MESSAGING AND MICROSERVICES

LEARN FROM KAFKA

AMAZON KINESIS

AZURE EVENT HUBS

GOOGLE PUB/SUB

MICROSERVICES

LEARN FROM KAFKA

LEARN FROM KAFKA
AMAZON KINESIS
AZURE EVENT HUBS
GOOGLE PUB/SUB
MICROSERVICES

APACHE KAFKA

Kafka is a Publisher / Subscriber (Pub-Sub) messaging system

Distributed

- Scales seamlessly

High throughput

- Capable of handling billions of messages per day

Replicated

- Safeguards data in case of machine failures

Created @ LinkedIn in 2010

- Now Apache Project (Open Source)

WHY LINKEDIN BUILT KAFKA?

They had lots of databases

- Built to store data
- Piles of data: relational / key-value / caches / search indexes

What was missing?

- Something to handle the continuous flow of data

Existing message systems like ActiveMQ, were not able to handle the volume LinkedIn was seeing

Hence, LinkedIn built Kafka from scratch

Trivia: Why name it Kafka? Kafka's co-creator Jay Kreps says "Kafka is optimized for writing" and named after popular author Franz Kafka

KAFKA'S GROWTH

Used by tens of thousands of organizations

Including over a third of the Fortune 500

Among the fastest growing open source projects

Immense ecosystem around it

At the heart of a movement towards managing and processing streams of data

KAFKA USE CASES / POWERED BY

LinkedIn

- 200 billion messages / day
- 400 nodes, Multiple data centers (mirroring)
- Used for: website interactions / sending emails / metrics

Netflix

- 80 billion events / day
- 1.5 million events / sec @ peak hours

Spotify

- Event delivery system
- User interactions (add to play list ..etc)

Find more use cases at: BigDataUseCases.info

KAFKA BENCHMARKS

Benchmark	Hardware	Performance
By Linkedin @ 2014	3 machines,- Xeon 2.5 G , 6 cores, - Six 7200 RPM SATA drives, - 32 G RAM, - 1G ethernet	Multiple test setups., One throughput, - 80 MB / sec, - 2 million messages / sec (each message 100 bytes)

The machines are medium scale

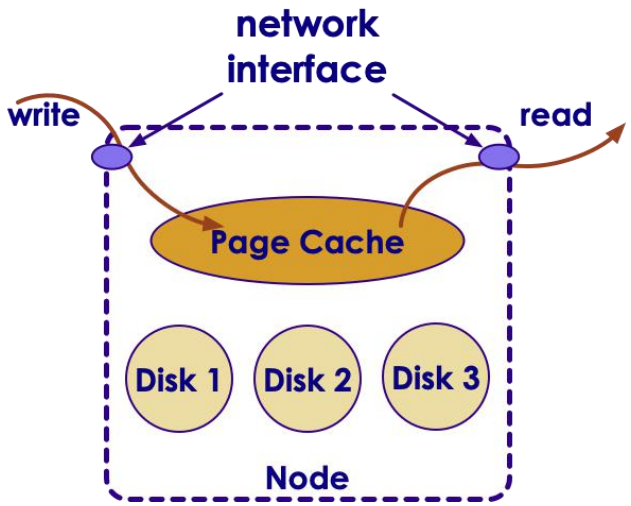
Notice more than one disk per machine

WHY IS KAFKA VERY FAST?

Write: Disk writes are buffered in page cache

Read: The data from page cache can be transferred to network interface very efficiently

99% of the time data is read from page cache, no disk access at all.(source: Loggly)



KAFKA VS. OTHER MESSAGE QUEUES

Message deletion

Other Message Queues:

- In Most systems, after a message is 'read' it can be deleted

Kafka:

- Messages are NOT deleted after they are consumed.
- There is not even a client API to delete a message.
- Messages are expired by Kafka automatically after a certain time (7 days default)
- Deletion is possible if compaction is enabled (more on this later).

