Aalto University
School of Science

Service and Integration Models in Big Data Platforms

Hong-Linh Truong
Department of Computer Science
linh.truong@aalto.fi

14.1.2026

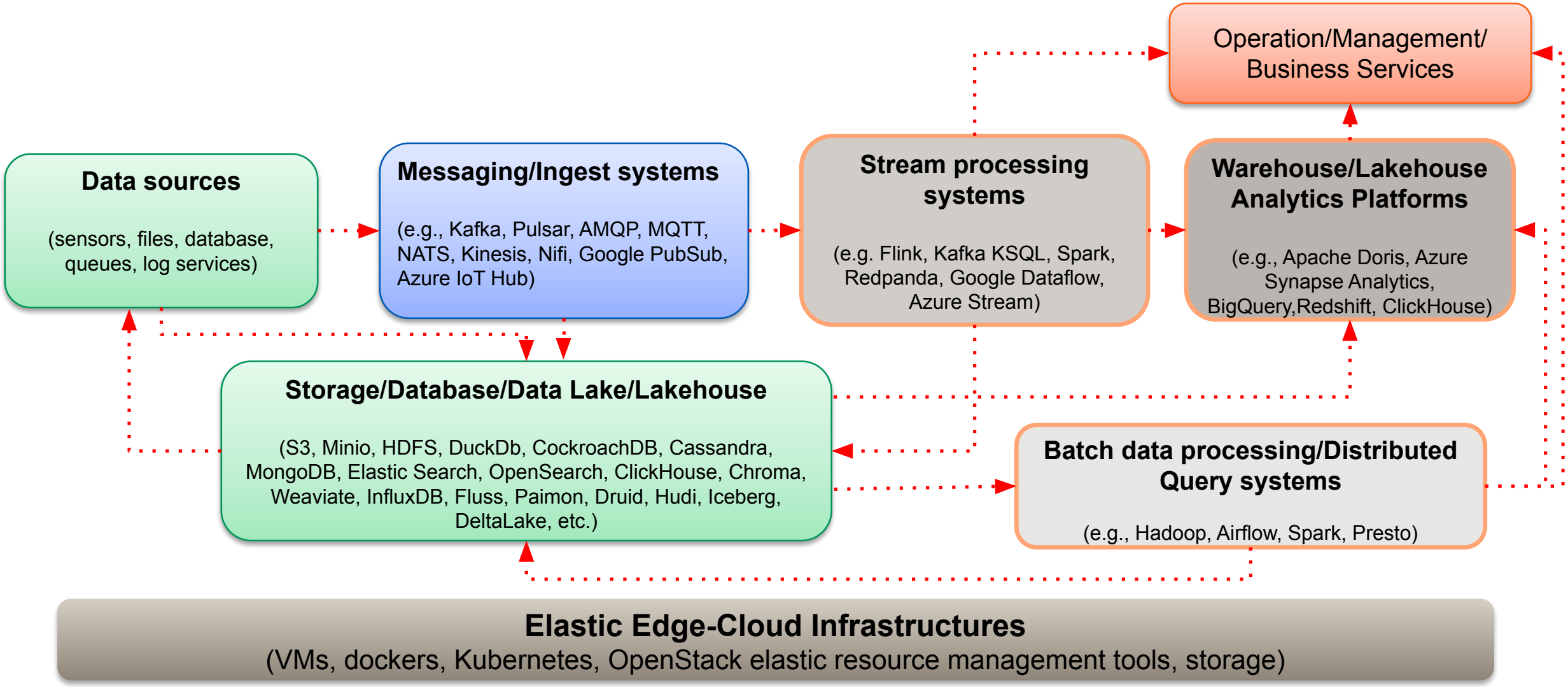
Learning objectives

- Understand common ways to bring data into platforms
- Study service requests and data partition for optimizing integration models
- Understand the role of service/data discovery and consensus
- Establish the links to follow-up lectures

Recall

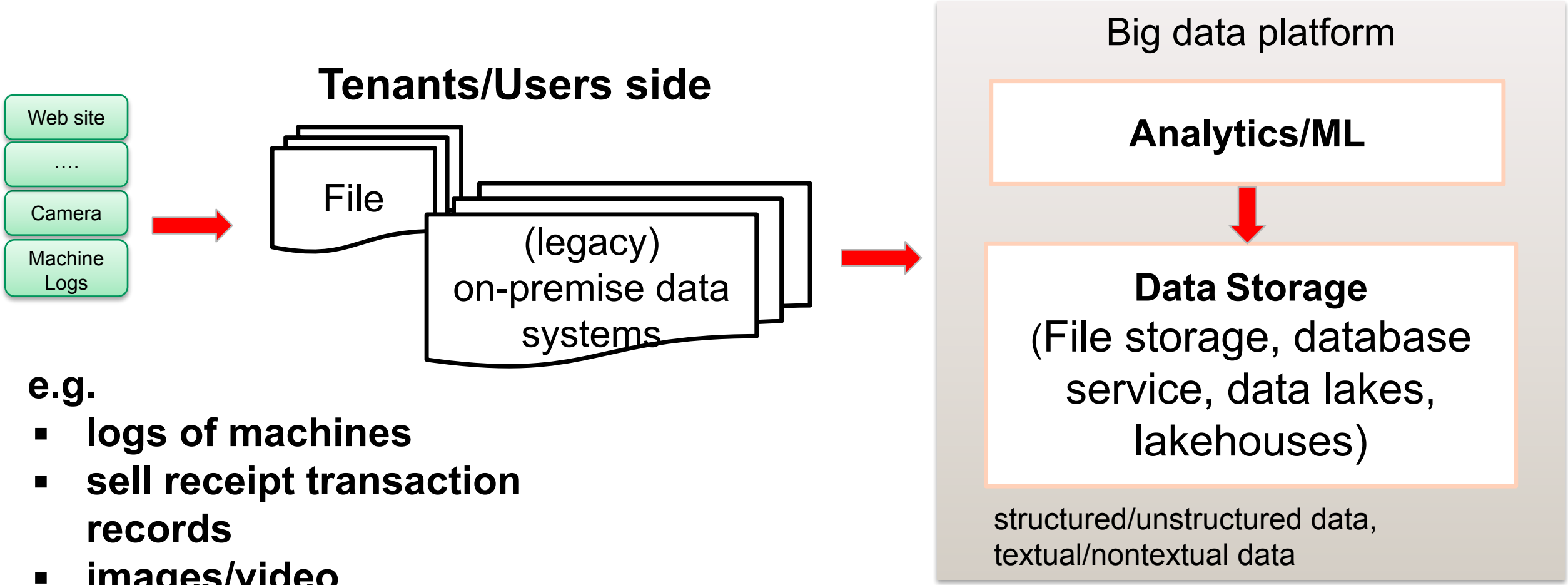
- Platforms must facilitate exchanges between many stakeholders centered around data products
- Platform services support many types of interactions with different protocols and APIs
- Some important aspects of interactions
  - APIs for encapsulating low-level details
  - protocols for interoperability
  - performance management
  - service/data discovery

Our big data at large-scale: the big picture in this course



Moving big data into the platform

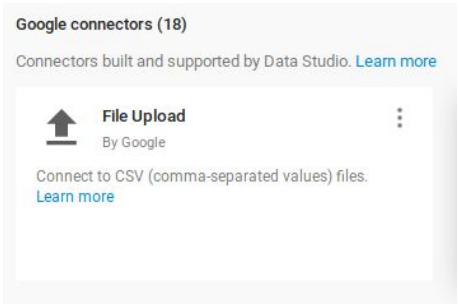
Bring files/static datasets/data sources into platforms

