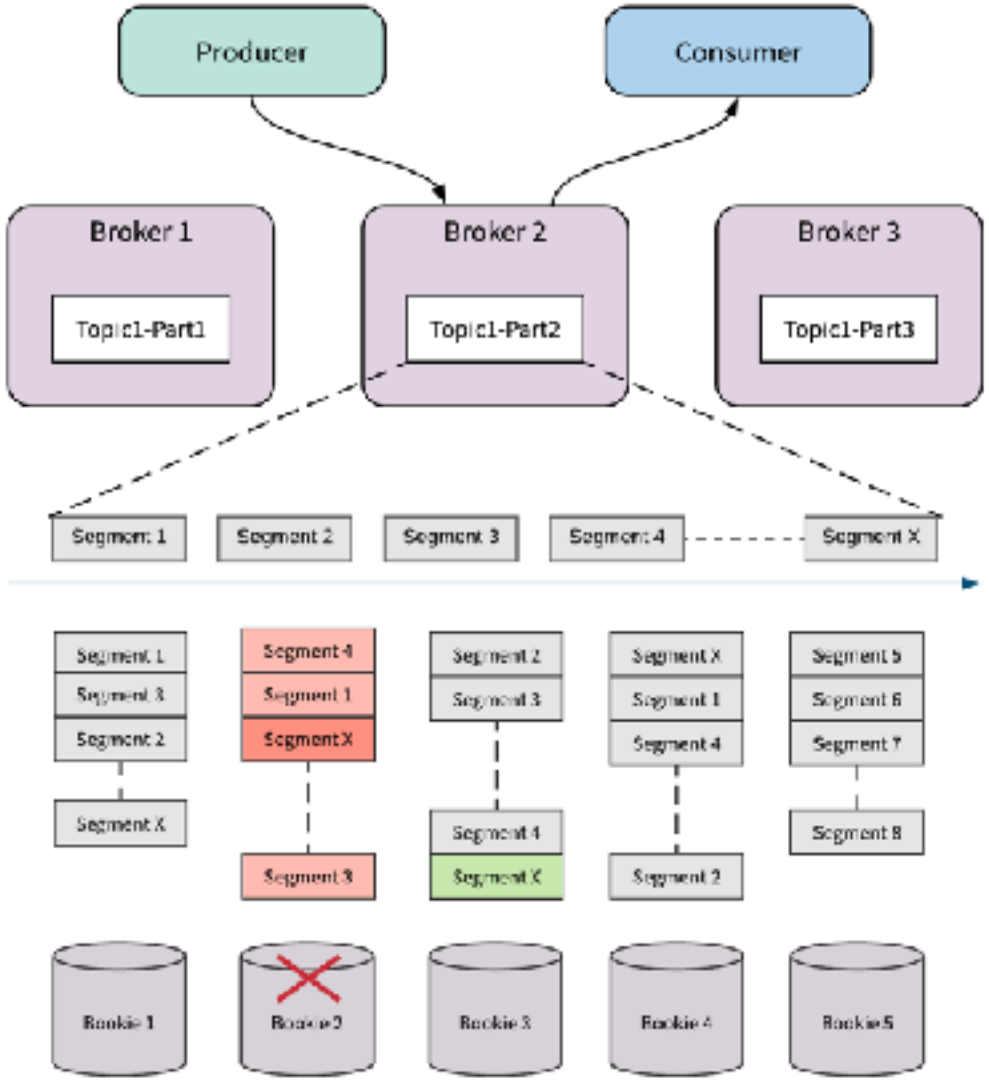


Broker Failure Recovery



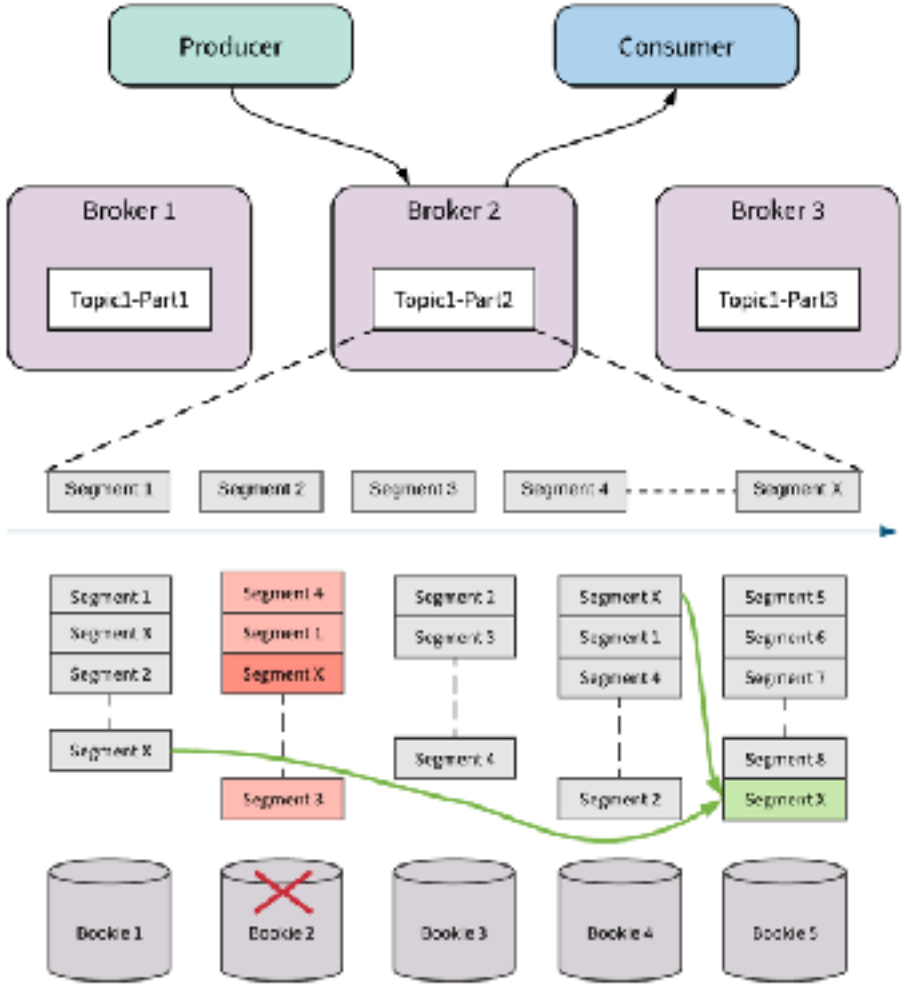
- Topic is reassigned to an available broker based on load
- Can reconstruct the previous state consistently
- No data needs to be copied
- Failover handled transparently by client library

Bookie Failure Recovery



- After a write failure, BookKeeper will immediately switch write to a new bookie, within the *same segment*.