Reason: Many applications can subscribe to a topic. A client deleting a message can deprive other clients of their input

KAFKA VS. OTHER MESSAGE QUEUES

Message read order

Other Message Queues:

- Usually messages are read in order
- FF & Rewind are not performant operations

Kafka:

- Messages can be read in any order
- Fast-forward & Rewind are very fast operations

Reason: Clients can choose to skip messages if need to be

KAFKA VS. OTHER MESSAGE QUEUES

Message processing guarantee

Other Message Queues:

- Hard to guarantee one message will only be processed by one client

Kafka:

- Guarantees messages are sent to one consumer
- No duplicate message processing
- (more on this in 'offset management' section)

KAFKA VS. OTHER MESSAGE QUEUES

Concurrency With Multiple Applications

Other Message Queues:

- Usually one / small number of clients can access simultaneously

Kafka:

- Large number of clients can read / write to Kafka at high speed
- No locking / blocking

KAFKA VS. RELATIONAL DATABASES

Feature	RDBMS	Kafka
Data Type	Designed for structured data	Can handle any type of data
Access Type	Can query for individual record (e.g. find customer_id=123)	Designed for sequential scan
Storage Duration	Long term	Short term (typically days)
SQL support	Supports full SQL	KSQL supports limited queries
Query patterns	Can query by any column	Not supported
Indexes	Indexes are fully supported	Not supported
Transactions	Transactions are supported	Not supported

ZOOKEEPER

Distributed service that provides

- Configuration
- Synchronization
- Name registry
- Consensus
- Leader election

Open source

Apache open source project

Battle tested with very large distributed projects

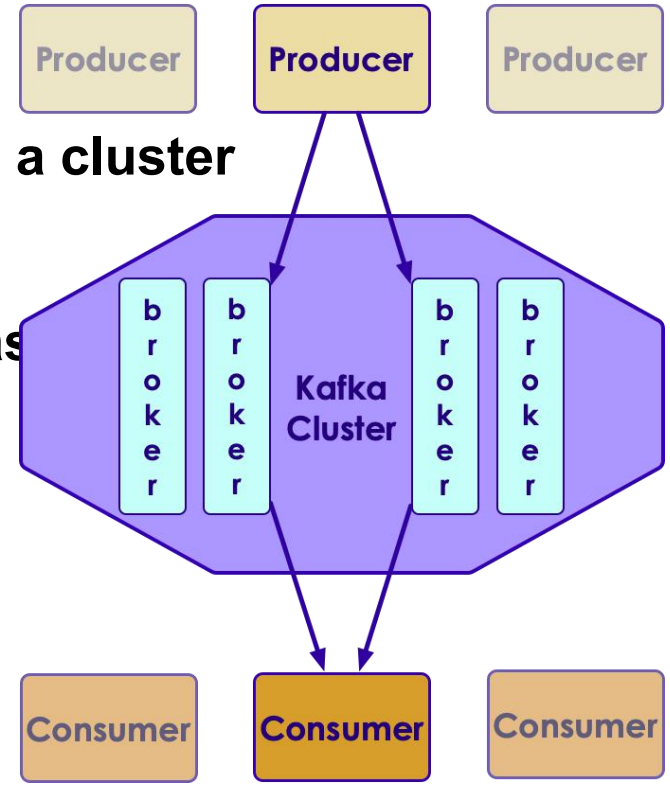
- Hadoop, HBase, Kafka

KAFKA ARCHITECTURE

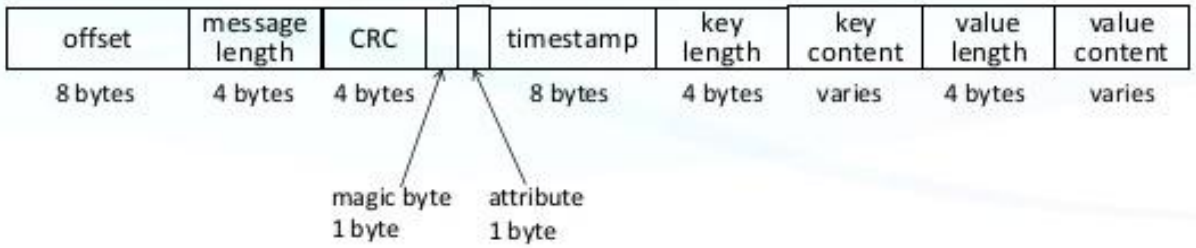
Kafka is designed to run on many nodes as a cluster

