



Aalto University  
School of Science

# Service and Integration Models in Big Data Platforms

*Hong-Linh Truong*

*Department of Computer Science*

*[linh.truong@aalto.fi](mailto:linh.truong@aalto.fi)*, *<https://rdsea.github.io>*

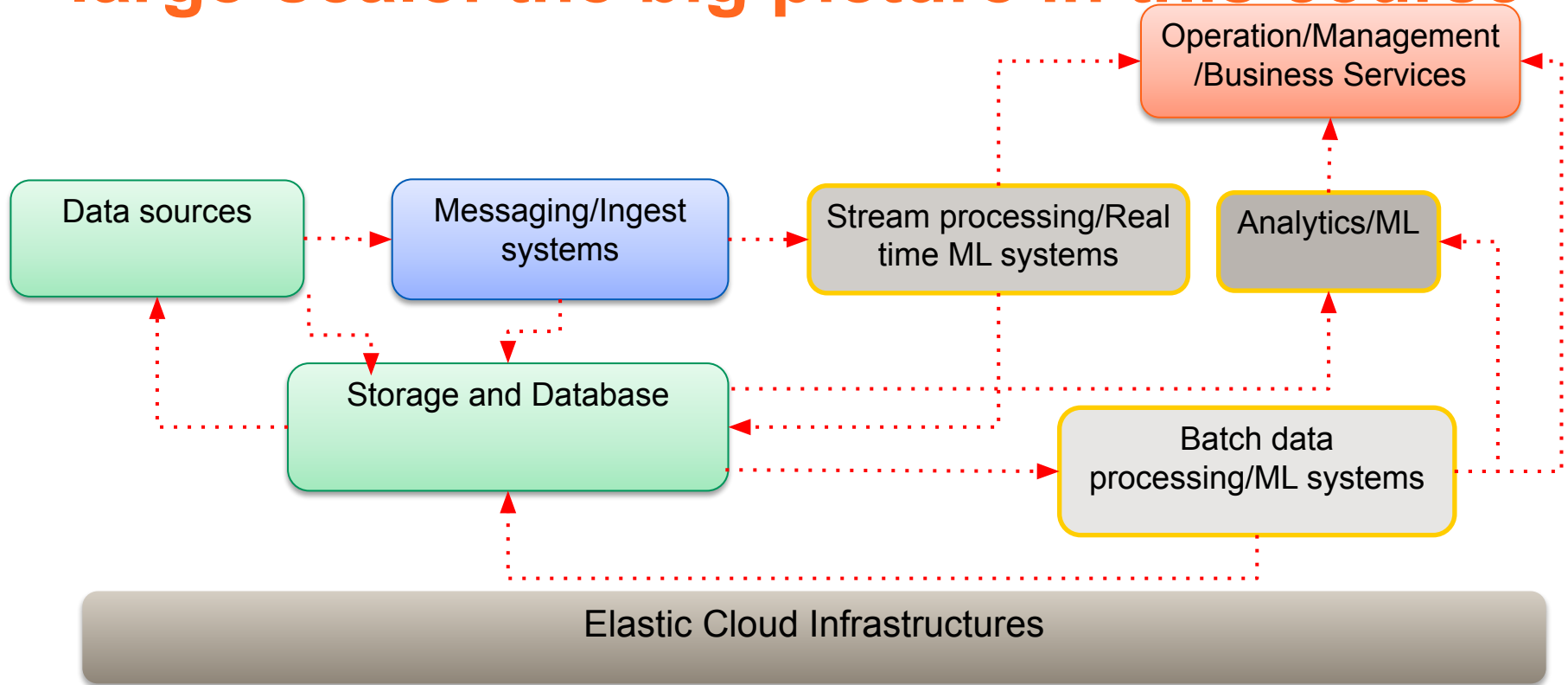
# Learning objectives

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

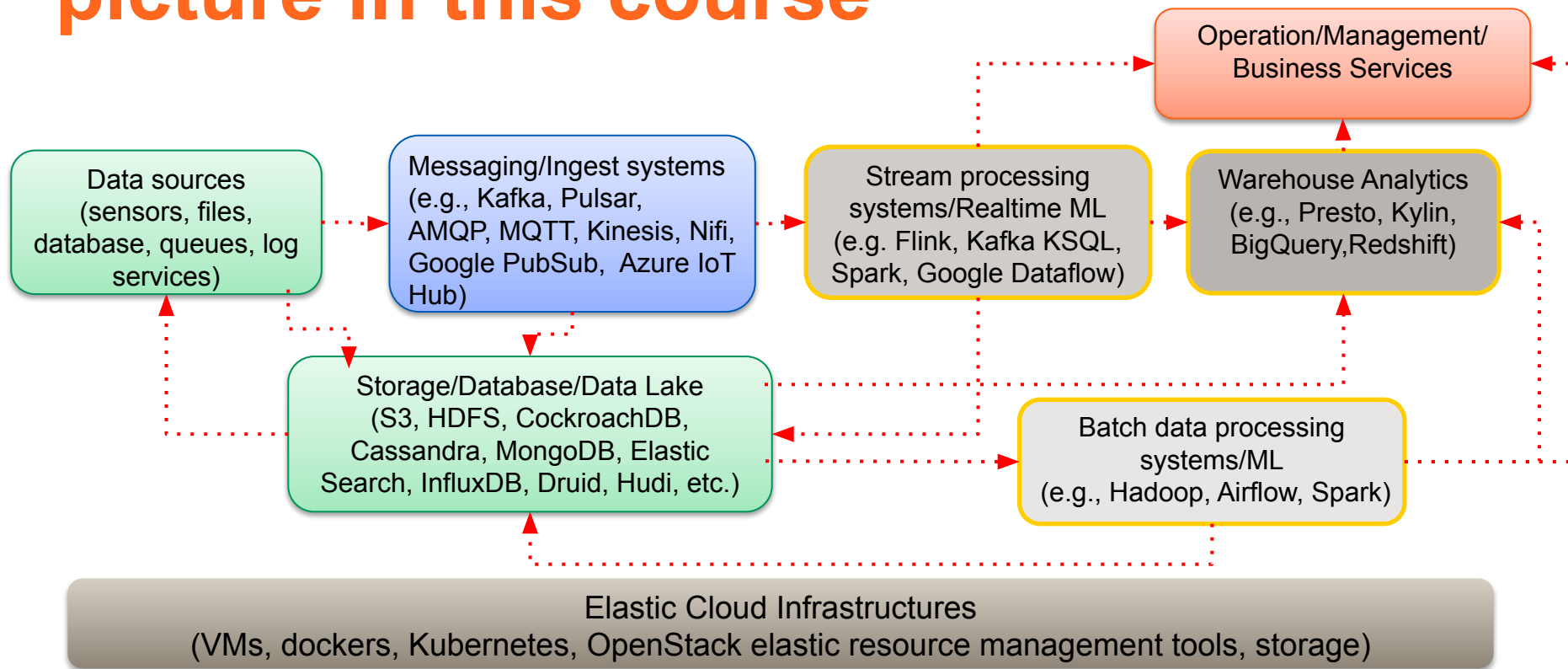
# Recall

- **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

# Basic building blocks for big data at large-scale: the big picture in this course



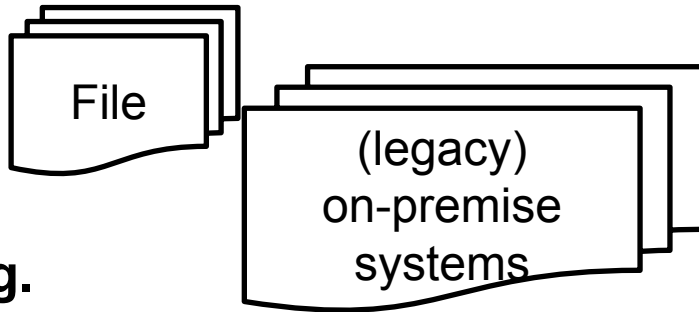
# Big data at large-scale: the big picture in this course



# Basics of moving big data to cloud

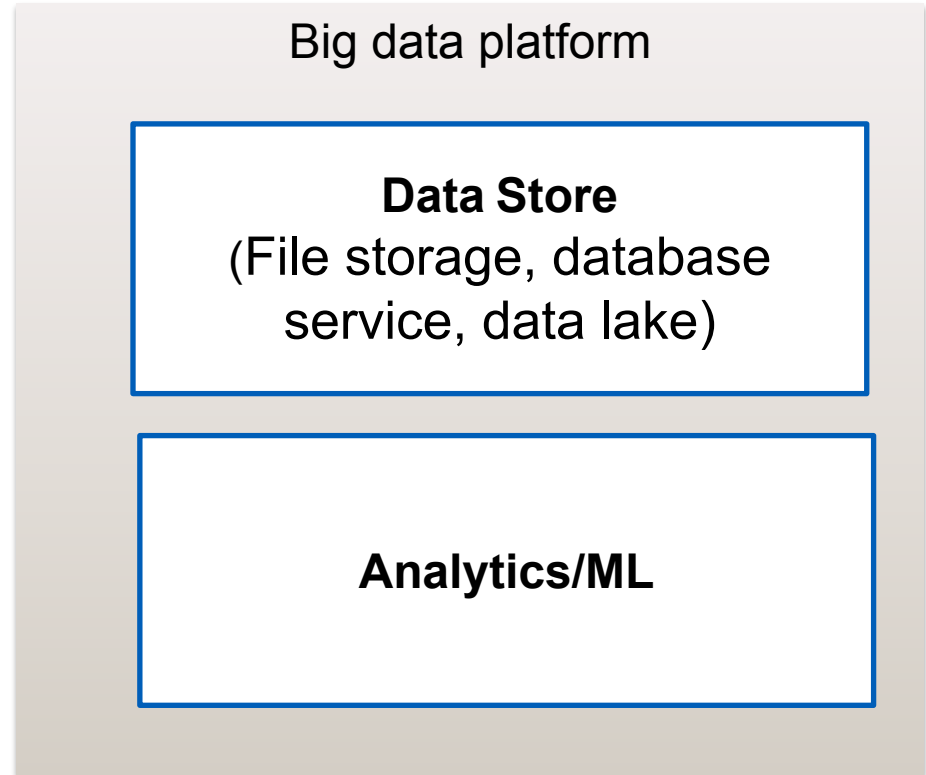
# Bring big data in files/datasets into platforms