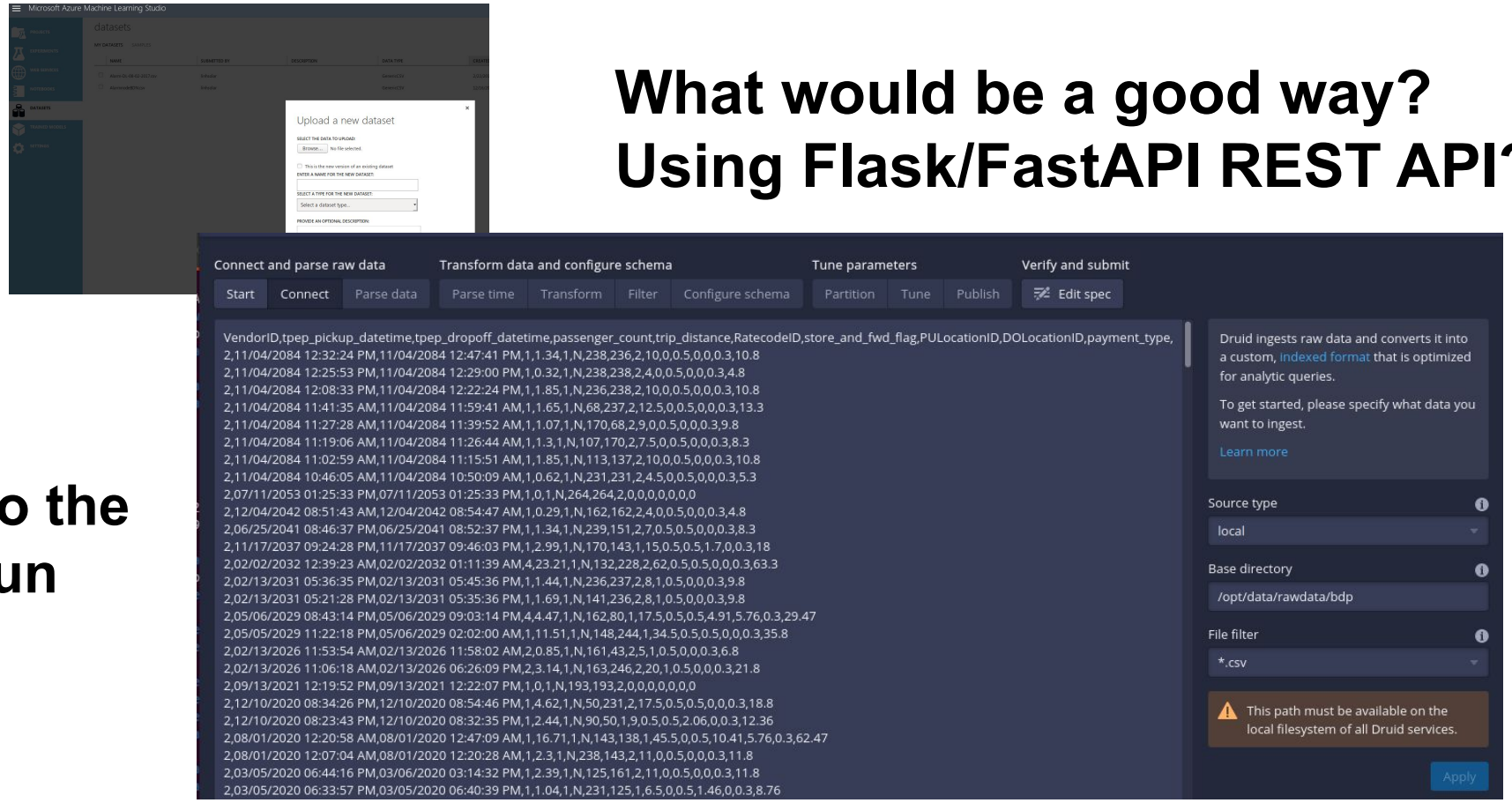


e.g.

- **logs of machines**
- **sell receipt transaction records**
- **images/video**

First obstacle: ingesting big data into cloud data storage/database services

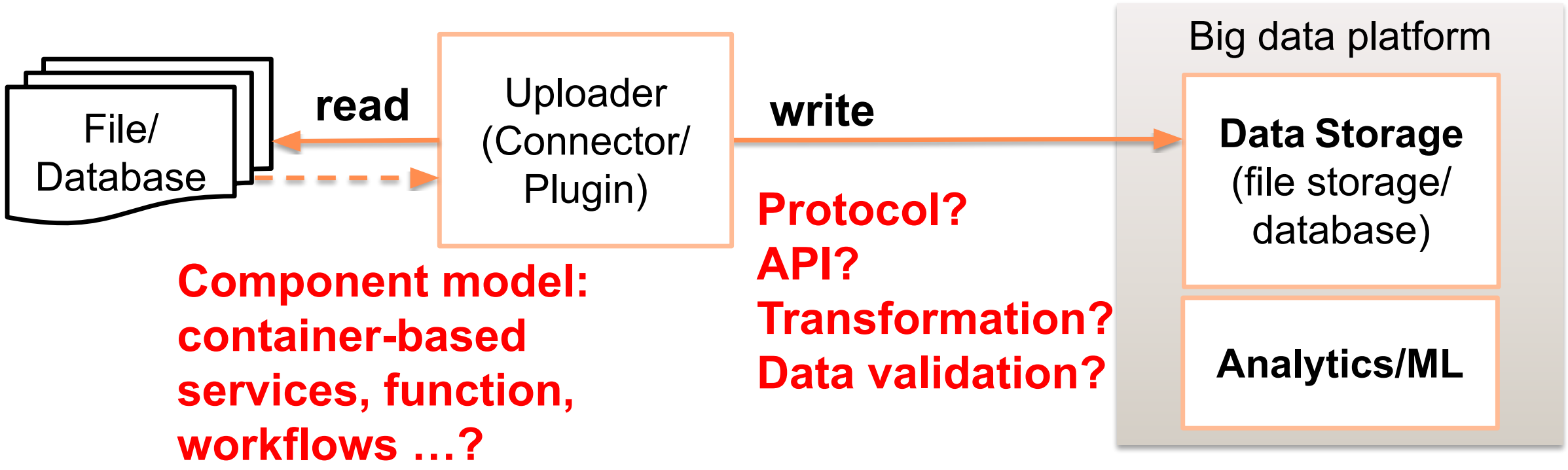




What would be a good way? Using Flask/FastAPI REST API?

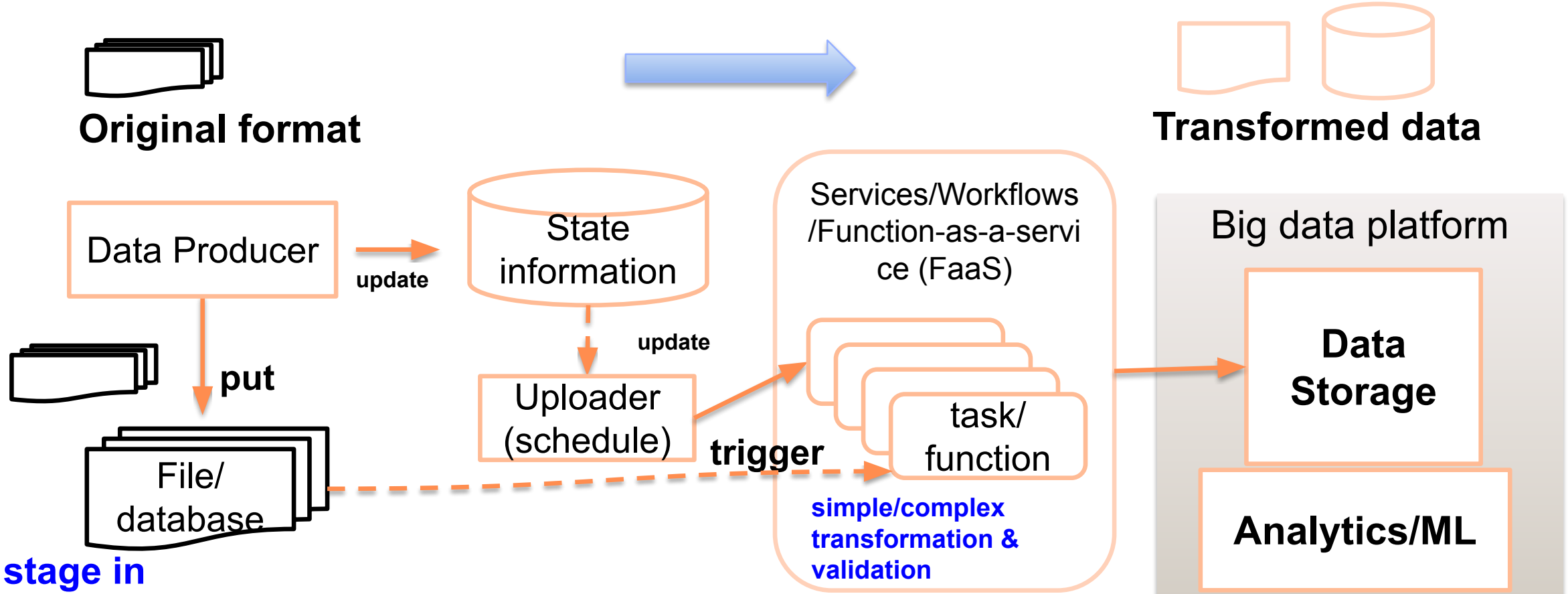
e.g., upload data into the cloud storage and run machine learning

Complex design details

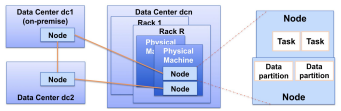


- **Practical issues for optimization:**
  - Handle very big files vs a lot of small files
  - Concurrent processing and distributed deployment
  - Different transformation/integration models

Complex design details: conceptual view



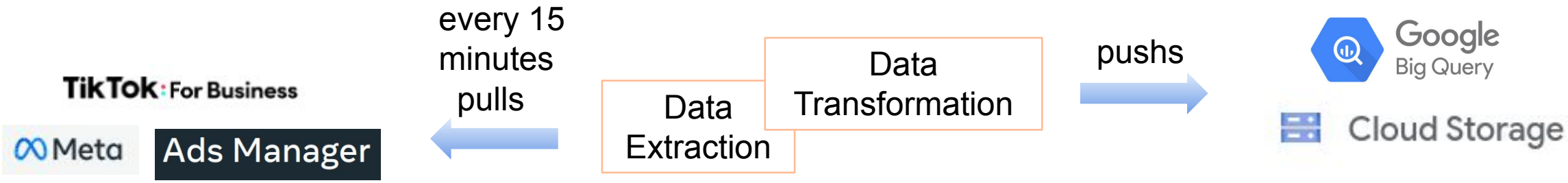
Where? next to data source or next to/within destination data storage?