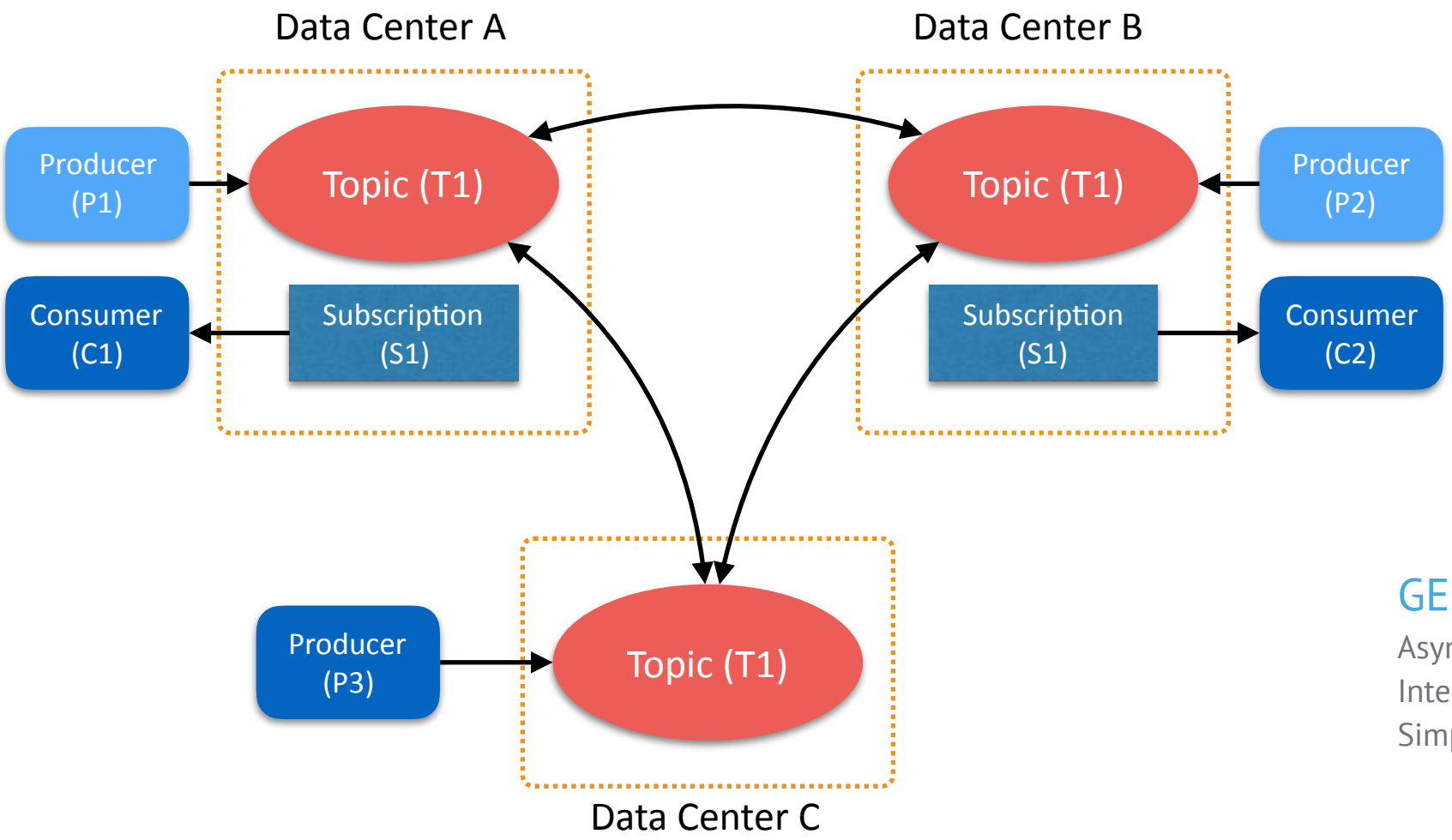
Bookie Failure Recovery



- In background, starts a many-to-many recovery process to regain the configured replication factor

Pulsar Architecture

Geo Replication



GEO REPLICATION

Asynchronous replication
Integrated in the broker message flow
Simple configuration to add/remove regions

Pulsar in Production

- 3+ years
- Serves 2.3 million topics
- 100 billion messages/day
- Average latency < 5 ms
- 99% 15 ms (strong durability guarantees)
- Zero data loss
- 80+ applications
- Self served provisioning
- Full-mesh cross-datacenter replication - 8+ data centers

Apache Heron

Heron Design Goals

Native Multi-Language Support
C++, Java, Python

Efficiency
Reduce resource consumption

Use of containers
Runs in schedulers - Kubernetes & DCOS & many more

Task Isolation
Ease of debug-ability/isolation/profiling

Support for diverse workloads
Throughput vs latency sensitive

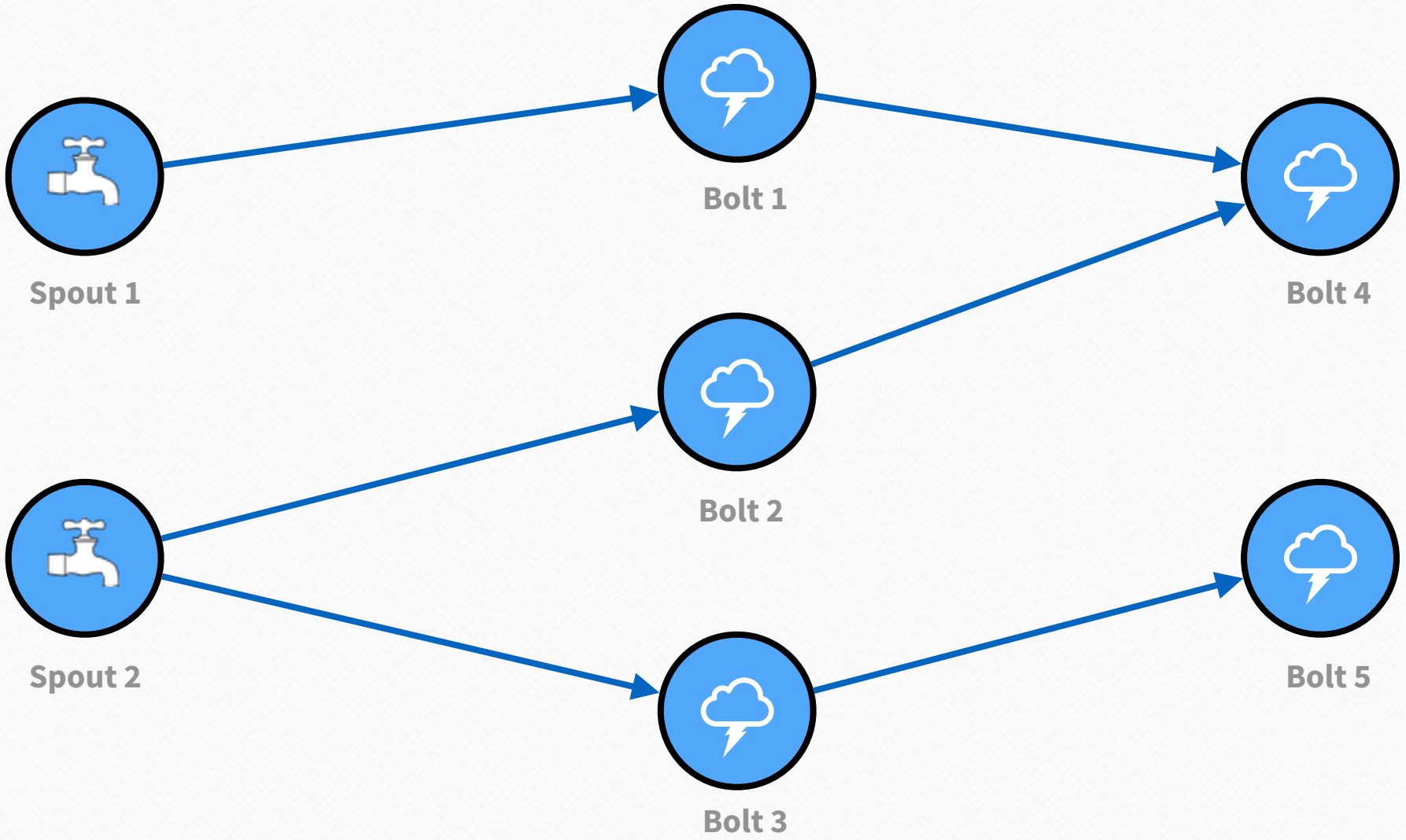
Multi-level APIs
Procedural, Functional and Declarative for diverse applications