## Tenants/Users side

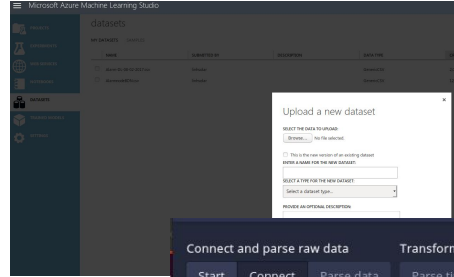
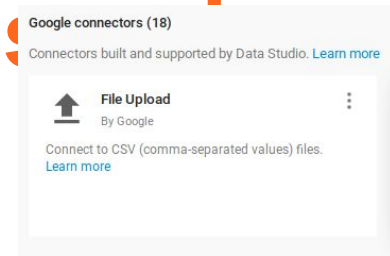


e.g.

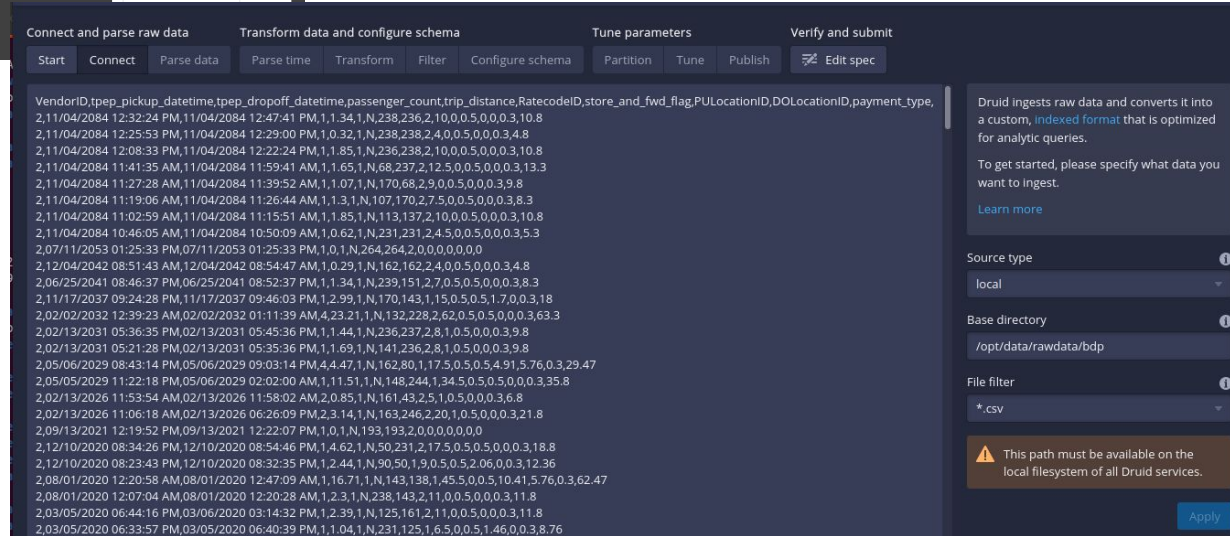
- logs of machines
- sell receipt
- transaction records
- Images/video



# First obstacle: uploading big data into cloud data storage/database



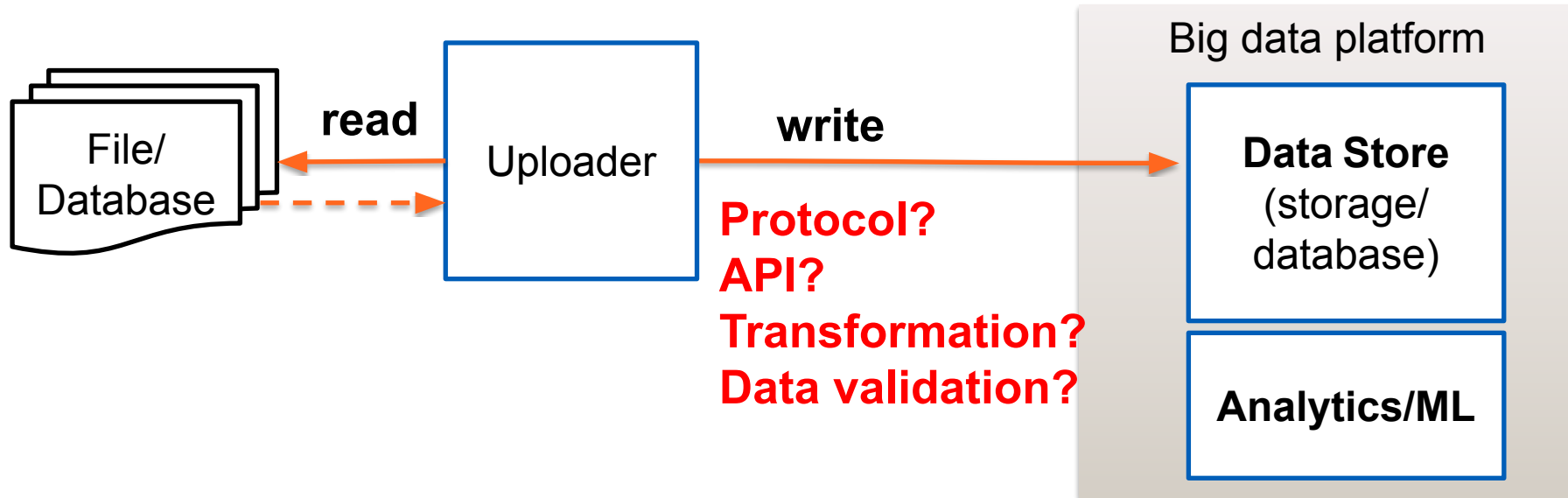
What would be a good way?  
Using Flask REST API?



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

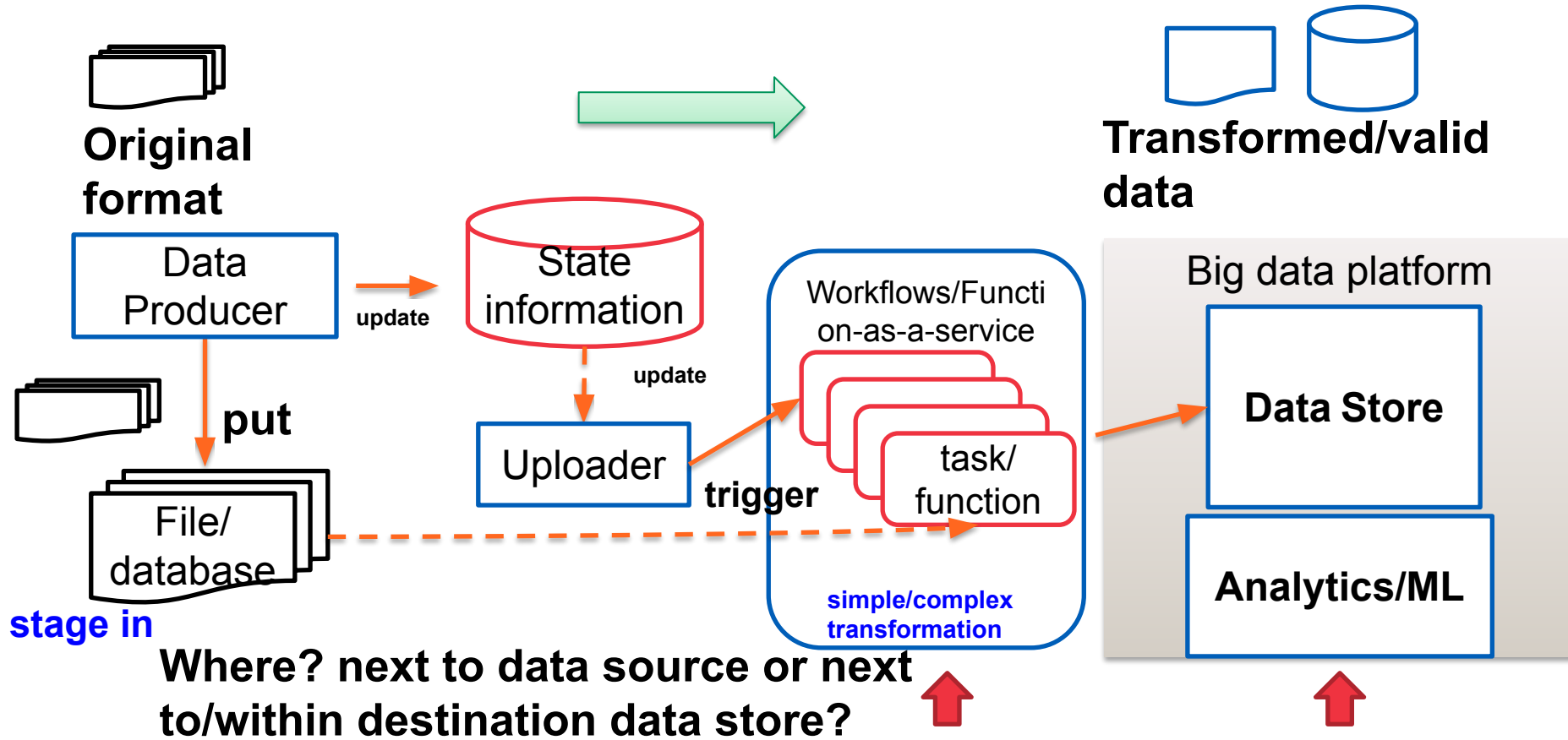


# Complex design details



- **Practical issues for optimization:**
  - What if you have very big files? Or a lot of small files?
  - Any ideas about possible techniques?