Several ways of implementation

- REST/gRPC APIs for uploading
  - APIs for calling/running batch of jobs
- File transfers and ingestion
  - move files into a stage space and run parallel tasks to process data files
- Containerized microservices moving files
  - detect changes/new files and move files
- Cron/workflow tasks
  - Workflows, Serverless/Function-as-a-service
- Complex design, including task management for multi tenants/users

Simple example: a Customer Data Platform



Say we want to store data into Google Storage in different spaces:

- hot, warm, cold and archive

We need uploaders and also movers (moving data between different spaces).
Mapped into specific technologies:

- Cloud Run: microservices/containers whose APIs can be triggered based on events
- Cloud Function: as serverless/function-as-a-service
- Cloud Composer (Airflow): a workflow engine
- bare Containers/VMs: write your own code, do your own way

Parallel/distributed processing

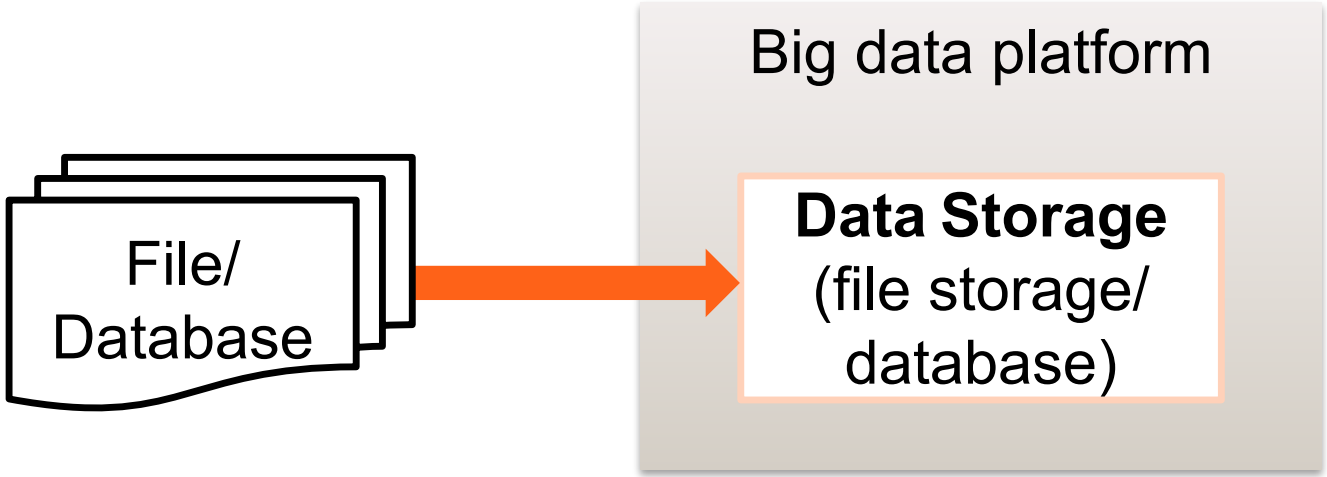
- Individual data file/dataset is big
  - parallel/distributed processing for single file
  - using suitable protocols and processing models, e.g., s5cmd (parallel S3) and workflows
- Multiple files/datasets but small individuals
  - parallel/distributed processing of tasks
  - task for a single file, e.g., concurrent, multi-thread processing and workflows
- Change data capture and failure handle could be tricky
- Complex bursty/ephemeral data processing
  - Leverage caching (e.g. using local disk or user space file systems)

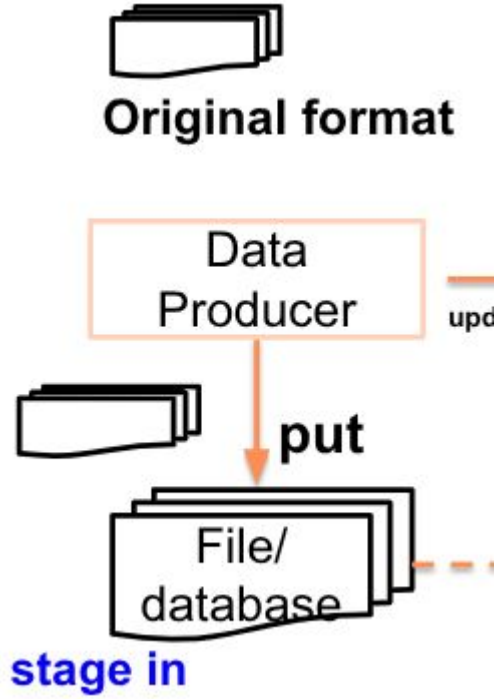
Remember we have a lot of data (files)

Data Staging

Why not direct ingestion?

When and when not?





Check the simple example in

https://github.com/rdsea/bigdataplatforms/tree/master/tutorials/queuebaseddataingestion

Integrate streaming data sources into platforms

(near real-time) streaming protocols/frameworks

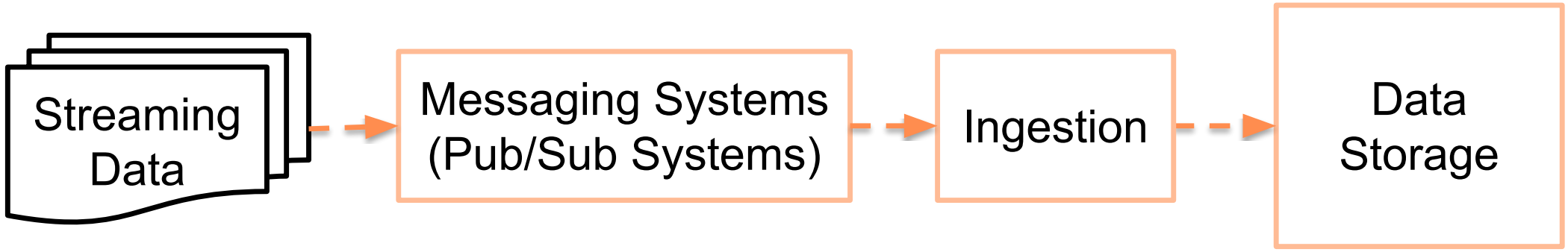
Big data platform

Data Storage (file storage/ database)

Example of scenarios

"A big data platform monitors network usage of devices from million+ customers. We have different levels: Sensor/Customer, Node (concentrator of multiple customers), Agent (concentrator of multiple Nodes) and the whole network. In a region, the real operator can generate 1.4 billion records per day ~ 72GB per day"

How do I move streaming data into the platform?



Protocol?
Data format
Message structure

Real-world technologies

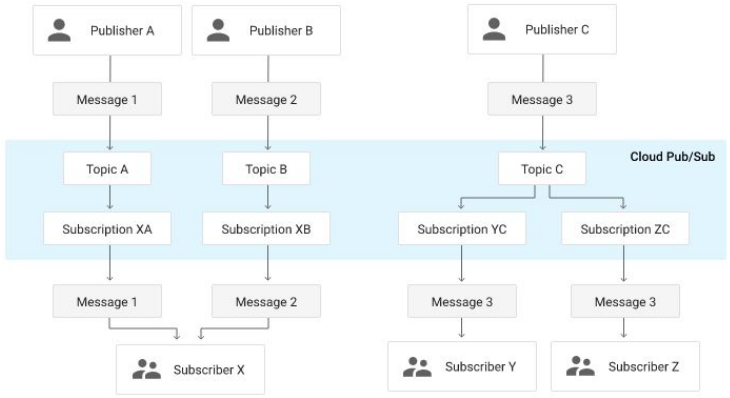


Figure source: *https://cloud.google.com/pubsub/docs/overview*

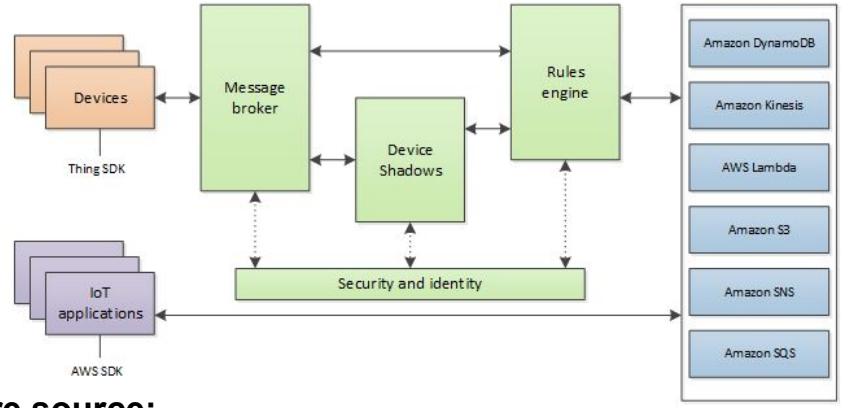


Figure source:
https://docs.aws.amazon.com/iot/latest/developerguide/aws-iot-how-it-works.html

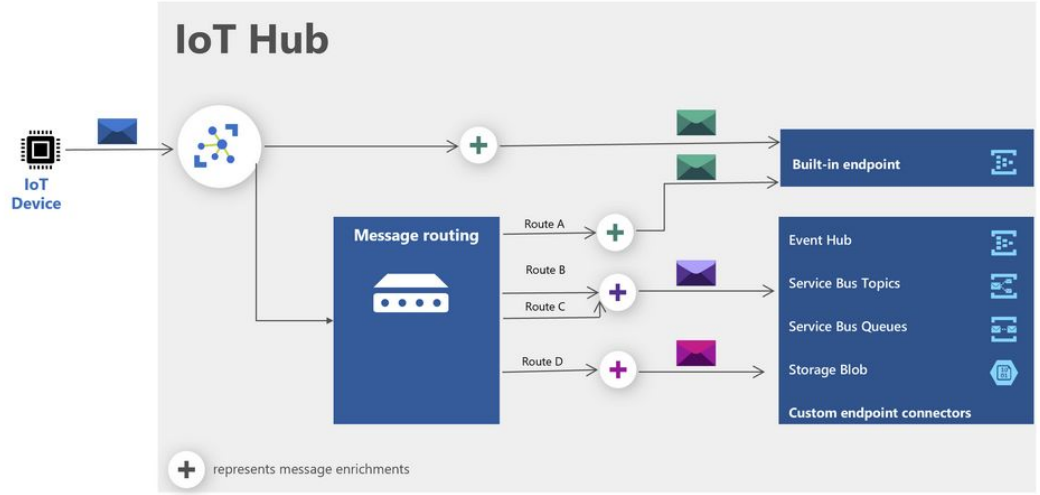


Figure source:
https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-message-enrichments-overview

Do you see common concepts/terms?