Support for back pressure
Topologies should be self adjusting

Support for multiple semantics
Atmost once, Atleast once, Effectively once

Diverse deployment models
Run as a service or pure library

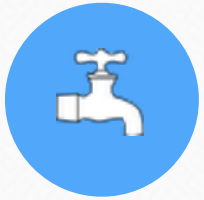
Heron Data Model



Writing Heron Topologies

Procedural - Low Level API

Directly write your spouts and bolts



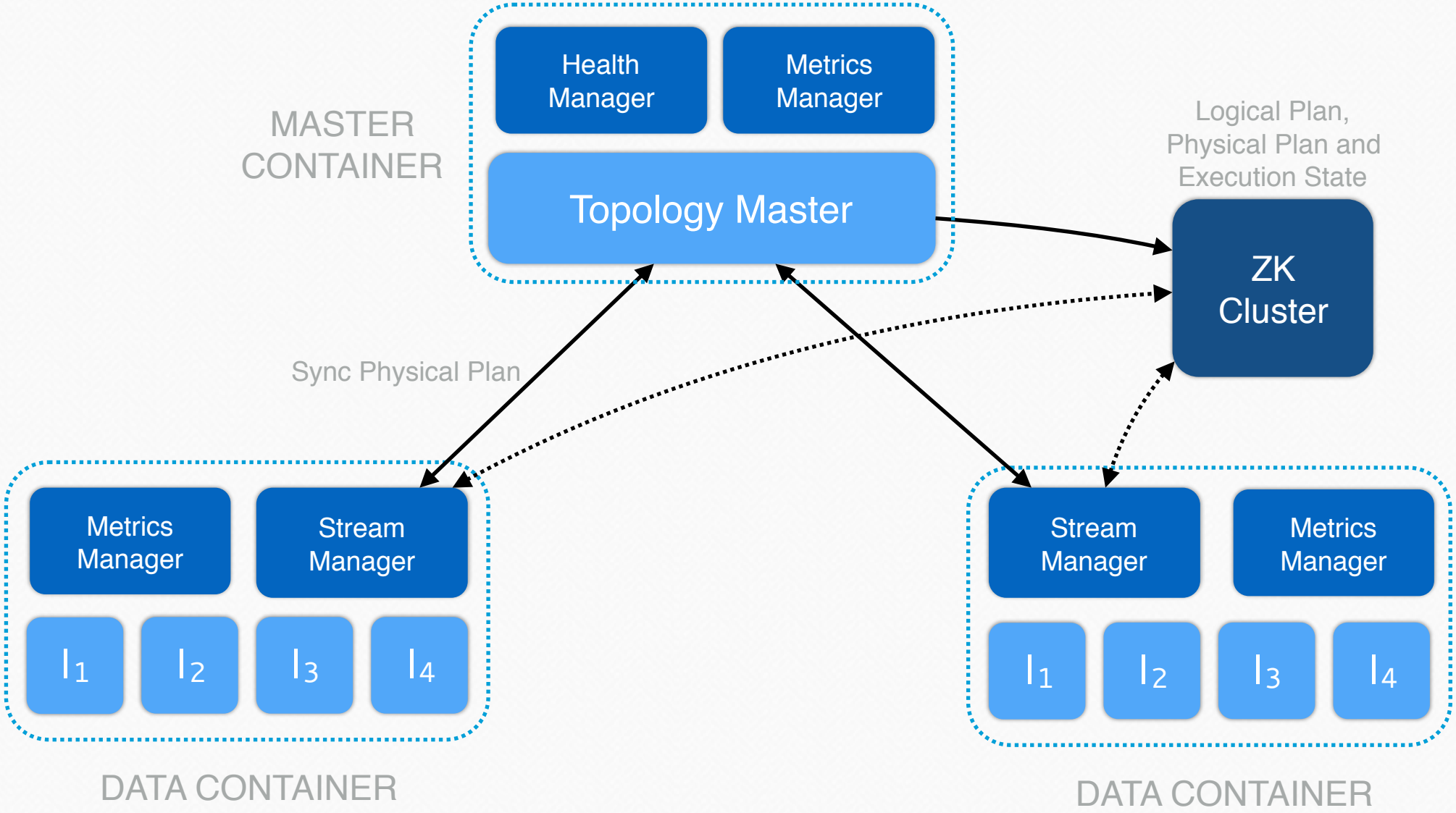
Functional - Mid Level API

Use of maps, flat maps, transform, windows

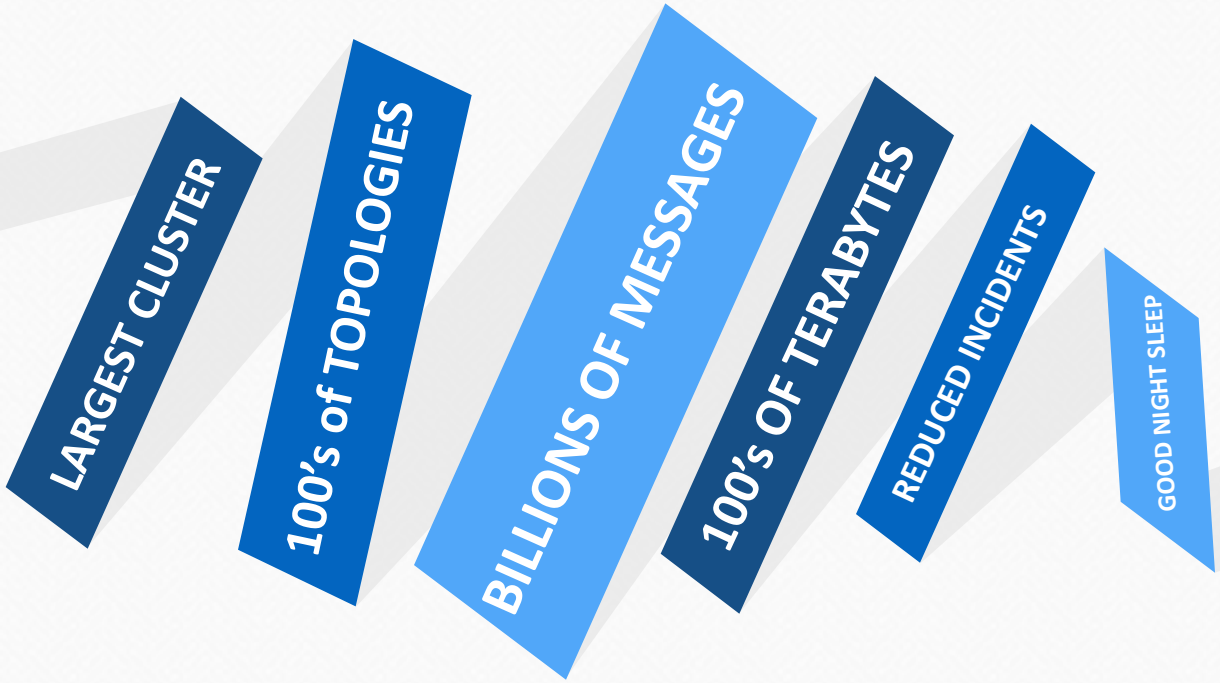
Declarative - SQL (in the works)

Use of declarative language - specify what you want, system will figure it out.