# The transformation/validation is needed?



# Several ways of implementation

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

# Parallel/distributed processing

- **Individual data file/set is big**
  - parallel processing for single file
  - using suitable models
    - *MapReduce, Workflow, etc.*
- **Multiple files/datasets but small**
  - parallel/distributed processing of tasks
  - task for a single file
    - *multi-thread, Dask, etc.*
- **Change data capture and failure handle could be tricky**

Remember we  
have a lot of data  
(files)

# Check the simple example in

<https://version.aalto.fi/gitlab/bigdataplatfroms/cs-e4640/-/tree/master/tutorials/queuebaseddataingestion>

# Integrate streaming data sources into platforms



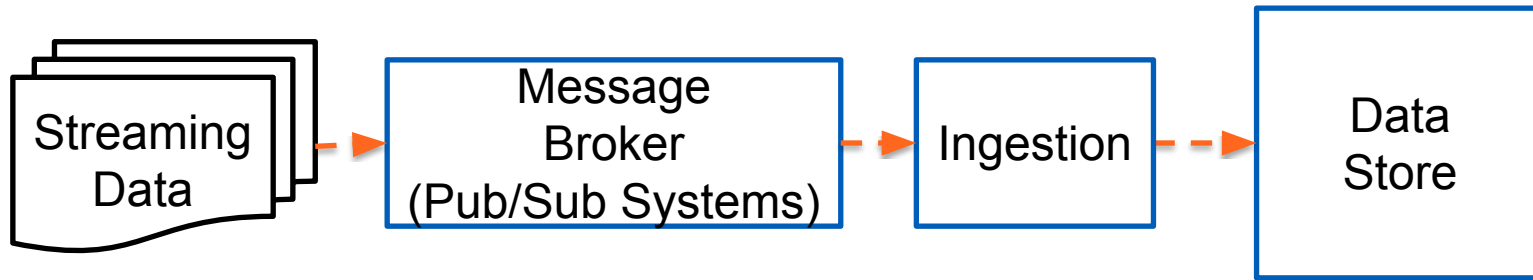
**Streaming  
protocols/frameworks**



# Quick check

**“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 cloud?



**Protocol?**  
**Data format**  
**Message structure**



# Real-world technologies

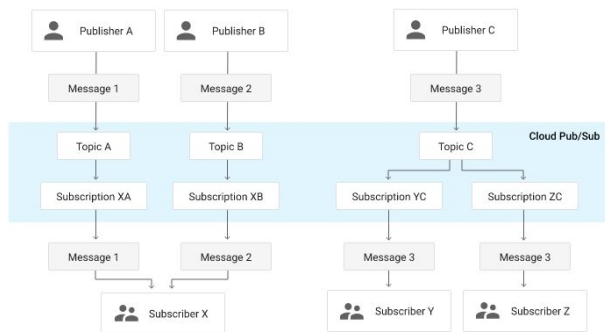


Figure source: <https://cloud.google.com/pubsub/docs/overview>

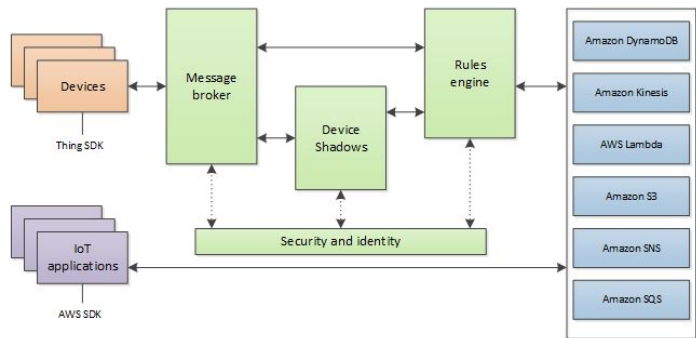


Figure source: <https://docs.aws.amazon.com/iot/latest/developerguide/aws-iot-how-it-works.html>

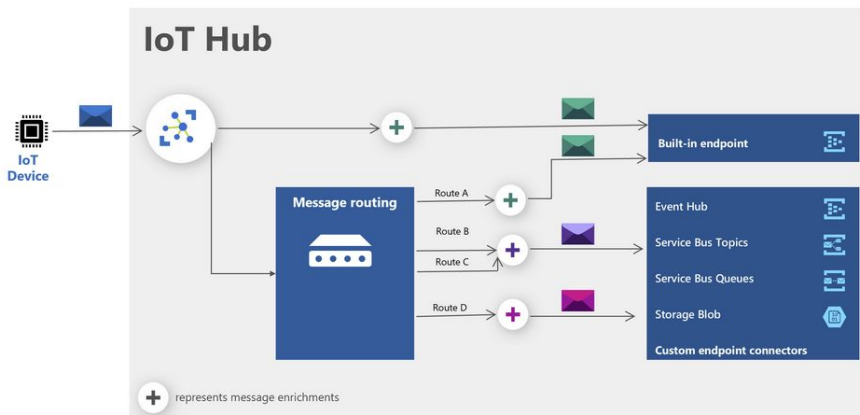


Figure source: <https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-message-enrichments-overview>

**Do you see common concepts/terms?**

# Some important protocols

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

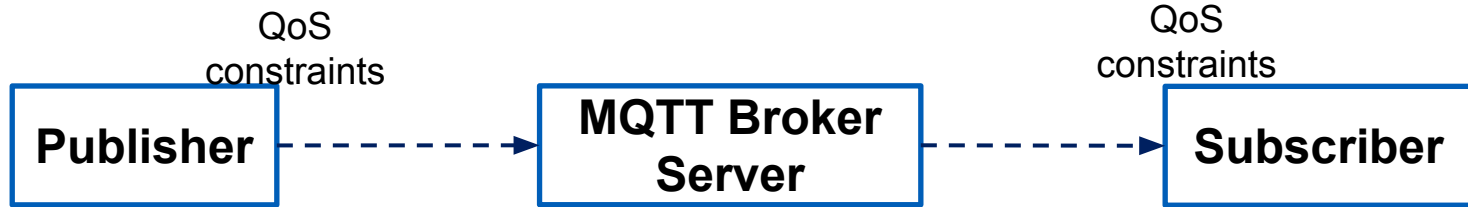
# AMQP - Overview

- **Protocol for message-oriented middleware**
  - Not language- or platform- specific
  - For Java, C#, Python, ....
- **Binary wire-level protocol for message exchange, rather than APIs**
- **<http://www.amqp.org>**
- **We use it for *big data movement and tasks coordination***

# MQTT Overview

- <http://mqtt.org>
- **OASIS Standard**
  - ISO/IEC 20922:2016 (Message Queuing Telemetry Transport (MQTT) v3.1.1)
- **IoT/M2M connectivity protocol atop TCP/IP**
- **MQTT brokers enable publish/subscribe messaging systems**
  - Publisher can publish a message within a topic that can be subscribed by many Subscribers
- **We use it mostly for *big data movement***

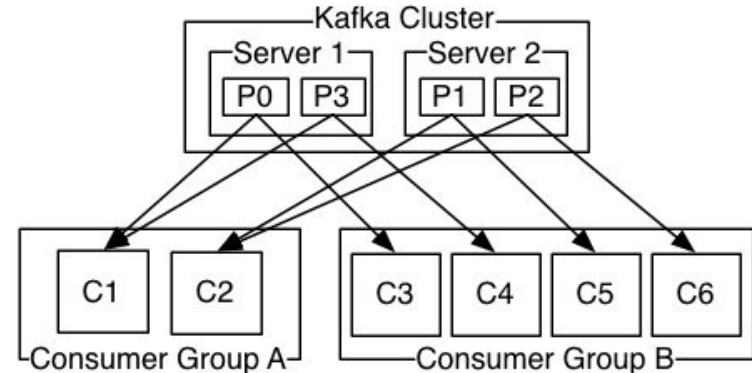
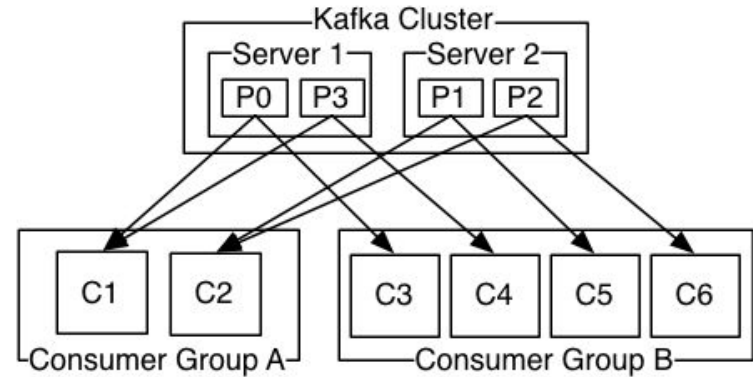
# Model and Implementation