Some important protocols

- Protocols
  - AMQP, MQTT, NATS (https://nats.io/), etc.
- Systems
  - Apache Kafka, Apache Pulsar, Apache RocketMQ, RedPanda, Google PubSub
- Distinguish between “protocols” and “specific frameworks”
  - how would they affect your design?

Hybrid data processing architectures

Batch processing, micro-batching, and streaming

- Batch processing
  - data to be processed: complete with a large size
  - high throughput but also high latency
  - triggered by scheduled/manual time or by data completion (size or time)
- Micro-batching
  - data to be processed: small; collected over a short, regular time; small batches
  - near real-time and low latency
- Streaming
  - data to be processed: as soon when it arrives; record-by-record
  - very low latency and near real-time

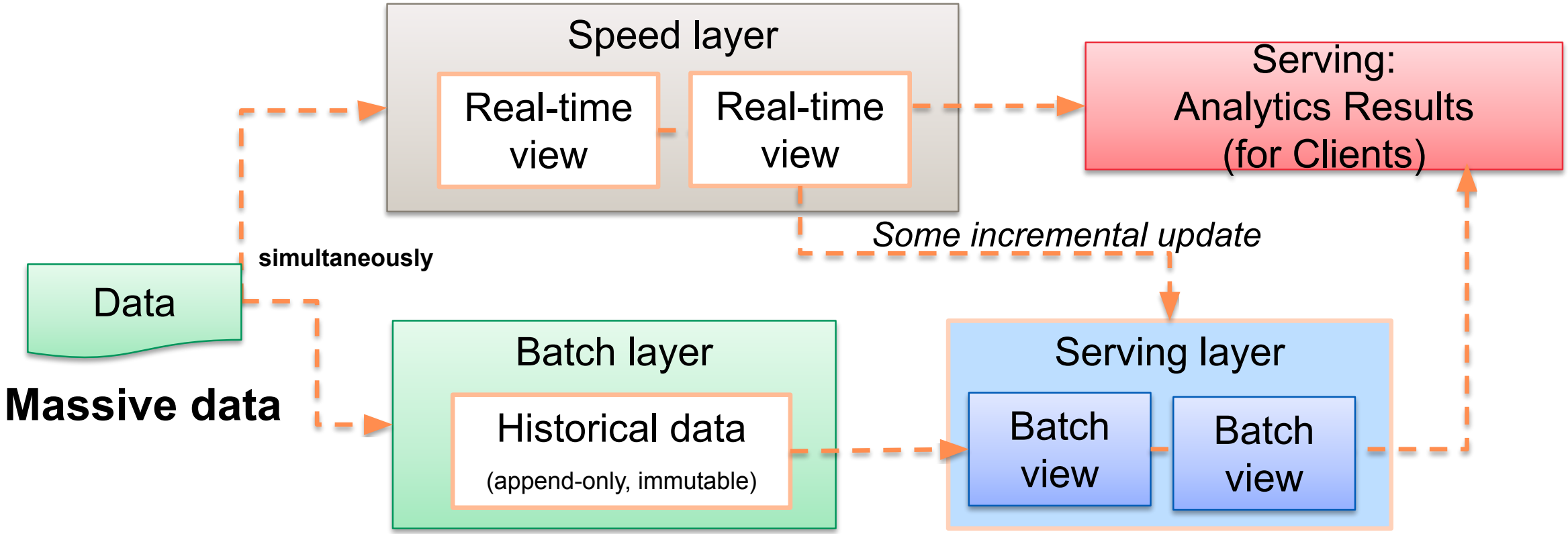
Recall:

"A big data platform monitors network usage of devices from million+ customers. We have different levels: Sensor/Customer, Node (concentrator of multiple customers), Agent (concentrator of multiple Nodes) and the whole network. In a region, the real operator can generate 1.4 billion records per day ~ 72GB per day"

How to allow both live analytics of the situation in the last 5 minutes vs of the historical situation in the last 6 month?

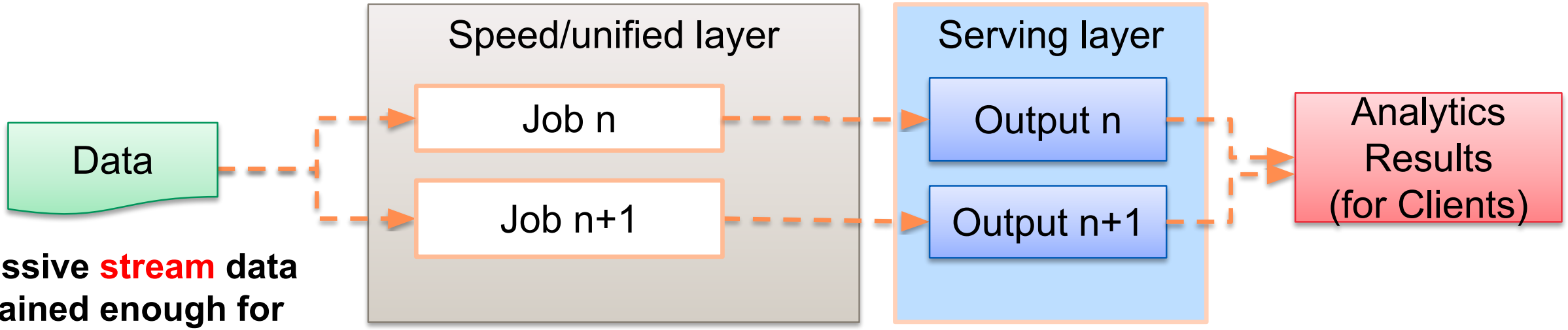
Can we write more or less a single pipeline/code for both cases?

Lambda architectural style



Check: https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/
https://www.oreilly.com/radar/questioning-the-lambda-architecture/

Kappa architectural style



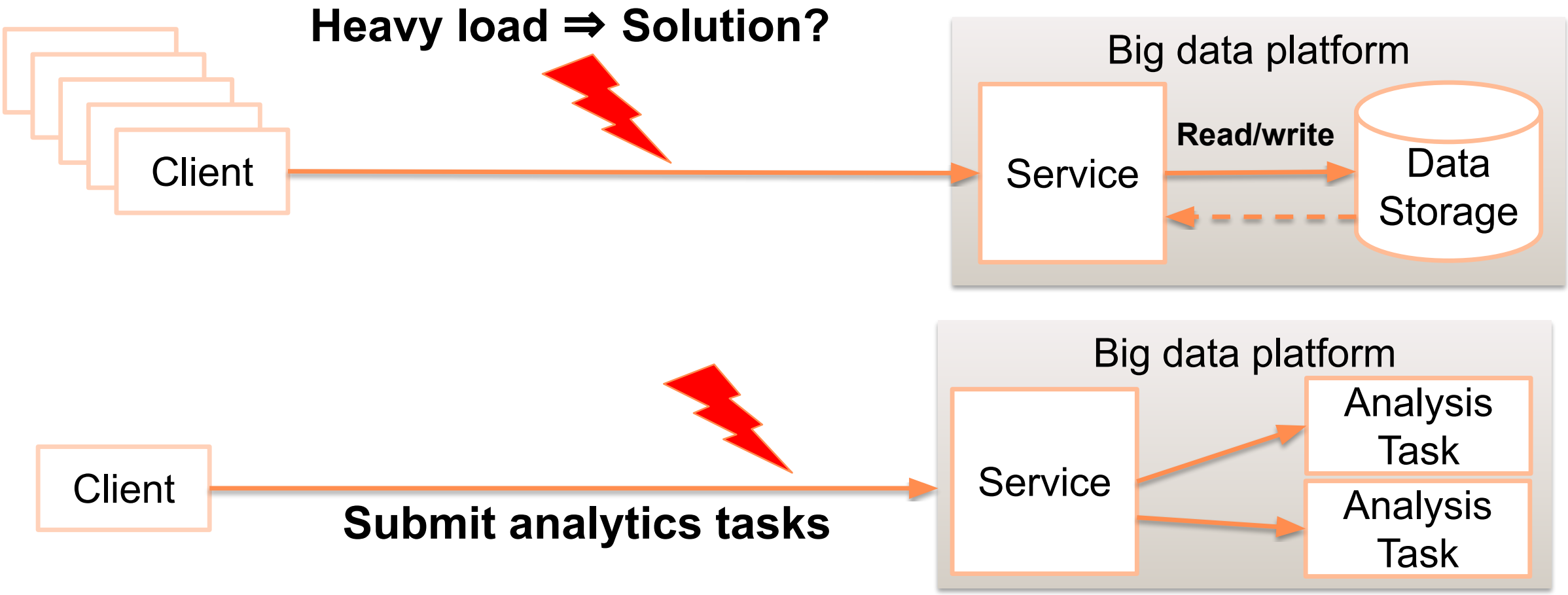
Massive stream data retained enough for different analytics needed (e.g., Lakehouse)

Job n: analytics/processing
Can switch from version n to version n+1 using a single software stack