Topology Execution

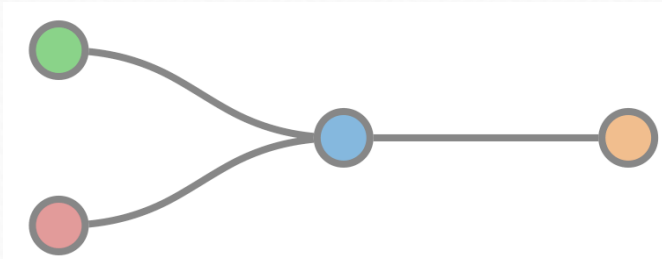


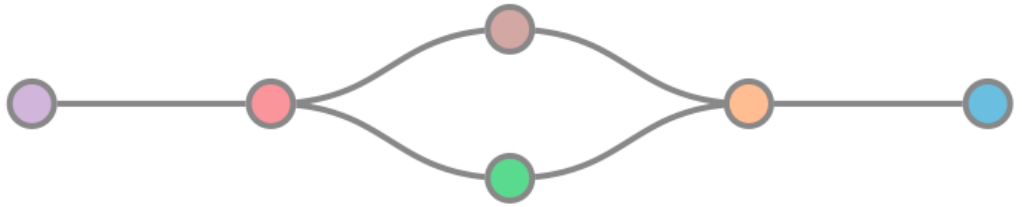
Heron @Twitter

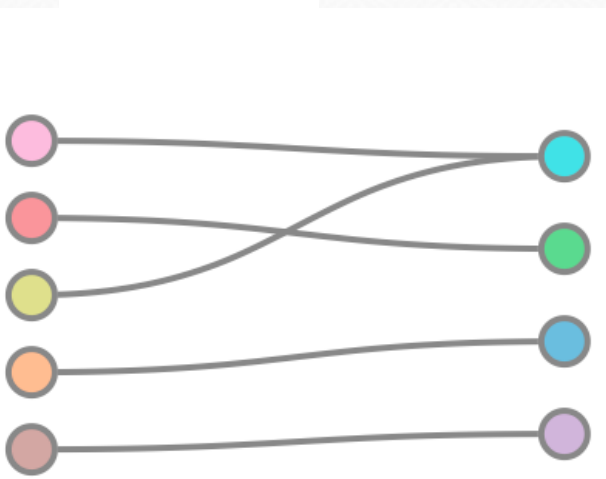


3X - 5X reduction in resource usage

Heron Topology Complexity







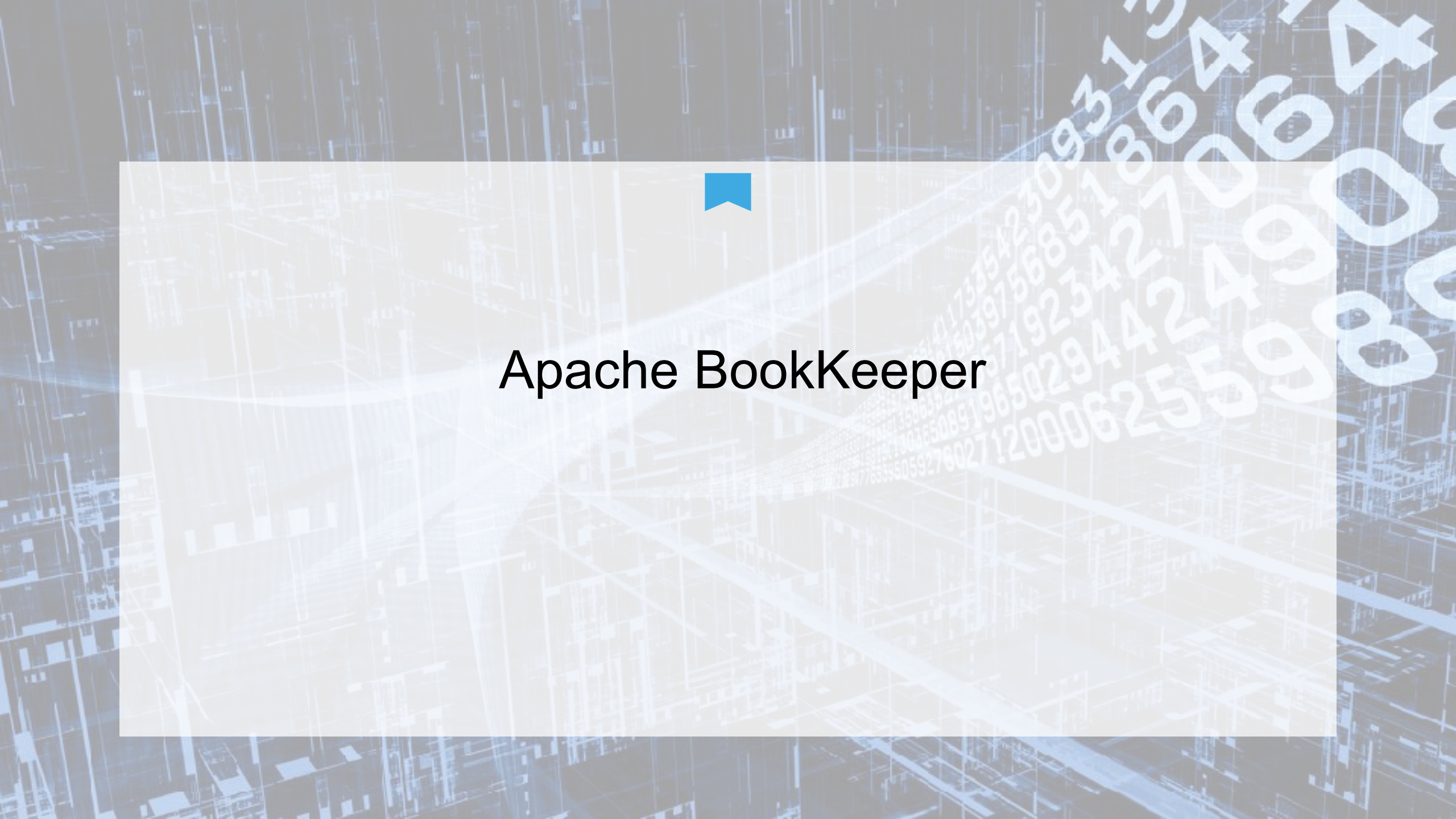
Heron Topology Scale

CONTAINERS - 1 TO 600

INSTANCES - 10 TO 6000

Heron Happy Facts :)

- ❖ No more pages during midnight for Heron team
- ➢ Very rare incidents for Heron customer teams
- ✓ Easy to debug during incident for quick turn around
- ▪ Reduced resource utilization saving cost



Apache BookKeeper

Observations

- Computation across batch/streaming is similar
  - Expressed as DAGS
  - Run in parallel on the cluster
  - Intermediate results need not be materialized
  - Functional/Declarative APIs
- Storage is the key
  - Messaging/Storage are two faces of the same coin
  - They serve the same data