Kafka machines are called 'brokers'

Kafka automatically backs up data on at leas
(broker)



KAFKA MESSAGE



In Kafka basic 'data unit' is a message

Kafka treats messages as 'bunch of bytes'

- Doesn't really care what the message payload is

Optionally messages can have metadata, like keys

- Keys are bytes too
- Keys are used to determine which partition to write to
  - Think 'hashing', Same key always go to same partition (more on this later)

Messages can have optional schema

- Uses AVRO to specify schema
- This is for the benefit of clients, Kafka doesn't care about schema

AMAZON KINESIS

LEARN FROM KAFKA
AMAZON KINESIS
AZURE EVENT HUBS
GOOGLE PUB/SUB
MICROSERVICES

AWS KINESIS

collect, process, and analyze

real-time, streaming data

react quickly to new information

KINESIS BENEFITS

Real-time

- ingest, buffer, and process streaming data

Fully managed

- fully managed and runs your streaming applications

Scalable

- can handle any amount of streaming data and process data

KINESIS CAPABILITIES

Kinesis Video Streams

- Capture, process, and store video streams

Kinesis Data Streams

- Capture, process, and store data streams

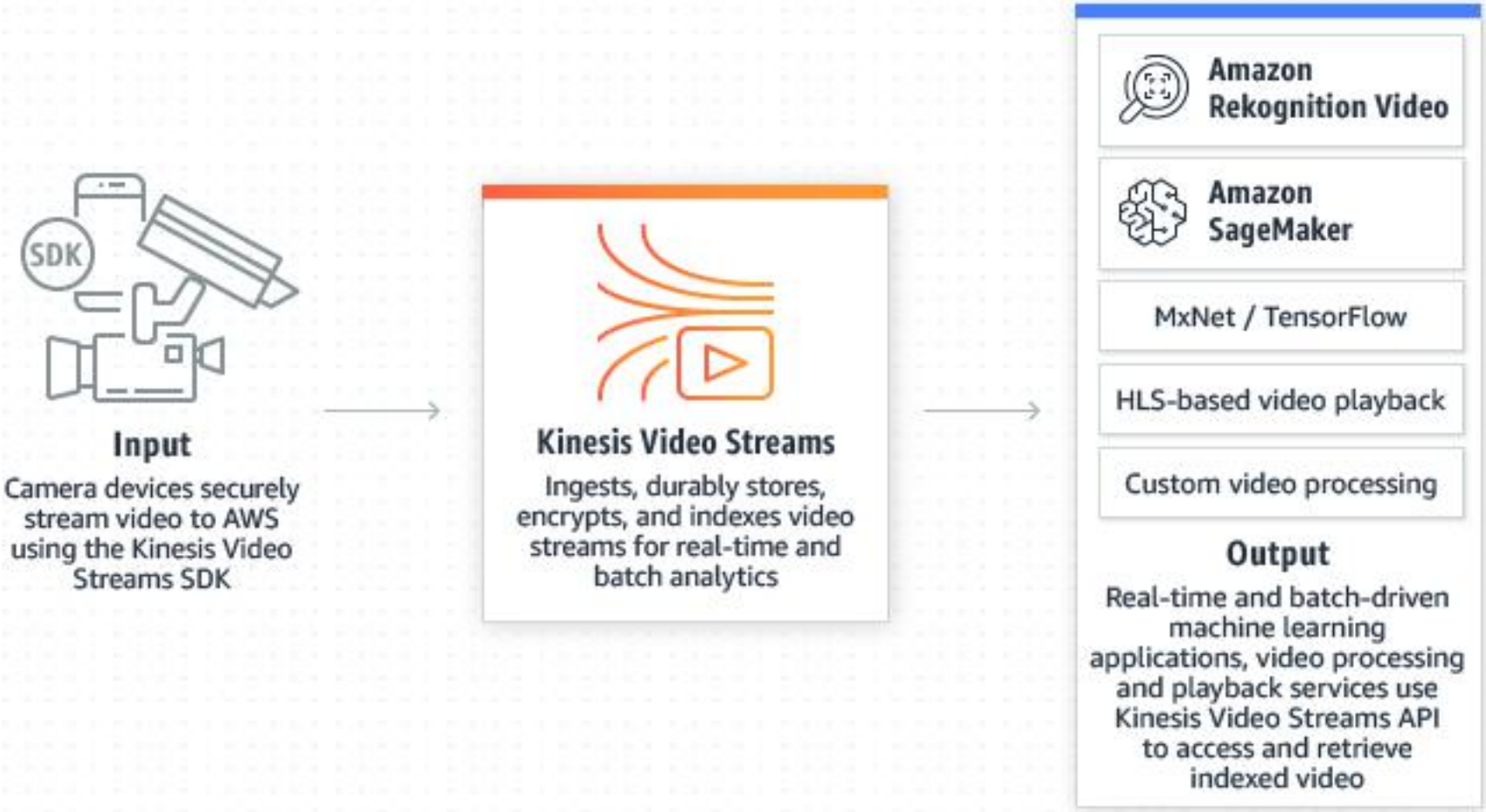
Kinesis Data Firehose

- Load data streams into AWS data stores

Kinesis Data Analytics

- Analyze data streams with SQL or Apache Flink

KINESES VIDEO STREAMS



KINESIS DATA STREAMS

Input

Capture and send data to Amazon Kinesis Data Streams

Amazon Kinesis Data Streams

Ingests and stores data streams for processing

Amazon Kinesis Data Analytics

Spark on EMR

Amazon EC2

AWS Lambda

Build custom, real-time applications using Amazon Kinesis Data Analytics, stream processing frameworks like Apache Spark, or your code running Amazon EC2 or AWS Lambda