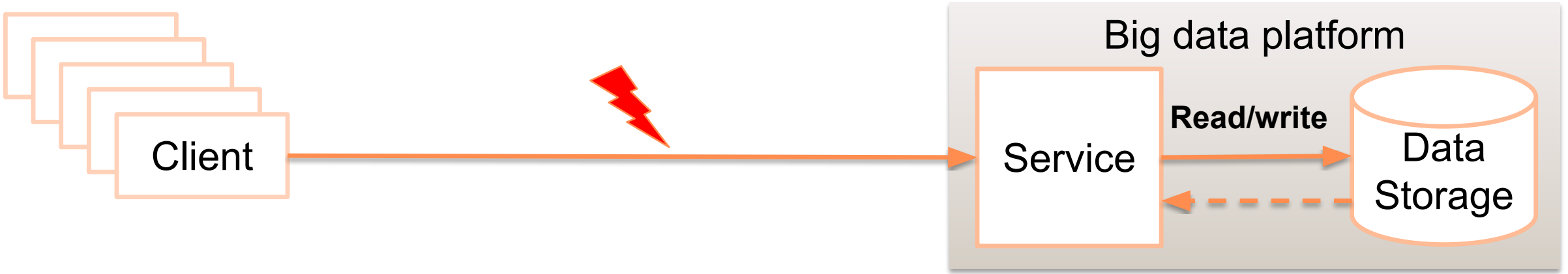
Check: https://milinda.pathirage.org/kappa-architecture.com/ & http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html

Optimize data service requests and functionalities
→ handling “unlimited vs limited amount of data”

Concurrent contention



Back-pressure or elasticity



Back-pressure: control, drop, and buffer

Prevent too many accesses?

A related situation: prevents clients to retry a (failed) operation
http://martinfowler.com/bliki/CircuitBreaker.html
https://msdn.microsoft.com/en-us/library/dn589784.aspx

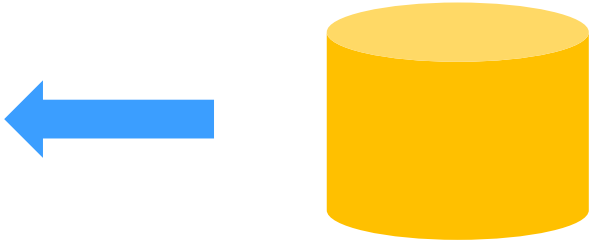
Throttling principle and SLA/contract

- Drop strategy: disable too many accesses and disable unessential services
  - dynamic vs static configuration
- E.g., using API Gateway Kong, Kubernetes


```
REST_FRAMEWORK = {
    'DEFAULT_THROTTLE_CLASSES': [
        'rest_framework.throttling.AnonRateThrottle',
        'rest_framework.throttling.UserRateThrottle'
    ],
    'DEFAULT_THROTTLE_RATES': {
        'anon': '100/day',
        'user': '1000/day'
    }
}
```

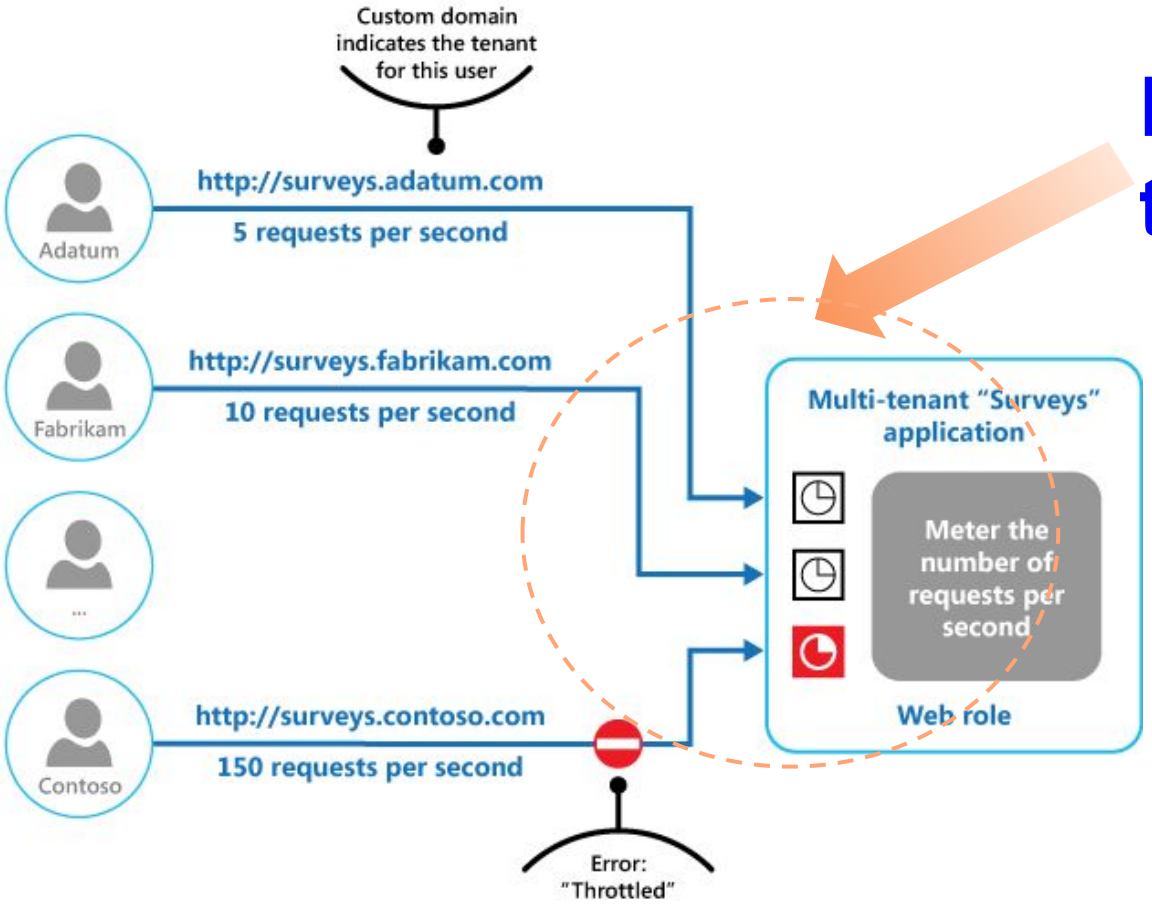
Code:
http://www.django-rest-framework.org/api-guide/throttling/#how-throttling-is-determined

Tenant profile/SLA/ Contract



Who will define them?

Example of throttling based on roles



How would you do this in the big data platform?

Big Data Platforms implement "business service models"/SLA for tenants

Figure source:
https://msdn.microsoft.com/en-us/library/dn589798.aspx

Using multiple instances of services and queues

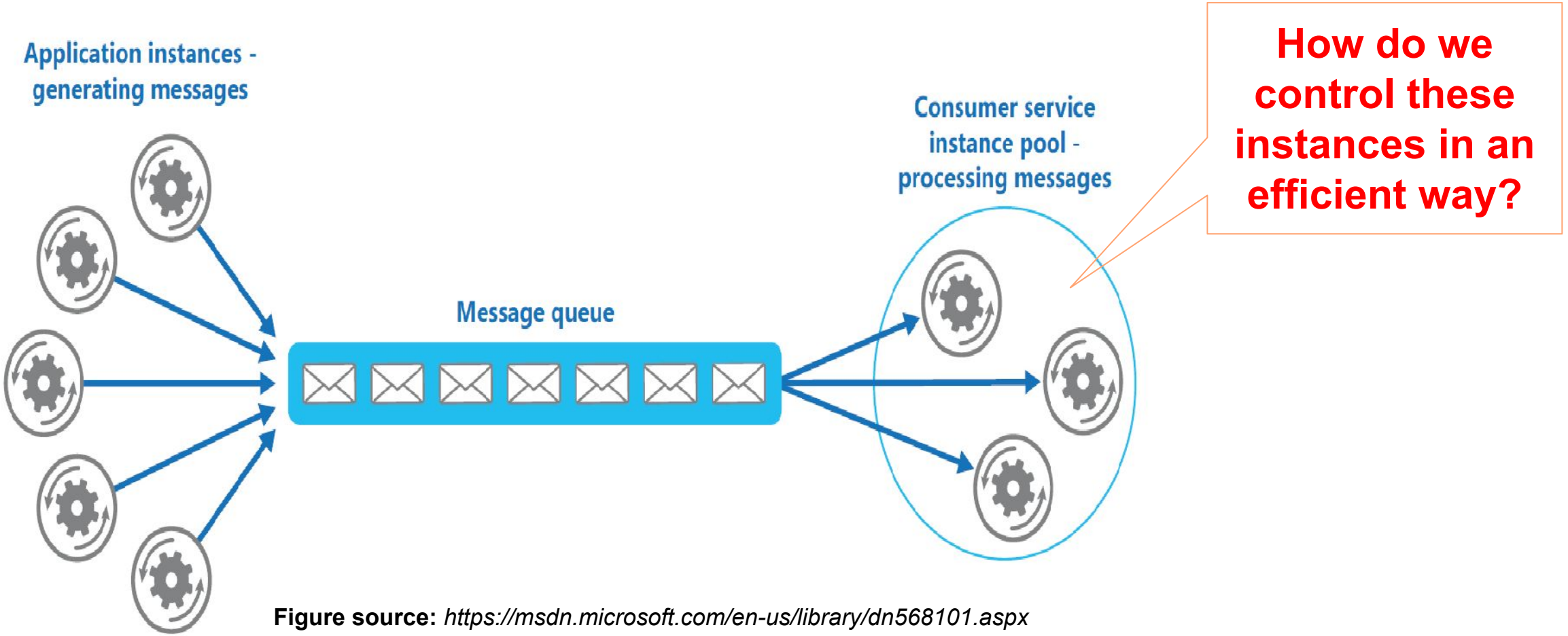
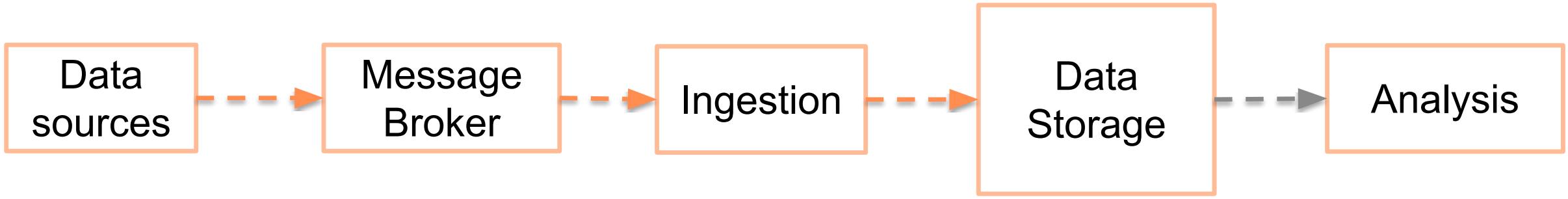