Storage Requirements

Requirements for a real-time storage platform

- Be able to write and read streams of records with low latency, storage durability
- Data storage should be durable, consistent and fault tolerant
- Enable clients to stream or tail ledgers to propagate data as they’re written
- Store and provide access to both historic and real-time data

Apache BookKeeper - Stream Storage

A storage for log streams

- Replicated, durable storage of log streams
- Provide fast tailing/streaming facility
- Optimized for immutable data
- Low-latency durability
- Simple repeatable read consistency
- High write and read availability

Record

Smallest I/O and Address Unit

- A sequence of invisible records
- A record is sequence of bytes
- The smallest I/O unit, as well as the unit of address
- Each record contains sequence numbers for addressing

Logs

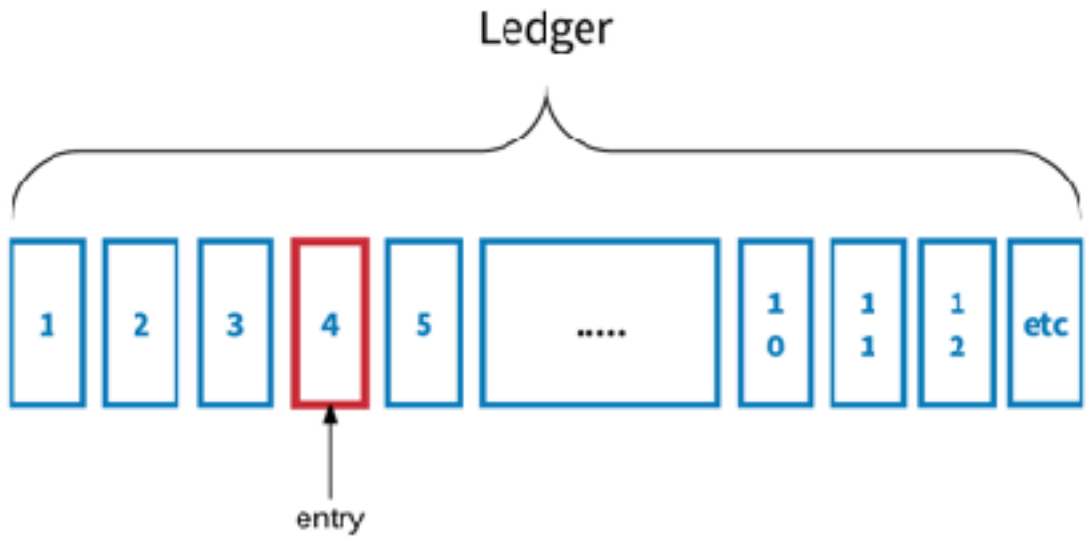
Two Storage Primitives

- Ledger: A finite sequence of records.
- Stream: An infinite sequence of records.

Ledger

Finite sequence of records

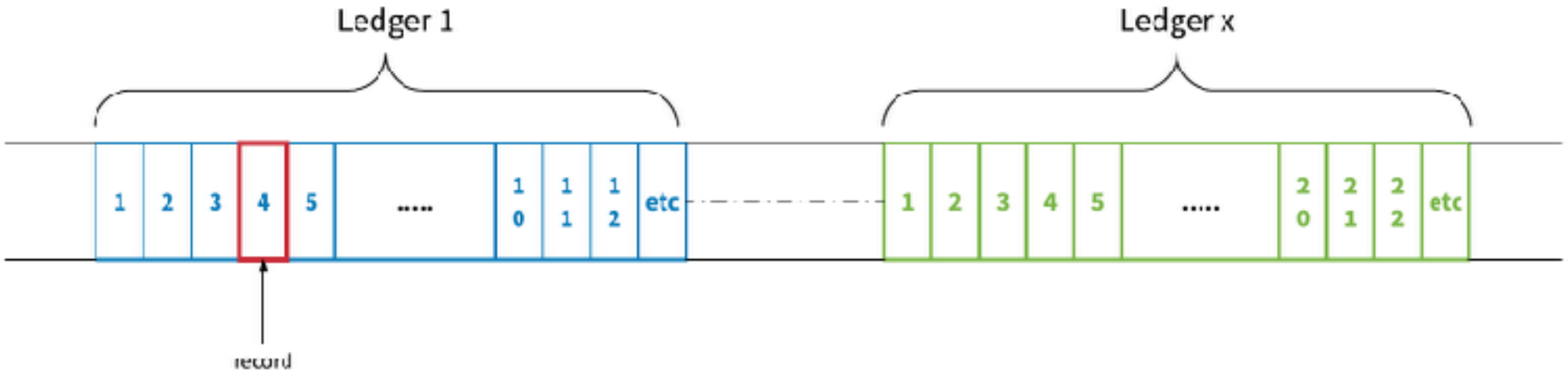
- Ledger: A finite sequence of records that gets terminated
  - A client explicitly close it
  - A writer who writes records into it has crashed.



Stream

Infinite sequence of records

- Stream: An unbounded, infinite sequence of records
  - Physically comprised of multiple ledgers