- Different programming languages for OS/devices
- Implementation examples
  - Mosquitto (<http://projects.eclipse.org/projects/technology.mosquitto>)
    - docker pull eclipse-mosquitto
  - Paho: <http://www.eclipse.org/paho/>
  - RabbitMQ: <https://www.rabbitmq.com/>
  - Cloud providers: <http://cloudmqtt.com> (offer a free instance)
  - Cluster of MQTT brokers: VerneMQ (<https://vernemq.com/>), EMQ (<https://www.emqx.io/>)

# Apache Kafka

- <http://kafka.apache.org/>
  - originally from LinkedIn, not a protocol!
- **Some components are commercialized by Confluent**
  - <https://www.confluent.io/>
- **Widely used for big data use cases,**
  - including message processing in large-scale enterprise service platforms



Figures source: <http://kafka.apache.org/documentation.html#majordesignelements>

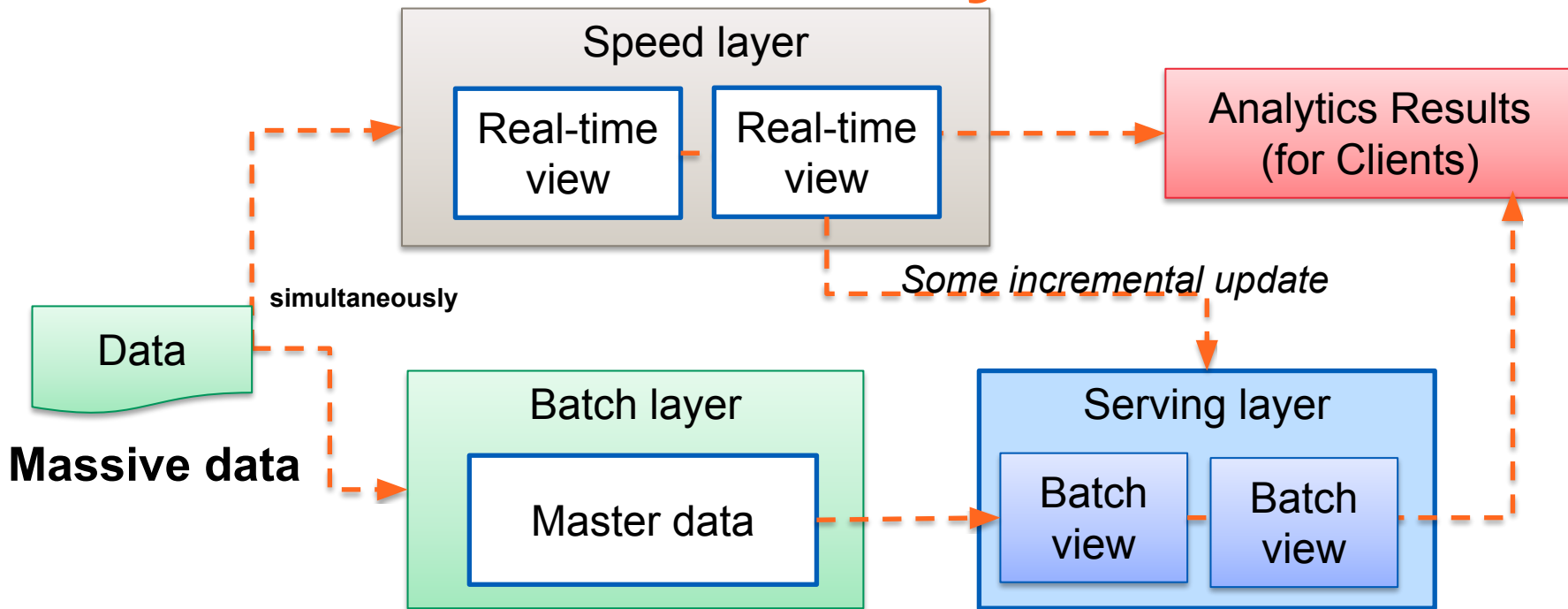
# So which one you think is suitable for this?

**“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”**

# Combining batch and streaming

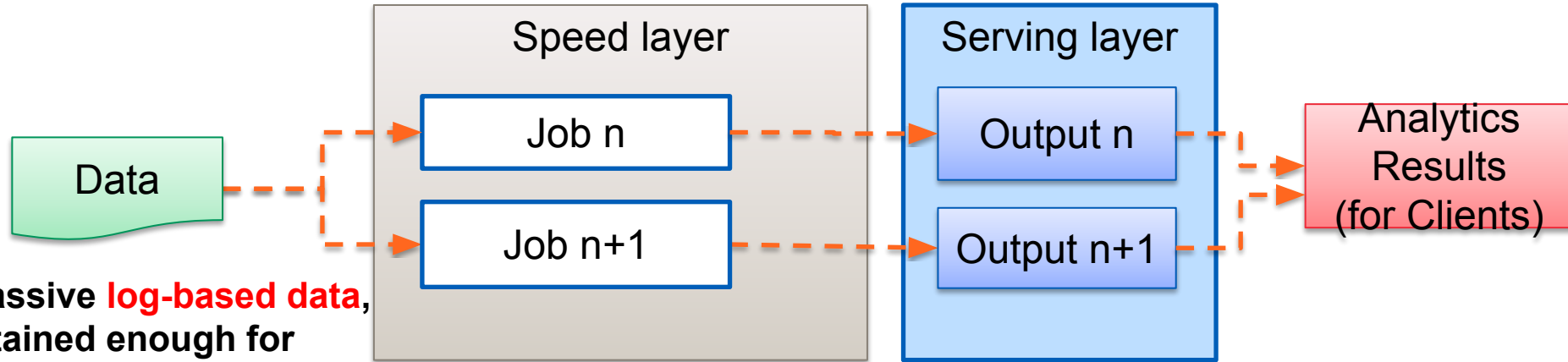


# Lambda architectural style



**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 **log-based data**,  
retained enough for  
different analytics  
needed

**Job n: analytics/processing**

**Can switch from version n to version n+1 using a  
single stack**

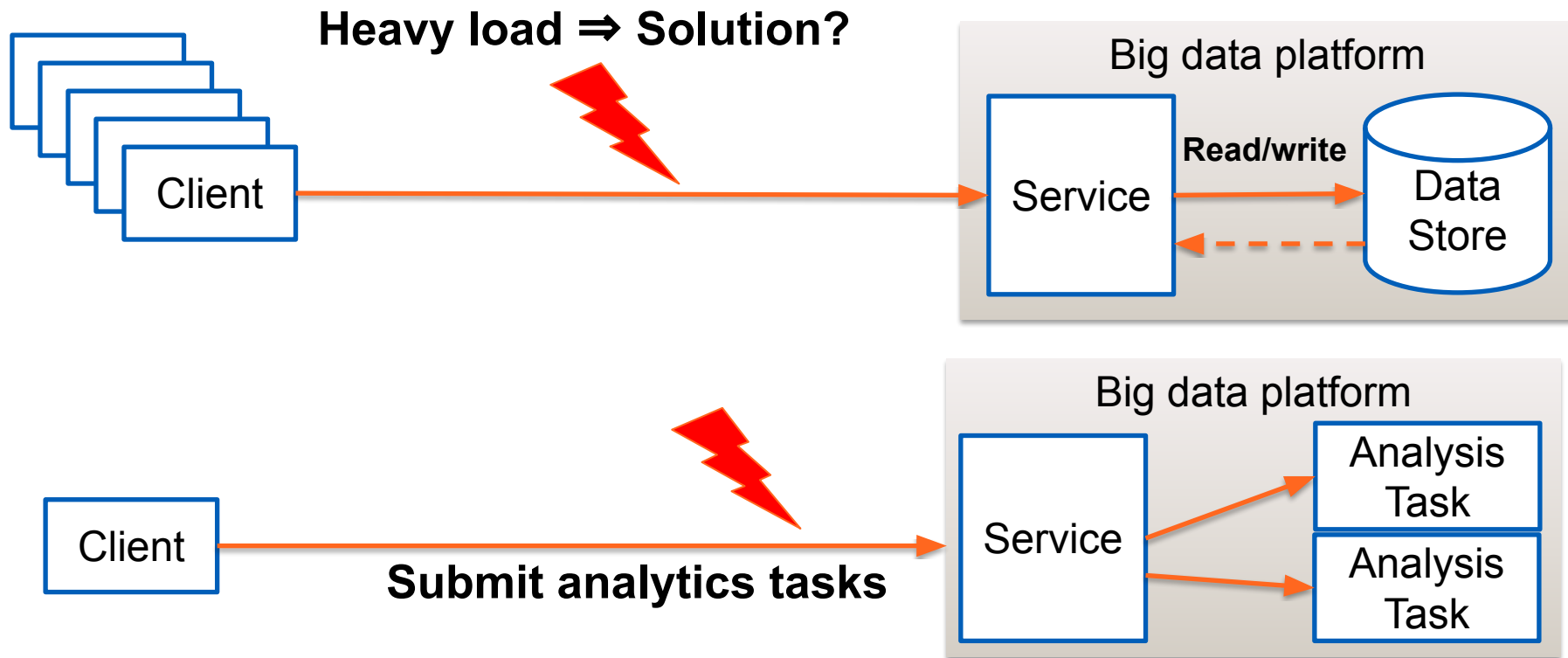
**Check: <https://milinda.pathirage.org/kappa-architecture.com/> &  
<http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html>**