Figure source: *https://msdn.microsoft.com/en-us/library/dn568101.aspx*

Scaling in every place of big data pipelines

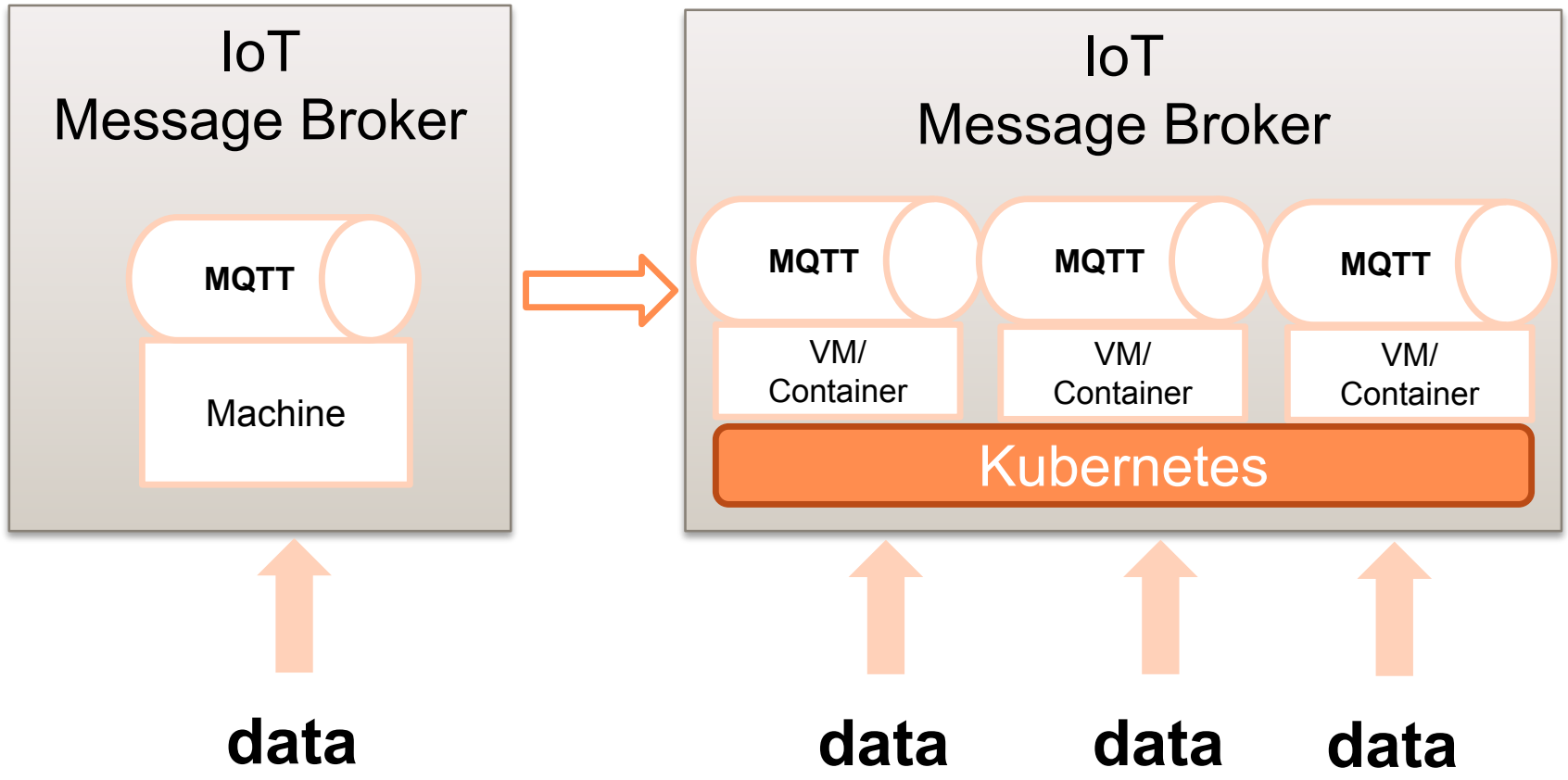


- Scaling
  - disk spaces for file storage
  - resources for data ingestion
  - resources for data analysis

Happen at different times and locations

Scaling middleware nodes

- Increase the number of brokers when more data arrive
- Provide dedicated brokers on-demand



Example: scaling compute nodes for data analysis

Monitoring Jobs VM Instances Configuration Web Interfaces

Name	Role	
thebasecluster-m	Master	SSH
thebasecluster-w-0	Worker	
thebasecluster-w-1	Worker	
thebasecluster-w-2	Worker	
thebasecluster-w-3	Worker	

Equivalent REST

4 nodes

On-demand change

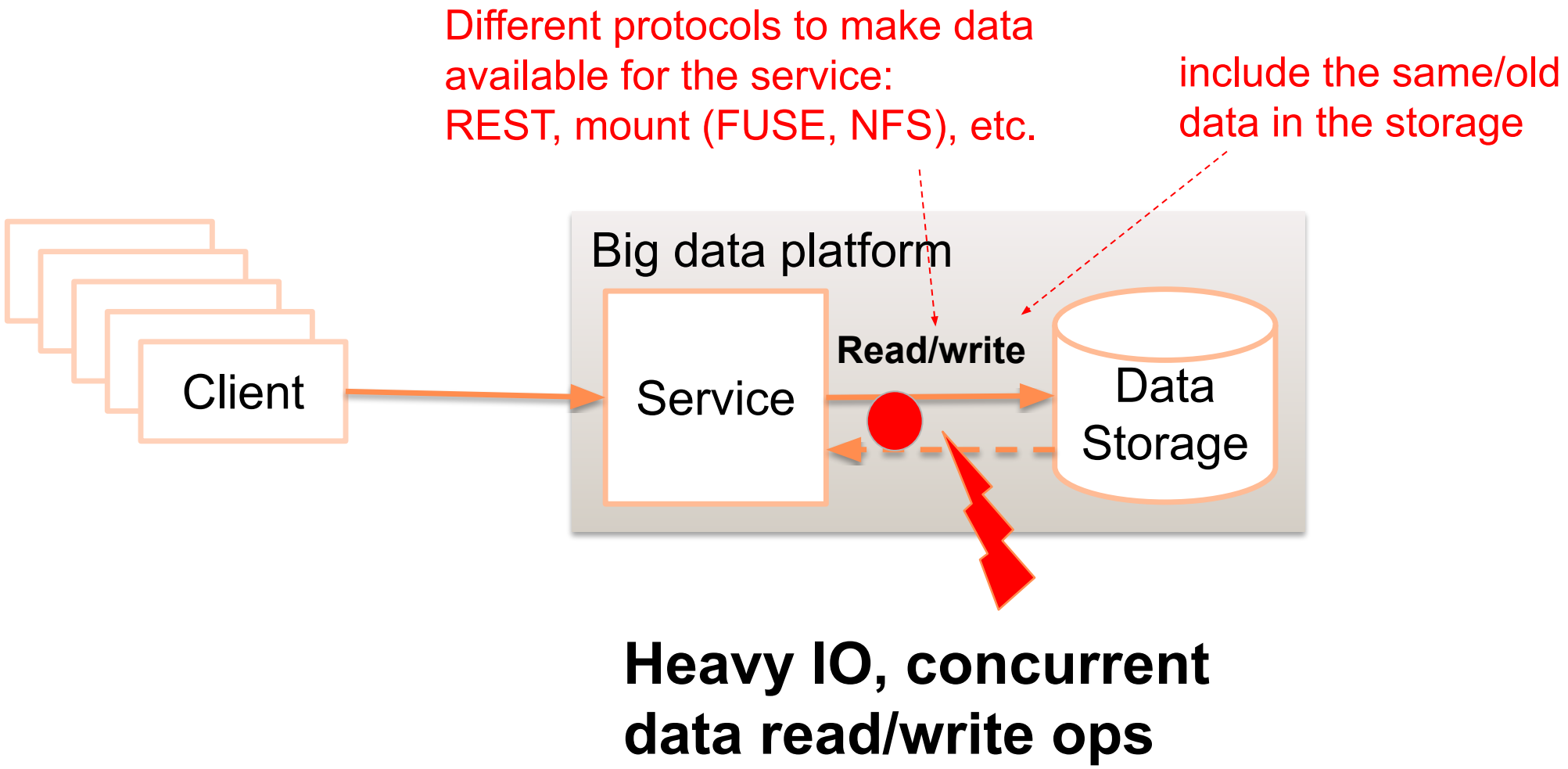
Name	thebasecluster
Region	europe-north1
Zone	europe-north1-a
Autoscaling	Off
Scheduled deletion	Off
Enhanced flexibility mode	Off
Master node	Standard (1 master, N workers)
Machine type	n1-standard-2 (2 vCPU, 7.50 GB memory)
Primary disk type	pd-standard
Primary disk size	500 GB
Worker nodes	6
Machine type	n1-standard-1 (1 vCPU, 3.75 GB memory)

Monitoring Jobs VM Instances Configuration Web Interfaces

Name	Role	
thebasecluster-m	Master	SSH
thebasecluster-w-0	Worker	
thebasecluster-w-1	Worker	
thebasecluster-w-2	Worker	
thebasecluster-w-3	Worker	
thebasecluster-w-4	Worker	
thebasecluster-w-5	Worker	

6 nodes

Caching and concurrency



Discovery and consensus

Where are available services and data?