Output

Analyze streaming data using your favorite BI tools

KINESIS DATA FIREHOSE

KINESIS DATA ANALYTICS

AZURE EVENT HUBS

LEARN FROM KAFKA
AMAZON KINESIS
AZURE EVENT HUBS
GOOGLE PUB/SUB
MICROSERVICES

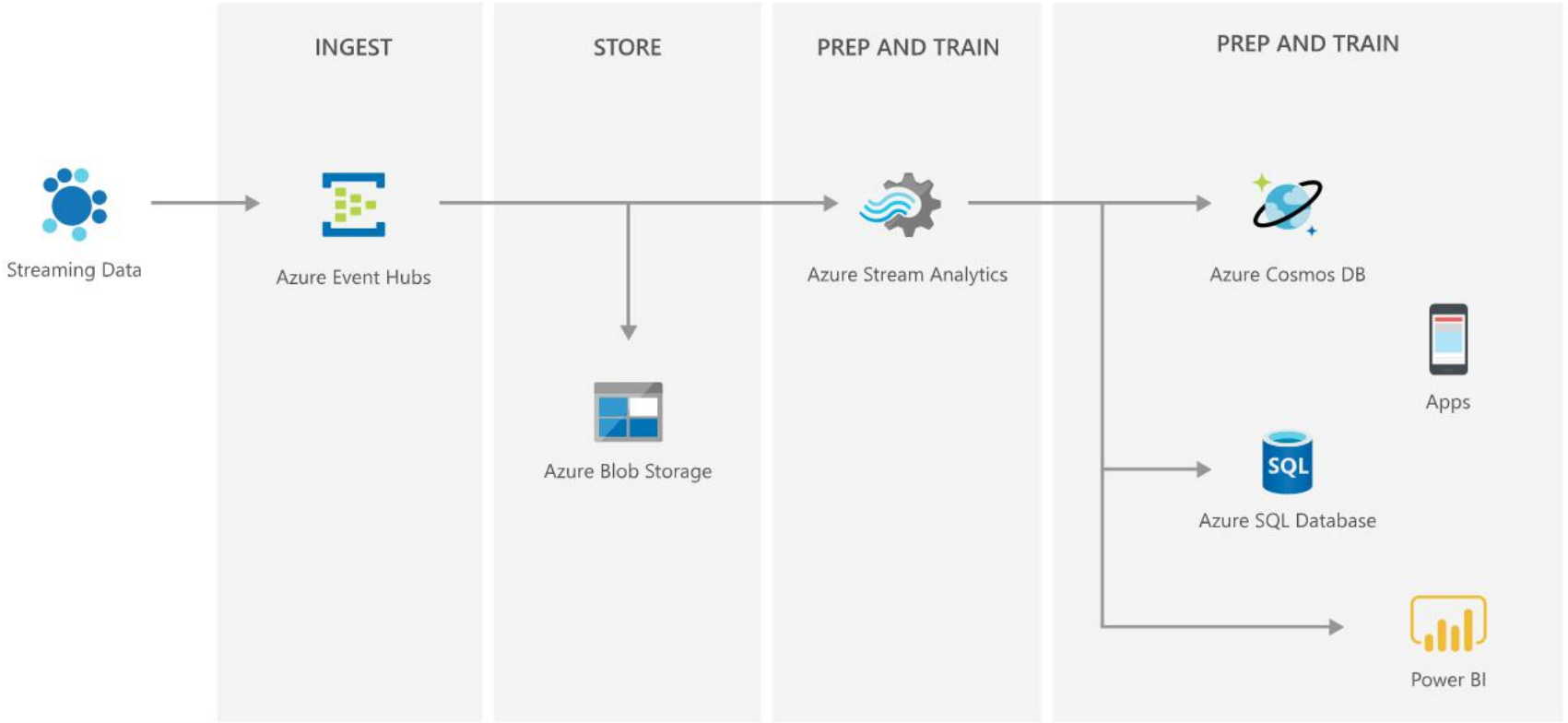
AZURE EVENT HUBS

Simple, secure, and scalable real-time data ingestion

Event Hubs

- fully managed
- real-time data ingestion service
- Keep processing data during emergencies using the geo-disaster recovery