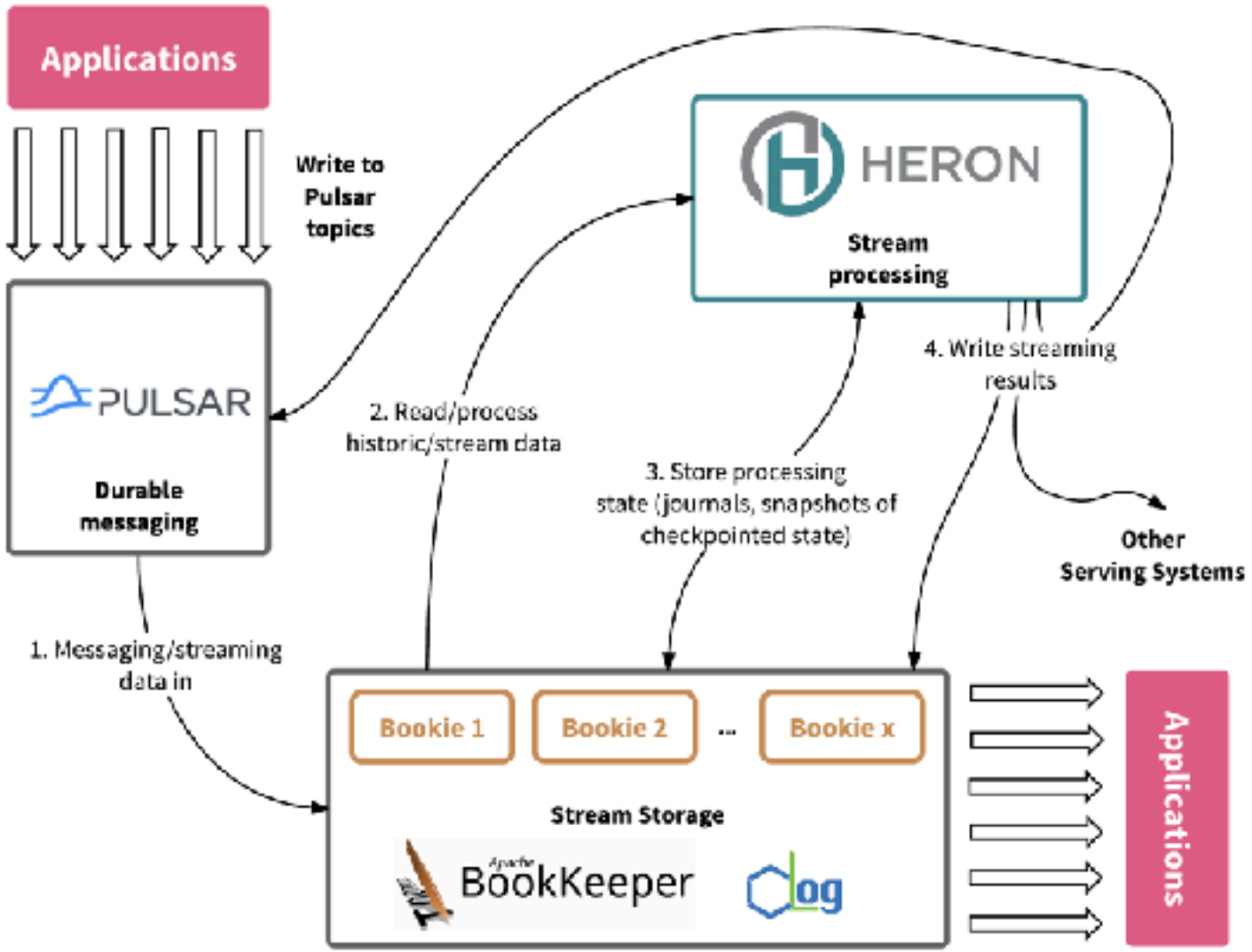
Bookies

Stores fragment of records

- Bookie - A storage server to store data records
- Ensemble: A group of bookies storing the data records of a ledger
- Individual bookies store fragments of ledgers

Using BookKeeper

Durable Messaging, Scalable Compute and Stream Storage



Companies using BookKeeper

Enterprise Grade Stream Storage

Companies using the projects

Enterprise Grade Stream Storage

Stanford University

INDIANA UNIVERSITY

Experiences with Kubernetes

First Implementation

- Use of naked pods for Heron containers
  - Never gets rescheduled
  - Have to create names similar to Heron container (topology-name-shard-id)
  - Have to keep track of every pod
- Cannot use deployment
  - Reproducible shard id
  - Deployments produce different ids after a pod failure
- Topology names have to be lowercase

Experiences with Kubernetes

Second Implementation

- Use of stateful sets
  - Nice mapping form stateful sets to container shard ids
- Initially used defaults
  - Very long startup time
  - Very long shutdown time
  - Long pod restart time (during machine failure & pod failure)

Experiences with Kubernetes

Second Implementation

- Investigation pointed out us to
  - Default pod management policy is set to OrderedReady
  - Default rescheduling pod on a node failure is set to 5 mins

- Solutions
  - Sets the tolerations to 10 seconds for restarts
  - node.kubernetes.io/not-ready
  - node.alpha.kubernetes.io/notReady
  - node.alpha.kubernetes.io/unreachable

Experiences with Kubernetes

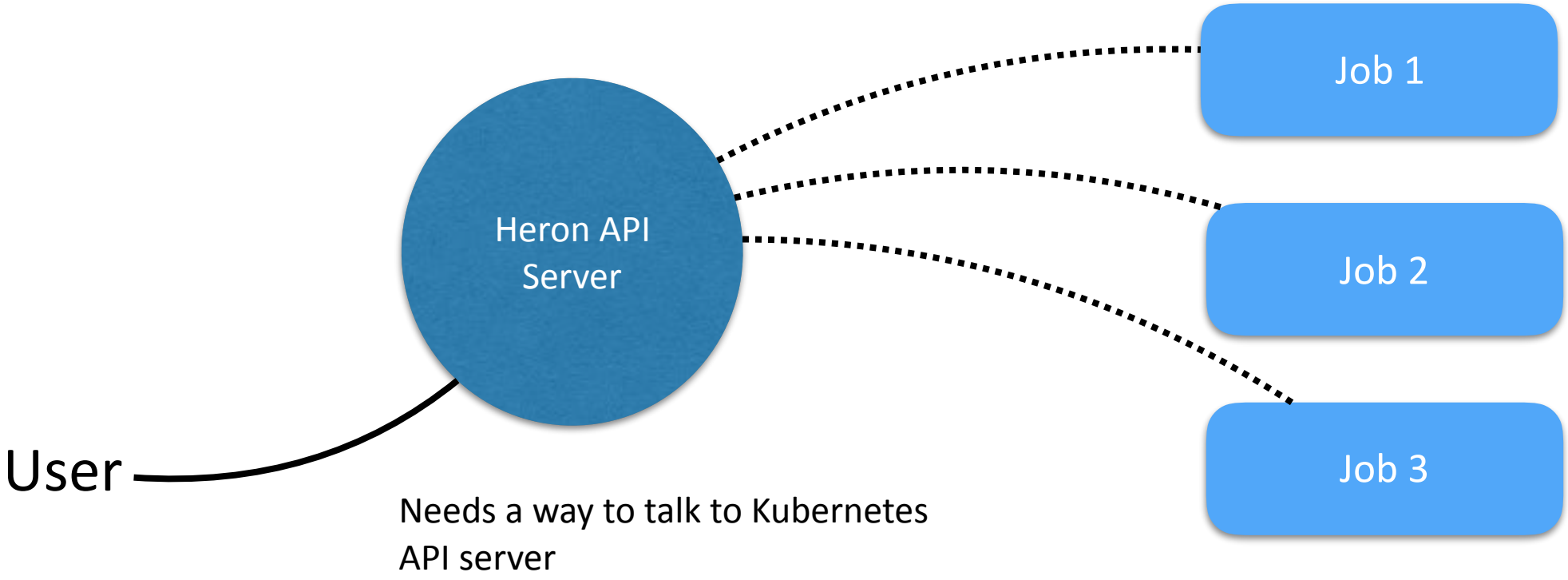
Second Implementation

- Dynamically updating the job
  - Patching stateful sets -
  - Kubernetes Java Client library has bug

- Solutions
  - Waiting for a fix!