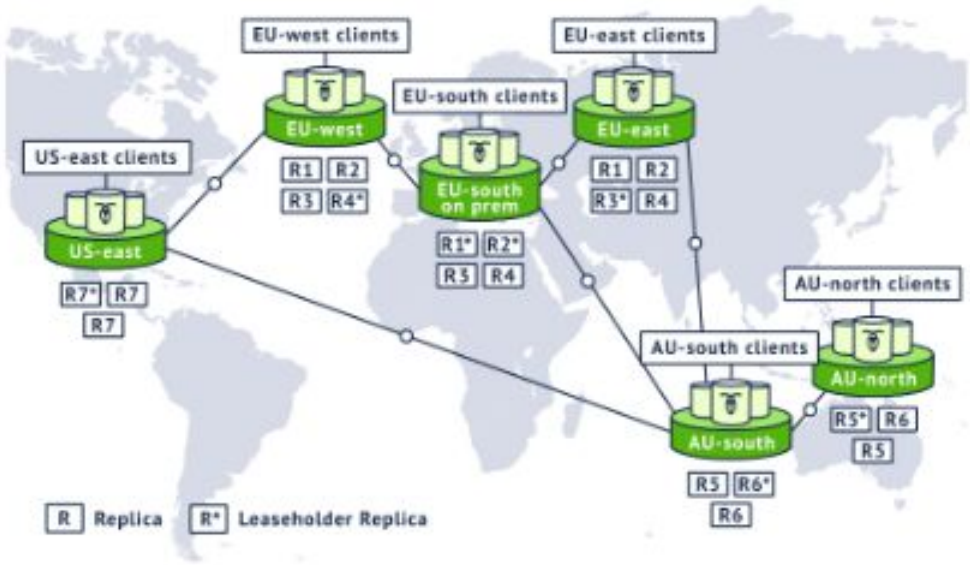


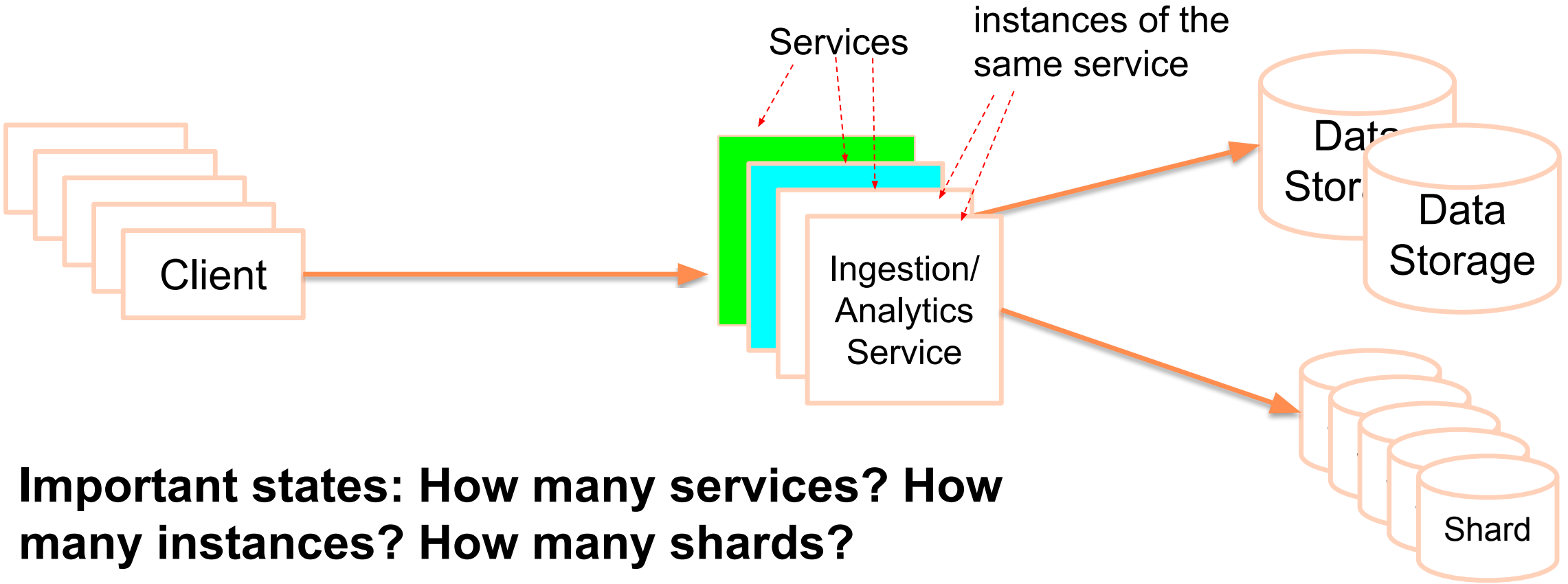
Figure 1: A global CockroachDB cluster

Figure source: Taft et al., CockroachDB: The Resilient Geo-Distributed SQL Database, https://dl.acm.org/doi/pdf/10.1145/3318464.3386134

"At the time of writing, our largest Druid cluster deployment uses more than 100 nodes for Historical processes and about 75 nodes for MiddleManager processes. We ingest over three million events per second and respond to over 250 queries per second. We keep seven days of queryable data in Druid Historical nodes and two years of data retention in S3 deep storage."

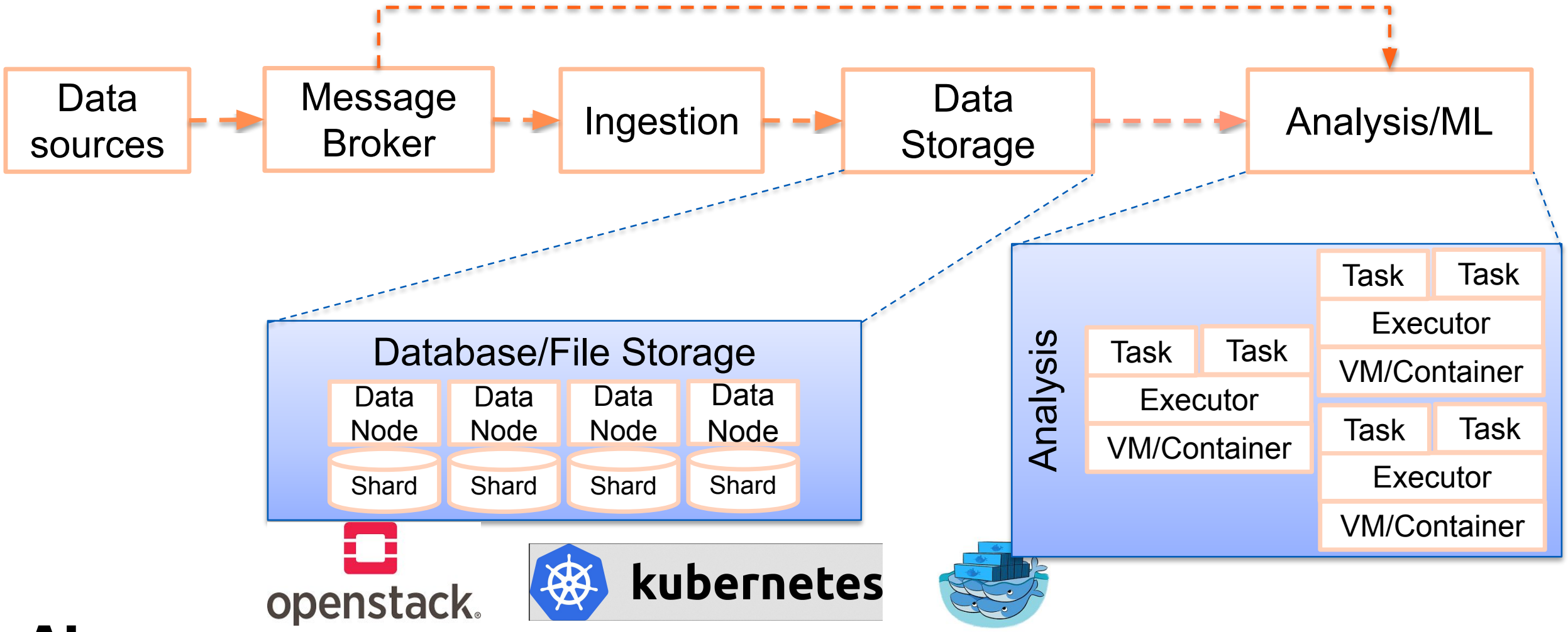
Source: *November 8, 2021, https://www.confluent.io/blog/scaling-apache-druid-for-real-time-cloud-analytics-at-confluent/*

We can create a lot of instances or we can create new services



Important states: How many services? How many instances? How many shards?