# Recall:

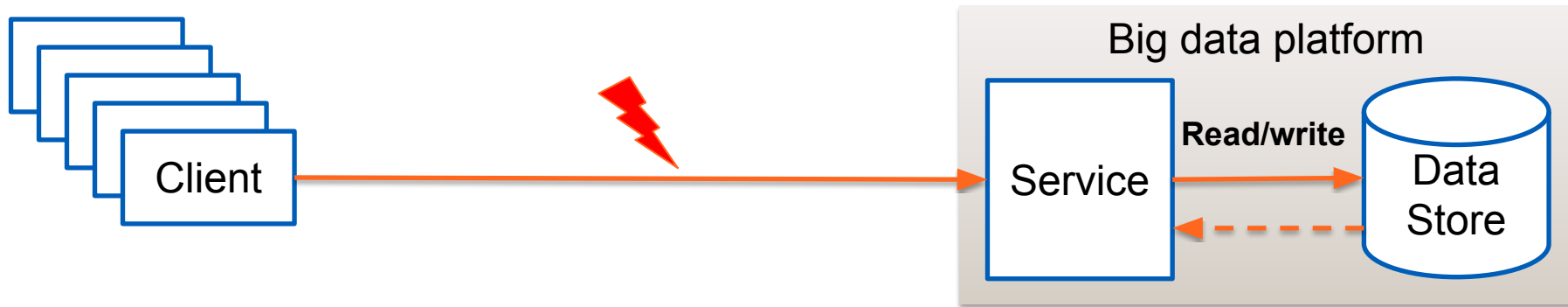
**“A big data platform monitors network usage of devices from million+ customers. We have different levels: **Sensor/Customer, Node (concentrator of multiple customers), Agent (concentrator of multiple Nodes) and the whole network.** In a region, the real operator can generate 1.4 billion records per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

# Optimize data service requests and functionalities

# Concurrent contention



# Back-pressure or elasticity



## Back-pressure: control, drop, and buffer

# Prevent too many accesses?

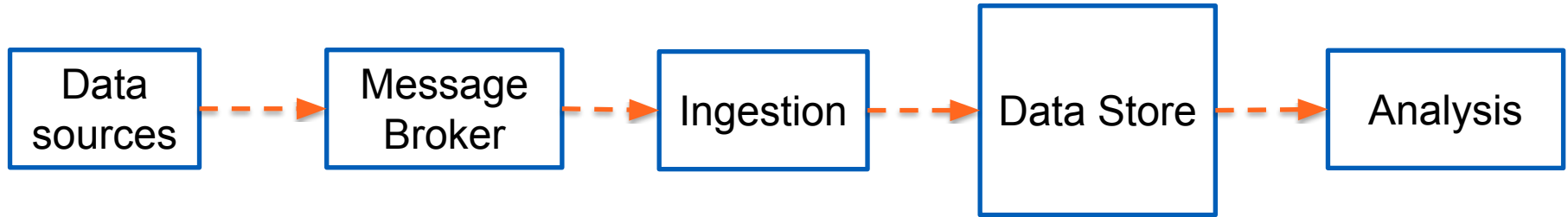


**A related situation:** prevents clients to retry a (failed) operation

<http://martinfowler.com/bliki/CircuitBreaker.html>

<https://msdn.microsoft.com/en-us/library/dn589784.aspx>

# Scaling in every place of big data pipelines



- **Scaling**

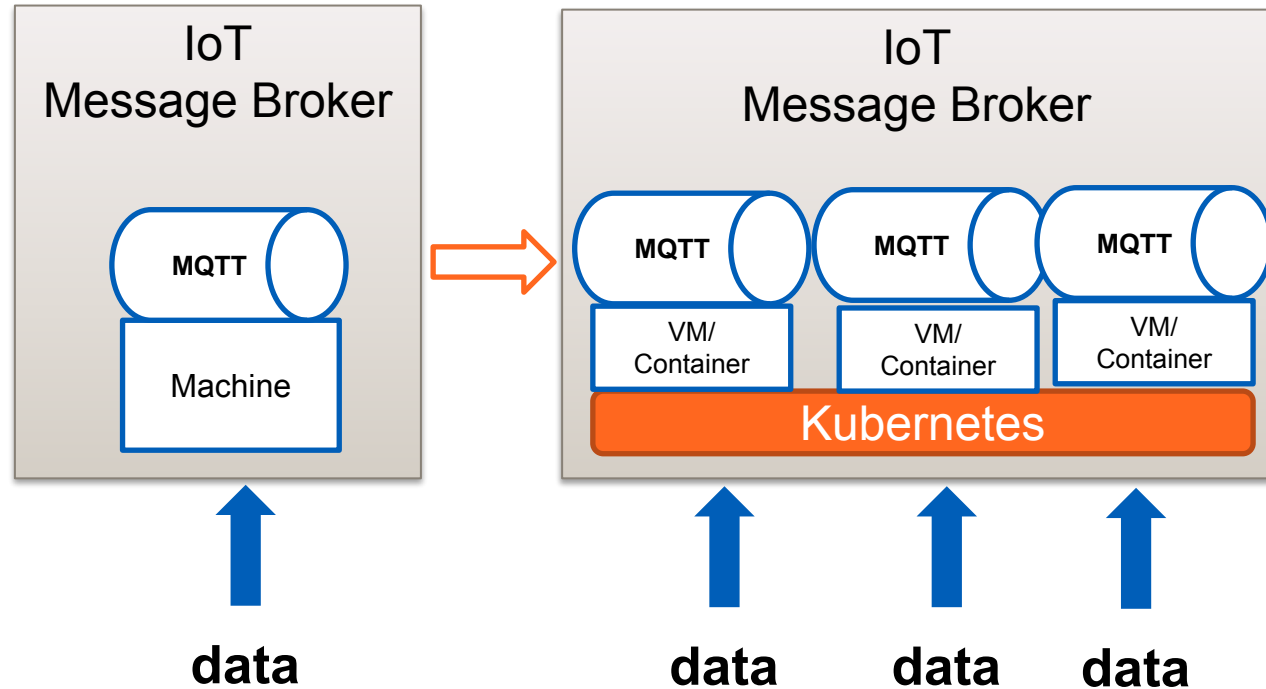
- disk spaces for file storage
- resources for data ingestion
- resources for data analysis

**Happen at  
different times  
and location**



# 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
✓ 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	<input type="text" value="6"/>
Machine type	n1-standard-1 (1 vCPU, 3.75 GB memory)

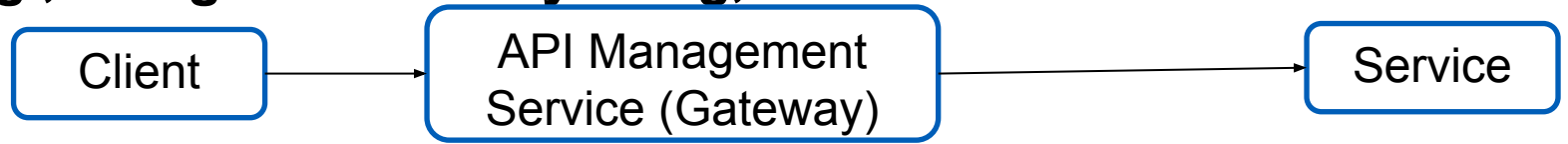
Monitoring Jobs **VM Instances** Configuration Web Interfaces

Name	Role
✓ thebasecluster-m	Master
✓ 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**

# Throttling principle

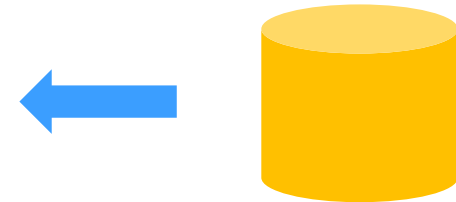
- **Drop strategy: disable too many accesses and disable unessential services**
  - dynamic vs static configuration
- **E.g., using API Gateway Kong, Kubernetes**



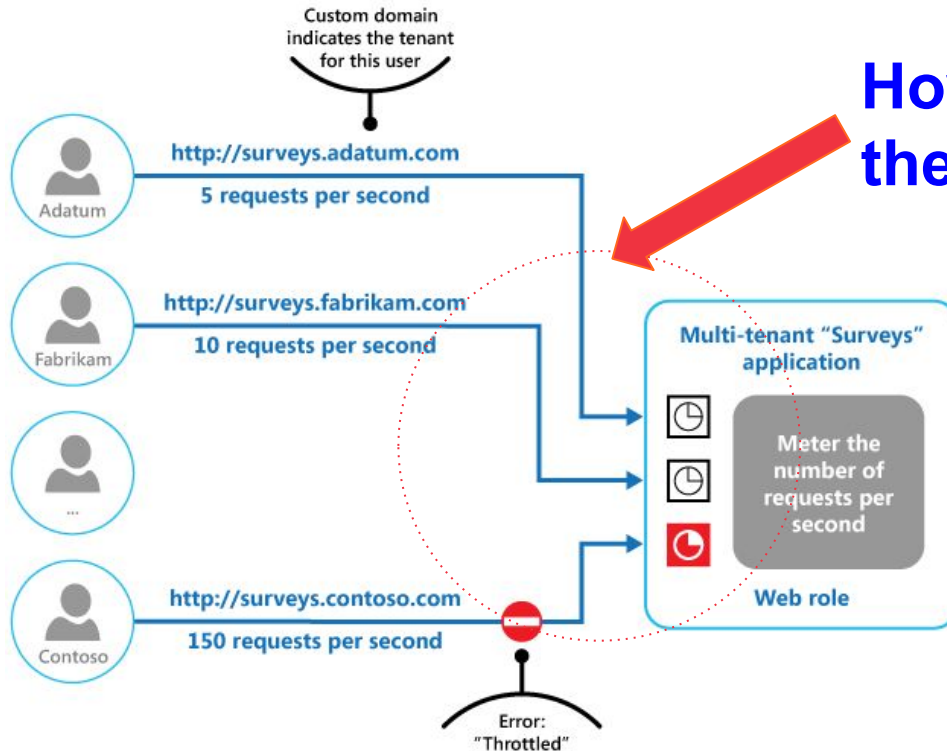
Code:  
<http://www.django-rest-framework.org/api-guide/throttling/#how-throttling-is-determined>

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

**Tenant profile/SLA**



# Example of throttling based on roles



How would you do this in the big data platform?