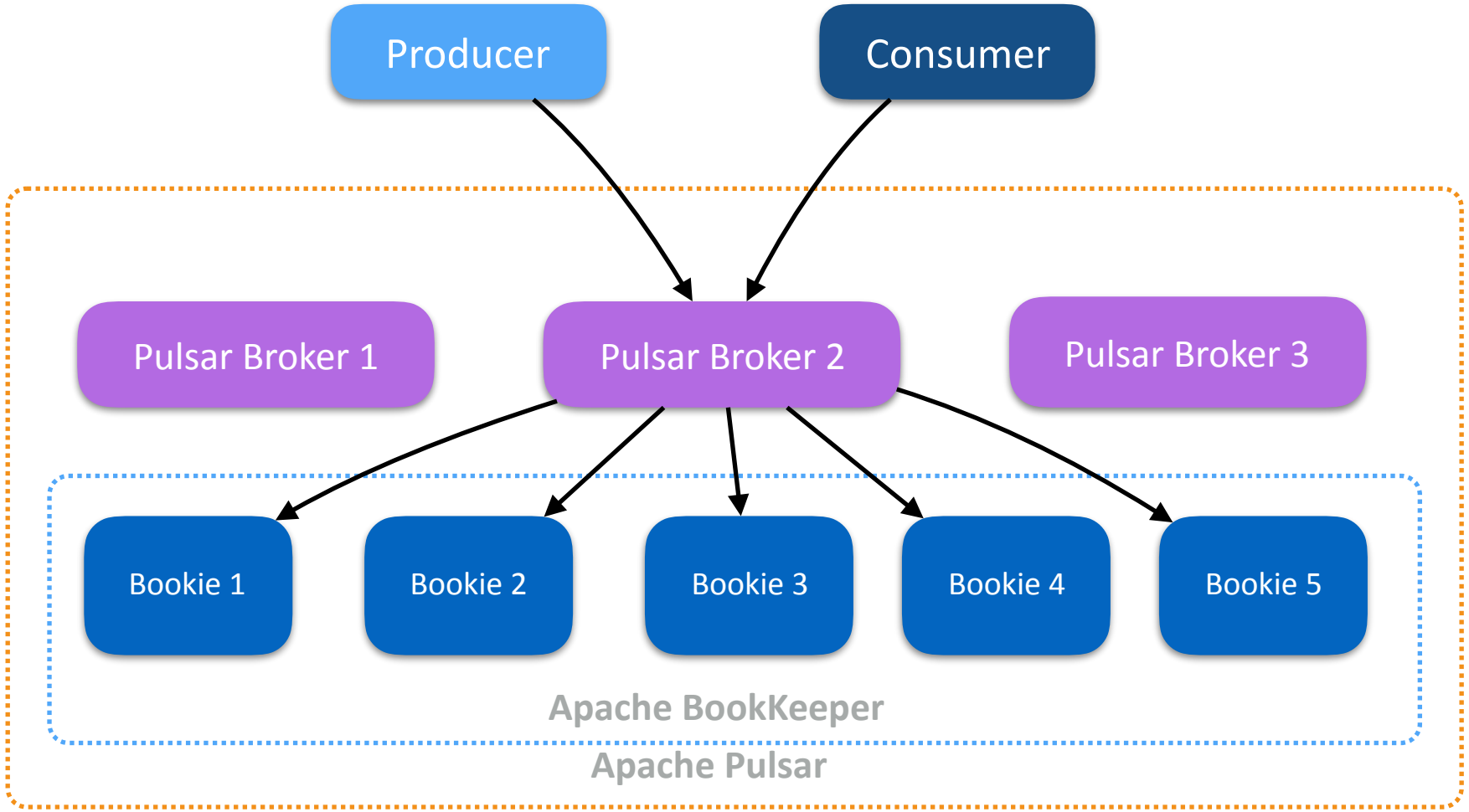
Experiences with Kubernetes

Submitting User jobs



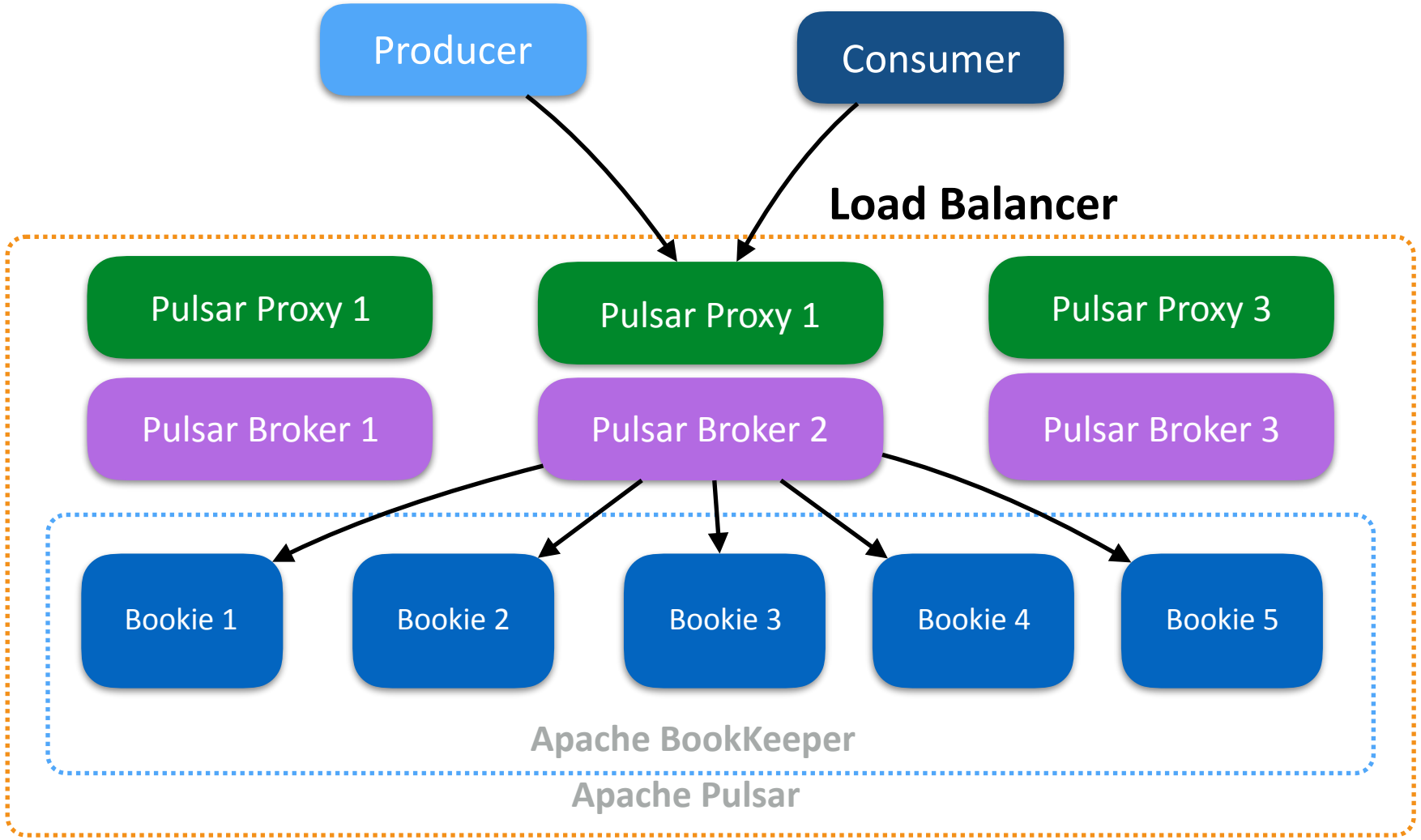
Solution - Run a slim side container with kubectl proxy