Runtime view of some components



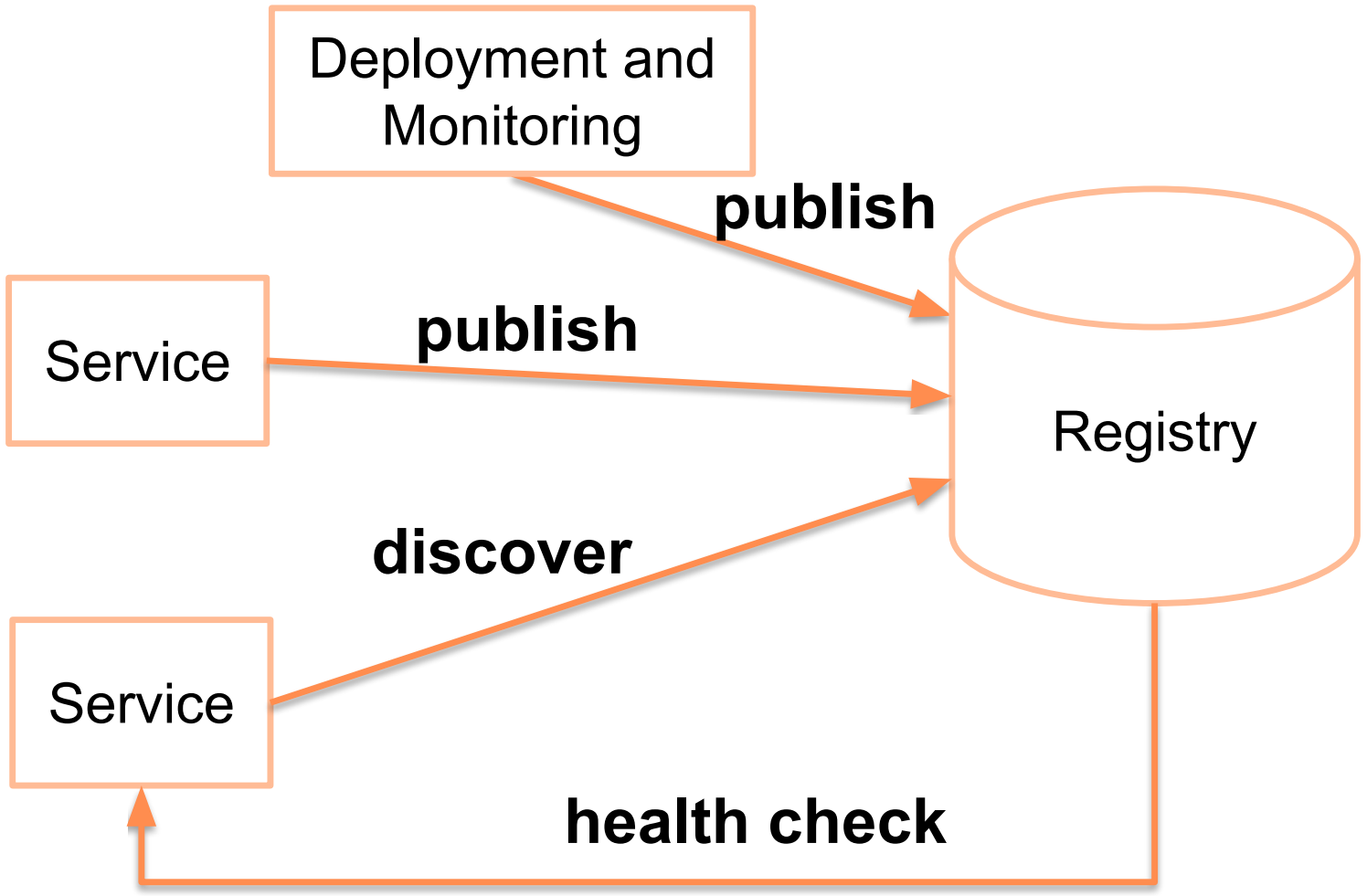
Multiple instances

- A building block of big data platforms can have many services and a service can have many instances
  - e.g., for replication and load balancing
  - a database service (e.g. MongoDB) has multiple data nodes, each responsible for a subset of shards/partitions
  - a processing engine (e.g., Spark or Airflow) can have many nodes, each executes different tasks of a process
- The same component can have many instances deployed
  - e.g., dedicated deployment of MongoDB for different customers

Service state management

- Service information
  - include states and other important configuration information
  - many instances
  - cross different infrastructures/data centers
- Related components
  - services themselves
  - monitoring component, deployment component, orchestration controllers
- Lifecycle: very dynamic in elastic environments
  - start, run, shutdown, restart, scale

Service Discovery principle

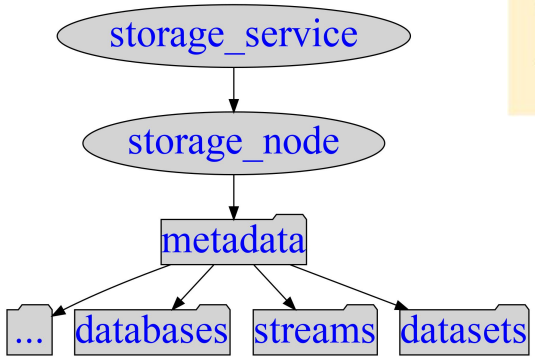


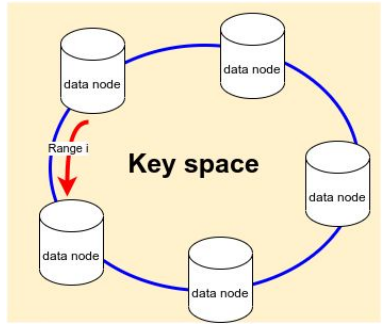
- Key requirements
  - fast
  - reliable
  - consistent
  - secure
  - cross data centers
  - simple APIs

But what about data discovery?

- Approach 1
  - pre-defined mapping: e.g., using consistent hashing or global name space
- Approach 2
  - discover relevant services and then ask relevant services about metadata about data
- Approach 3
  - use a dedicated data discovery service and ask the service for metadata about data

Metadata is also big data!





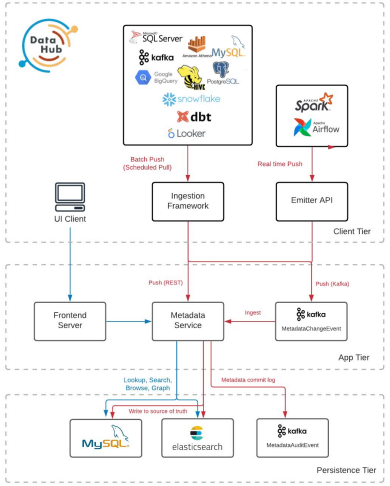


Figure source:
https://datahubproject.io/docs/architecture/architecture

Example:

https://github.com/rdsea/bigdataplatforms/tree/master/tutorials/servicediscovery

Consensus for big data platforms

- Consensus is about to agree on something
- Very important for replication and fault tolerance in big data platforms
  - distributed lock, master selection
- Scope
  - platform level and service component levels
  - single data center or cross-data center
- We will have to deal with them in several frameworks for big data

Distributed coordination

- A lot of algorithms, etc.
  - Paxos family
- Well-known in the cloud and distributed systems

Notes from the paper: “server replication (SR), log replication (LR), synchronization service (SS), barrier orchestration (BO), service discovery (SD), group membership (GM), leader election (LE), metadata management (MM) and distributed queues (Q)”

TABLE 4. PATTERNS OF PAXOS USE IN PROJECTS

		Usage Patterns								
Project	**Consensus System**	**SR**	**LR**	**SS**	**BO**	**SD**	**GM**	**LE**	**MM**	**Q**
GFS	Chubby			✓				✓	✓	
Borg	Chubby/Paxos	✓				✓		✓		
Kubernetes	etcd						✓		✓	
Megastore	Paxos		✓							
Spanner	Paxos	✓								
Bigtable	Chubby						✓	✓	✓	
Hadoop/HDFS	ZooKeeper	✓						✓		
HBase	ZooKeeper	✓		✓			✓		✓	
Hive	ZooKeeper			✓					✓	
Configerator	Zeus								✓	
Cassandra	ZooKeeper					✓		✓	✓	
Accumulo	ZooKeeper		✓	✓					✓	
BookKeeper	ZooKeeper						✓		✓	
Hedwig	ZooKeeper						✓		✓	
Kafka	ZooKeeper						✓	✓	✓	
Solr	ZooKeeper							✓	✓	✓
Giraph	ZooKeeper		✓		✓				✓	
Hama	ZooKeeper				✓					
Mesos	ZooKeeper							✓		
CoreOS	etcd					✓				
OpenStack	ZooKeeper					✓				
Neo4j	ZooKeeper			✓				✓		

Source: *Ailidani Ailijiang, Aleksey Charapkoy and Murat Demirbasz, Consensus in the Cloud: Paxos Systems Demystified, http://www.cse.buffalo.edu/tech-reports/2016-02.pdf*

What you should do this week

- Look at the list of data sources and start think which data sources you will use for your study
- Lambda and Kappa architecture styles
- Check and play with basic ingestion: simple queue, MQTT/AMQP (from the cloud background)
- Brush up patterns for scaling and failure handling
- Look at how service discovery and consensus are implemented in big data systems

Note: materials/links are in our git and slides

Thanks!

Hong-Linh Truong
Department of Computer Science

rdsea.github.io

—

Kiitos
aalto.fi