How this related to your “business service models”/SLA for your platform?

Figure source:

<https://msdn.microsoft.com/en-us/library/dn589798.aspx>

# Using tasks and queue-based load leveling pattern

How this affects the internal design of your big service?

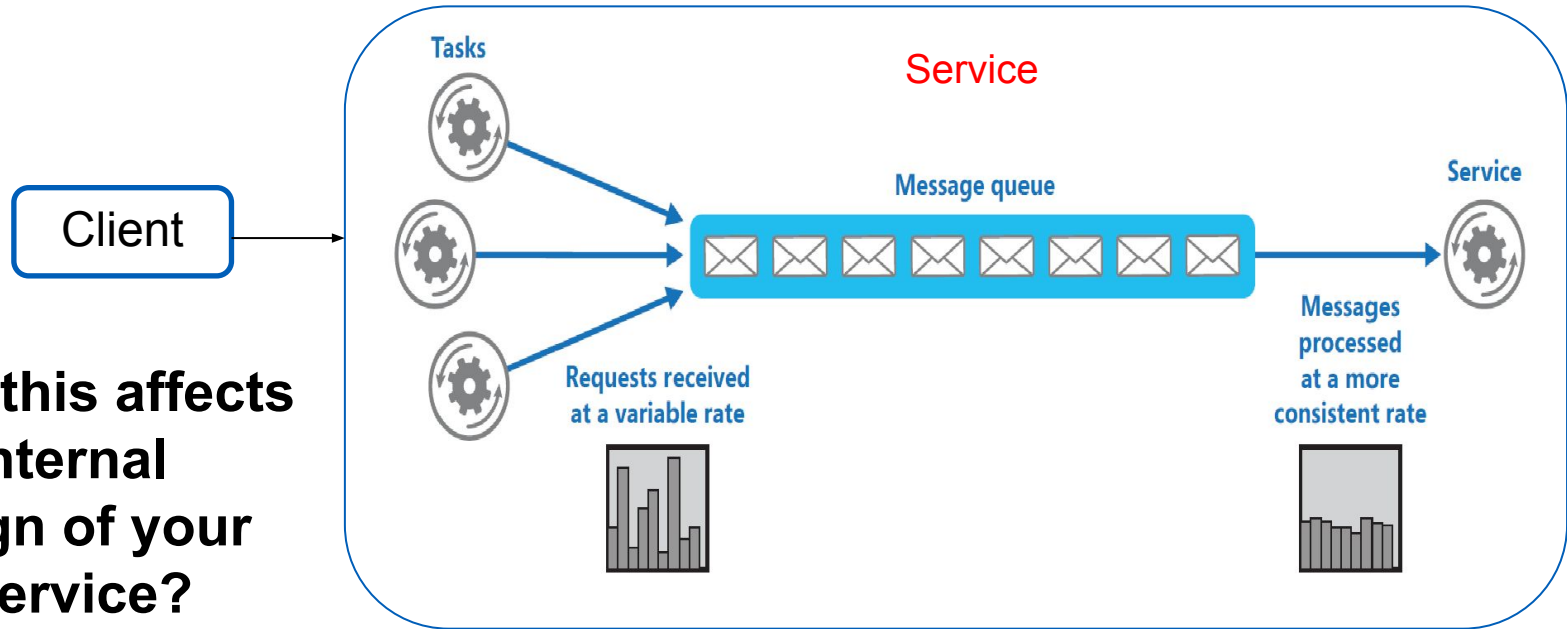
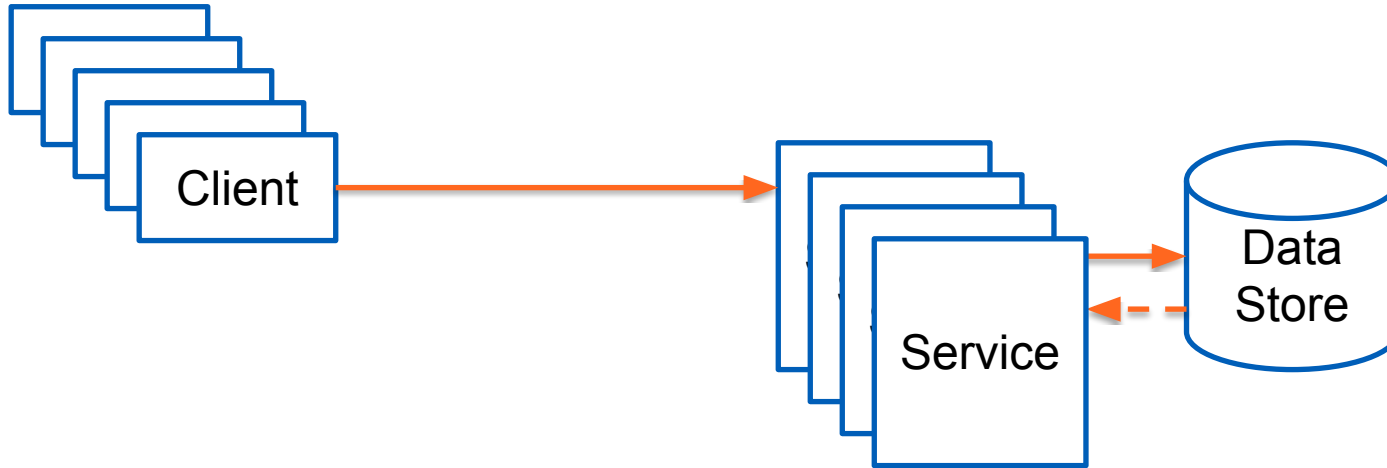