Ingesting Data to Pulsar

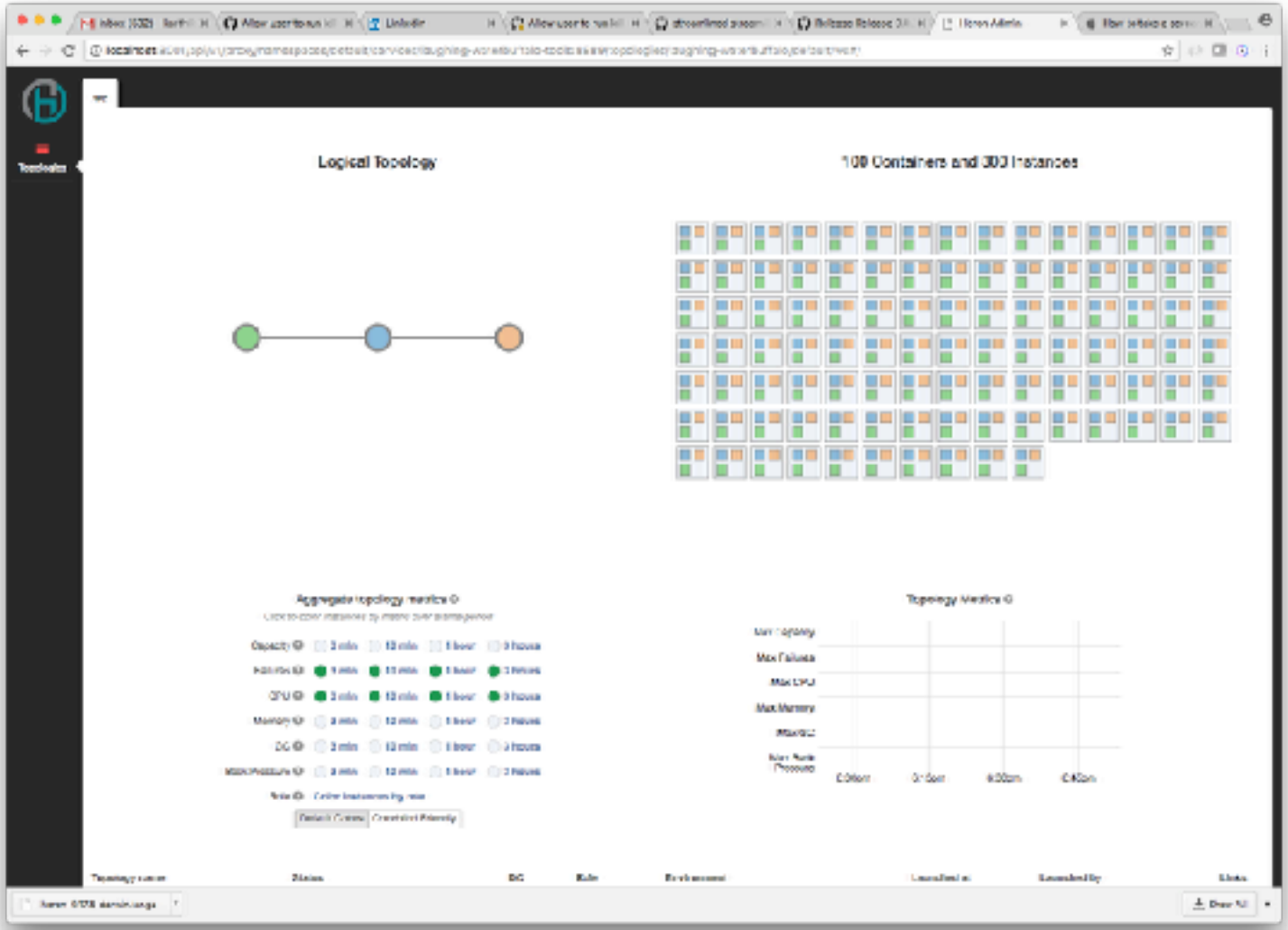


Internal IPs

Ingesting Data to Pulsar



Finally...




Q&A


WHAT
WHERE
HOW
?
WHEN
WHY
WHO

GET IN TOUCH

CONTACT US

@kramasamy

955 Alma Street, Palo Alto, CA

karthik@streaml.io

Thank you

Traditional Batch Processing

Challenges

- Introduces too much “decision latency”
- Responses are delivered “after the fact”
- Maximum value of the identified situation is lost

- Decisions are made on old and stale data

- Data at Rest

The New Era: Streaming Data/Fast Data

- Events are analyzed and processed in real-time as they arrive
- Decisions are timely, contextual and based on fresh data
- Decision latency is eliminated

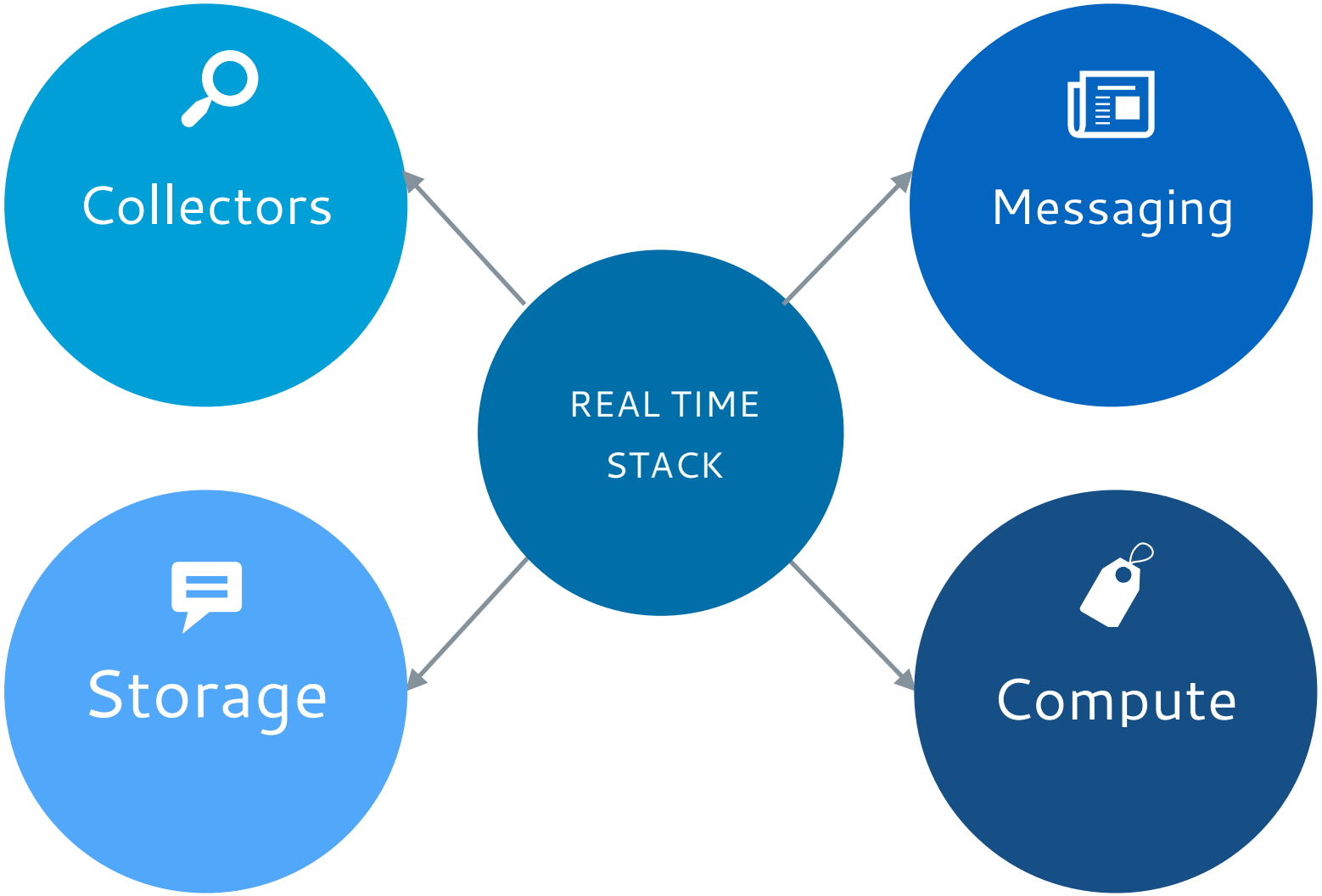
- Data in motion

Modern Data Pipelines  Streaming Data Pipelines

Streaming Data Pipeline

A **Streaming Data Pipeline** captures events for **analysis, process** as they **arrive** and produce **continuous** results

Streaming Stack



Towards Simplification & Unification