Figure source: <https://msdn.microsoft.com/en-us/library/dn589783.aspx>

# Heavy load between service serving request and data store



**Elastic solution: scale out or up**

# Using multiple instances of services and queues

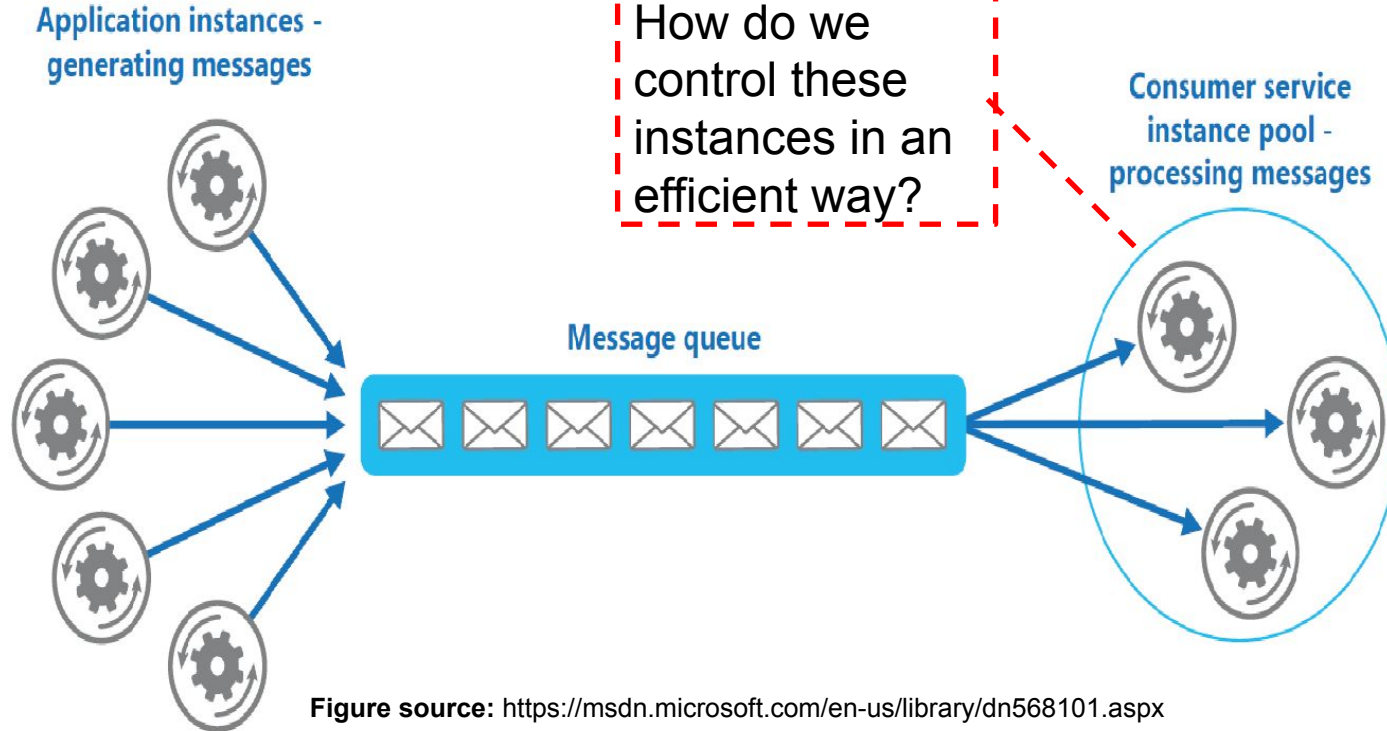
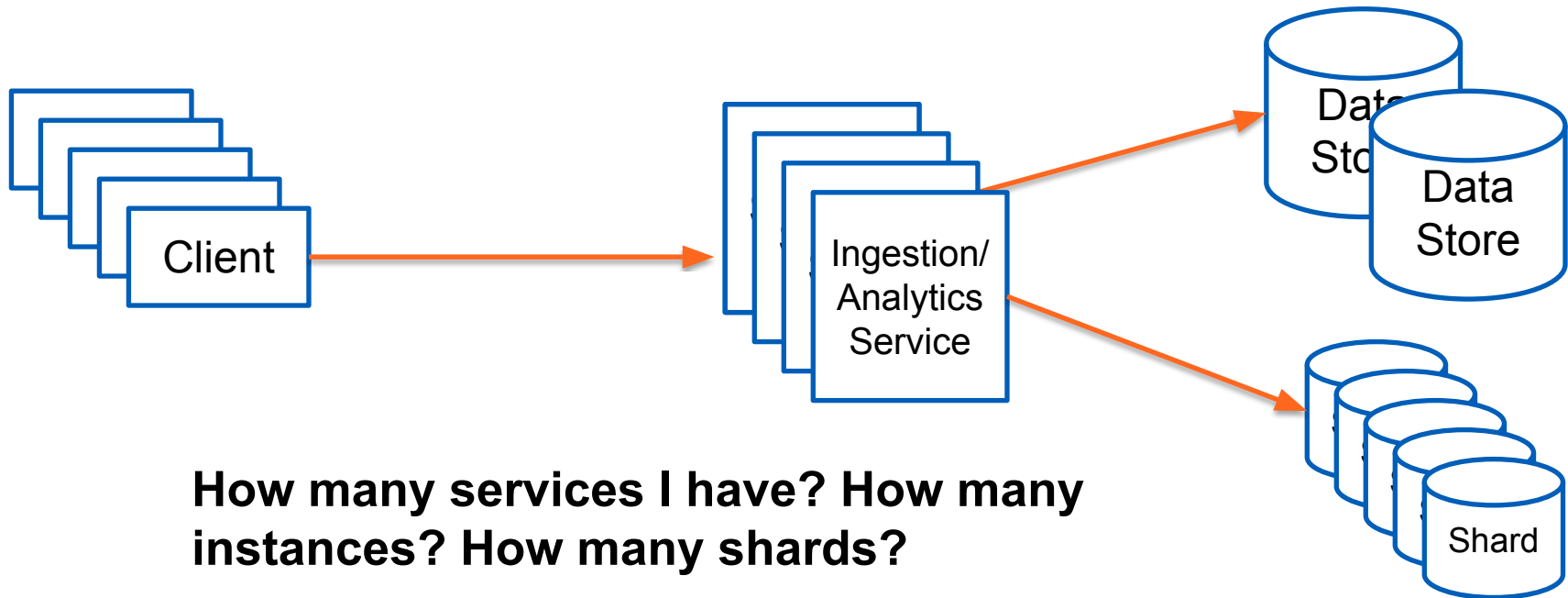


Figure source: <https://msdn.microsoft.com/en-us/library/dn568101.aspx>

# Discovery and consensus

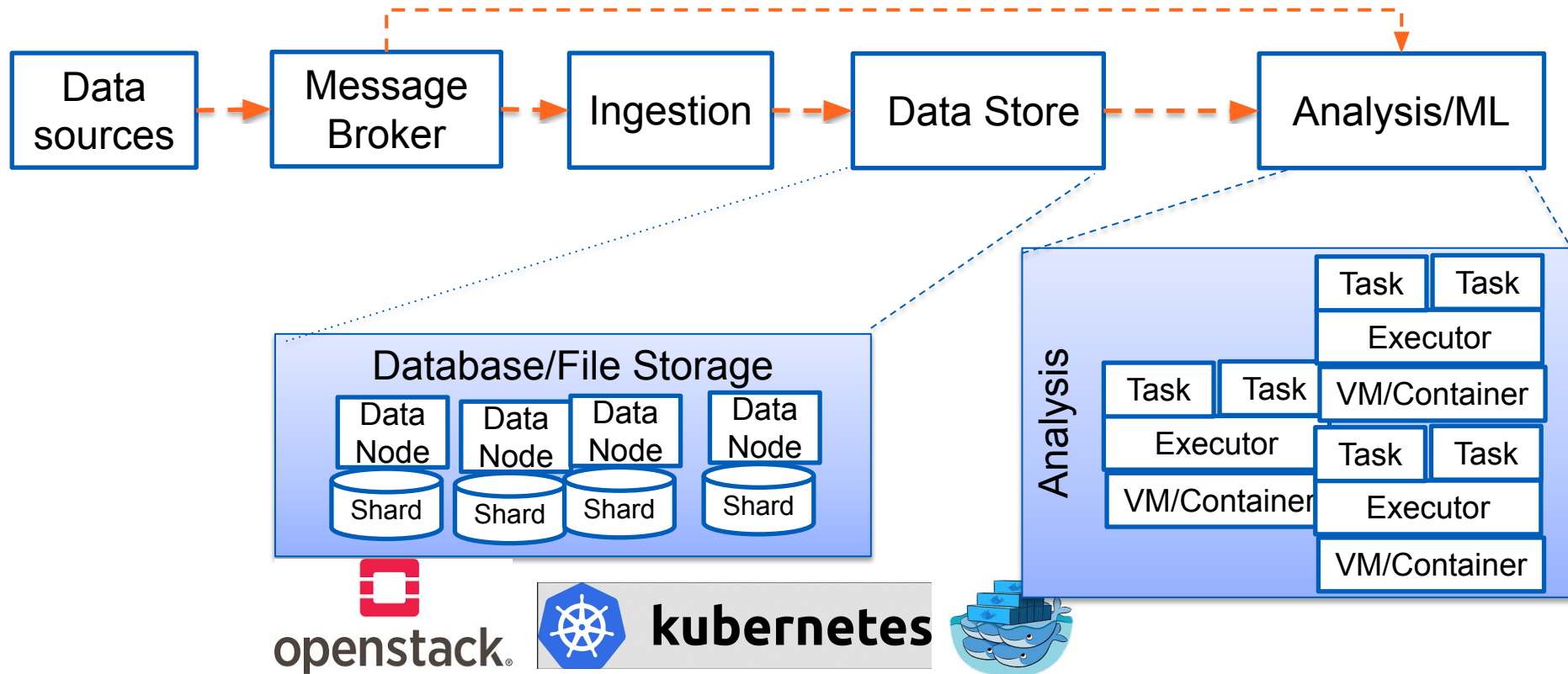


# We can create a lot of instances or we can create new services



**How many services I have? 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**

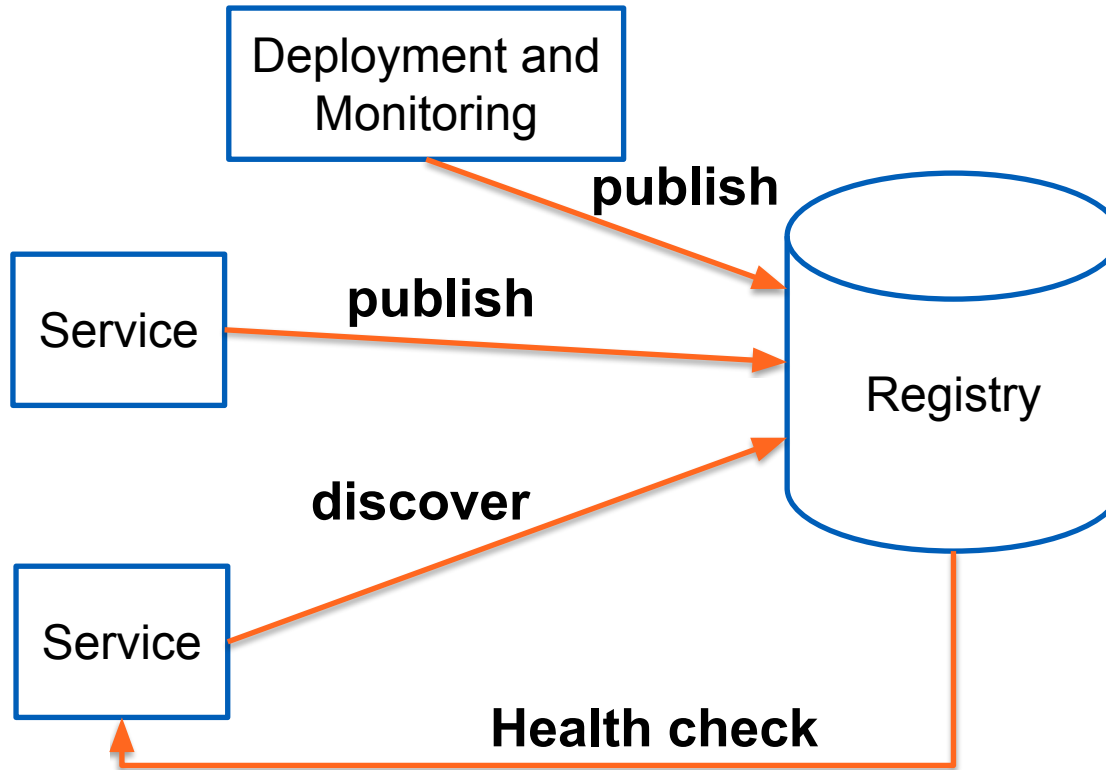
- services themselves
- monitoring component, deployment component, orchestration controllers

- **Lifecycle: very dynamic in elastic environments**

- *Start, run, shutdown, restart, scale*

# Why is it important to know the state of services and what we can do with that?

# Service Discovery principle



- **Key requirements**

- fast
- consistent
- secure
- cross data centers
- simple APIs

# Example:

<https://version.aalto.fi/gitlab/bigdataplatforms/cs-e4640/-/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, e.g. Apache Spark, Hadoop and Kafka**



# Distributed coordination

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

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

Project	Consensus System	Usage Patterns								
		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			✓					✓	
Configurator	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			✓				✓		

What if they do not fit into your big data platforms?

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

# How do we know where are available services and data?

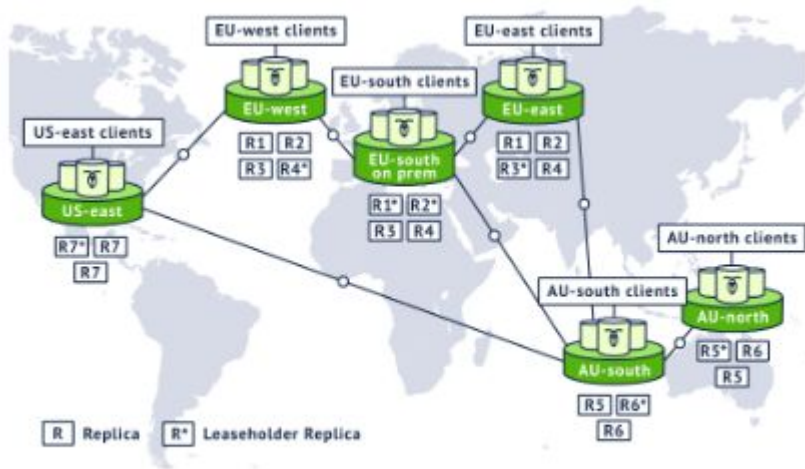


Figure 1: A global CockroachDB cluster

**Figure source:** Taft et al., CockroachDB: The Resilient Geo-Distributed SQL Datasheet, <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/>

# Technology choices: ZooKeeper

- <https://zookeeper.apache.org/>
- Support service discovery, configuration information and distributed synchronization
- Centralized registry service
- Data is organized into a shared hierarchical name space
  - small data size
- Highly available and reliable

# ZooKeeper Service

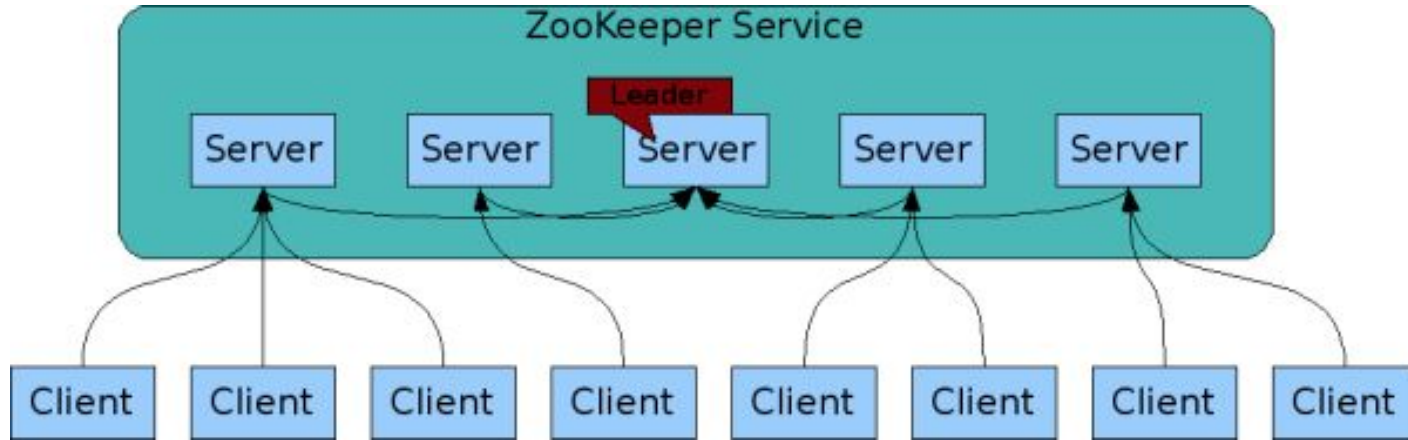


Figure source: <https://zookeeper.apache.org/doc/r3.4.10/zookeeperOver.html>

# ZooKeeper data -- znodes

- Data nodes called znodes
- Missing data in a znode  $\Rightarrow$  problems with the entity that the znode represents
- Persistent znode
  - /path deleted only through a delete call
- Ephemeral znode, deleted when
  - the client created it crashed
  - session expired

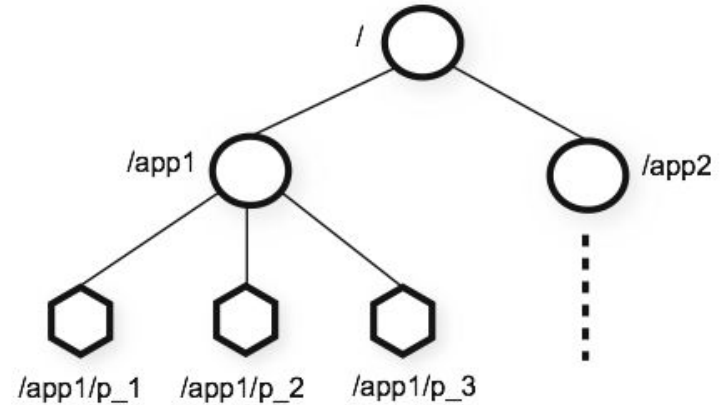


Figure source:

<https://zookeeper.apache.org/doc/r3.4.10/zookeeperOver.html>

# Technology choices: Consul

- <https://www.consul.io>
- Cross data centers
- End-to-end service discovery
  - include health check

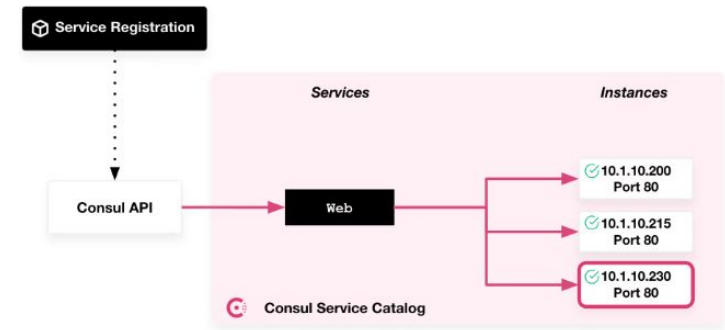


Figure source:

<https://developer.hashicorp.com/consul/docs/concepts/service-discovery>

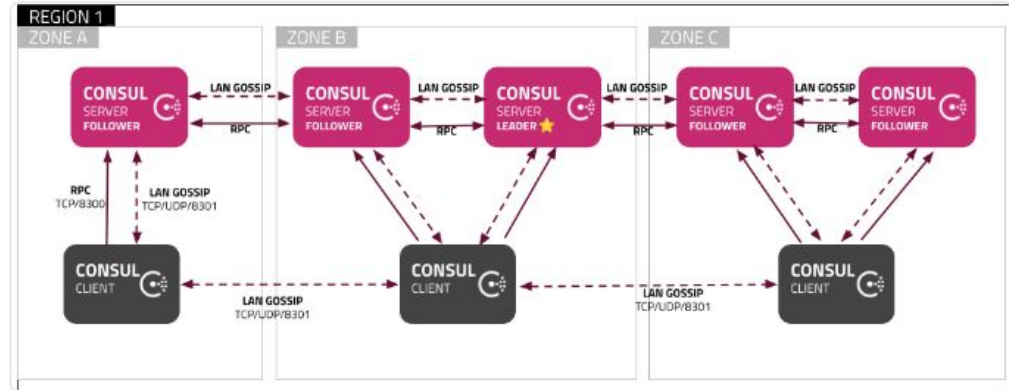


Figure source:

<https://developer.hashicorp.com/consul/tutorials/production-deploy/reference-architecture>

# Technology choices: etcd

- **Consistent, distributed key-value store**
- **Allow monitor changes of keys/directories**
  - enable reactive actions based on changes
- **Widely used for**
  - service discovery and state/configuration management
  - distributed key locking
  - e.g. in Kubernetes

# 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**