EVENT HUBS PLATFORM



EVENT HUBS FEATURES

Ingest millions of events per second

Enable real-time and micro-batch processing concurrently

Get a managed service with elastic scale

Easily connect with the Apache Kafka ecosystem

Build a serverless streaming solution

Ingest events on Azure Stack Hub and realize hybrid cloud solutions

GOOGLE PUB/SUB

LEARN FROM KAFKA
AMAZON KINESIS
AZURE EVENT HUBS
GOOGLE PUB/SUB
MICROSERVICES

DECOUPLING WITH PUB/SUB

Use Kinesis on AWS

Event Hub on Azure

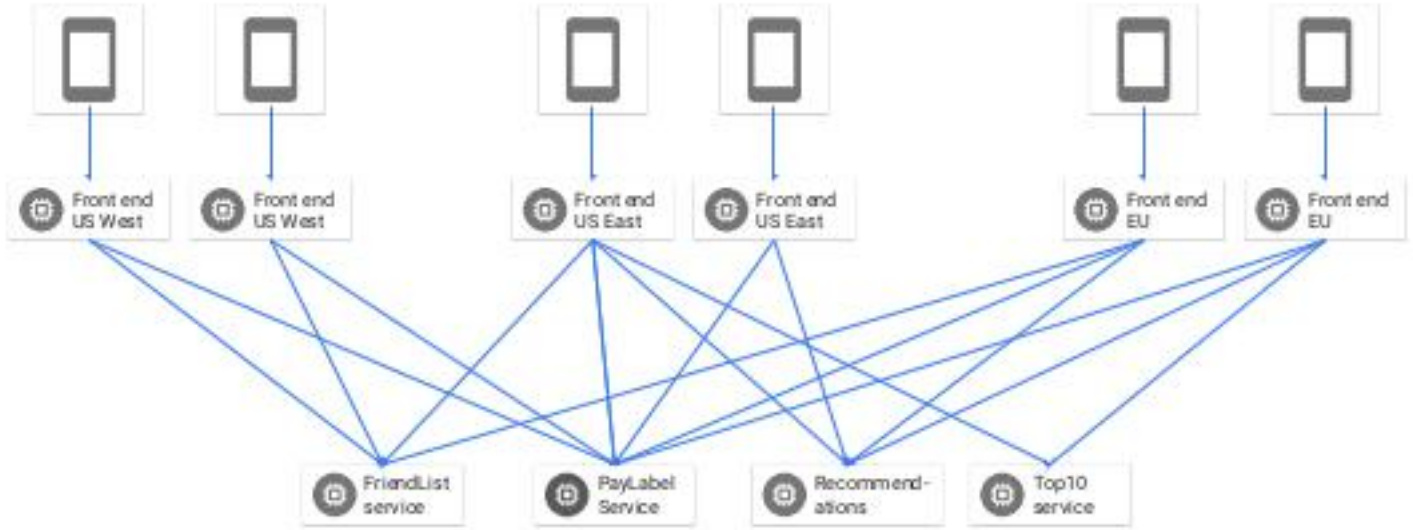
Using Google Cloud Pub/Sub to Integrate Components of Your Application

GOAL

Organizations have to rapidly ingest, transform, and analyze massive amounts of data

ORCHESTRATE

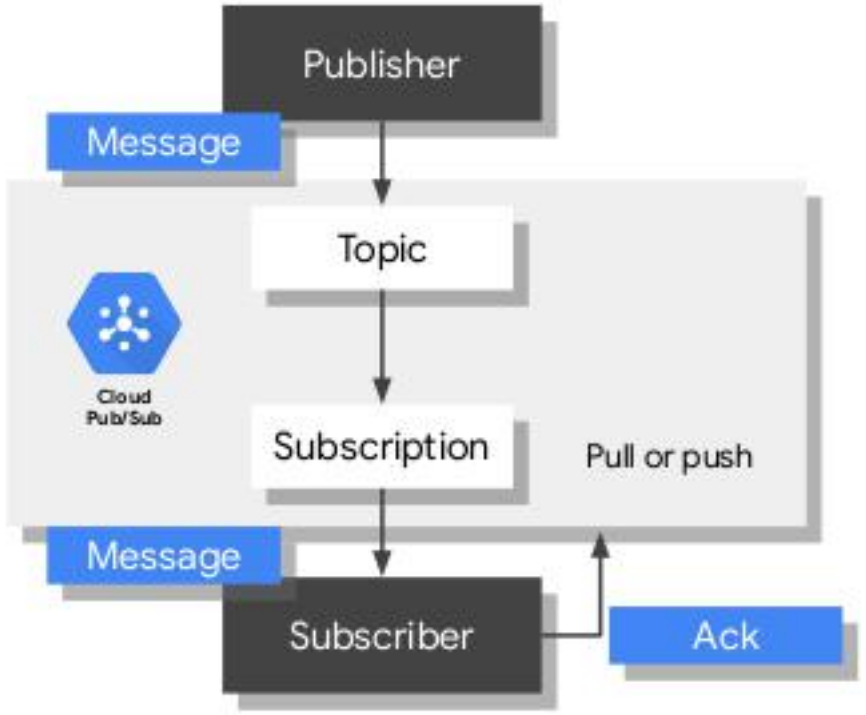
Organizations have to orchestrate complex business processes



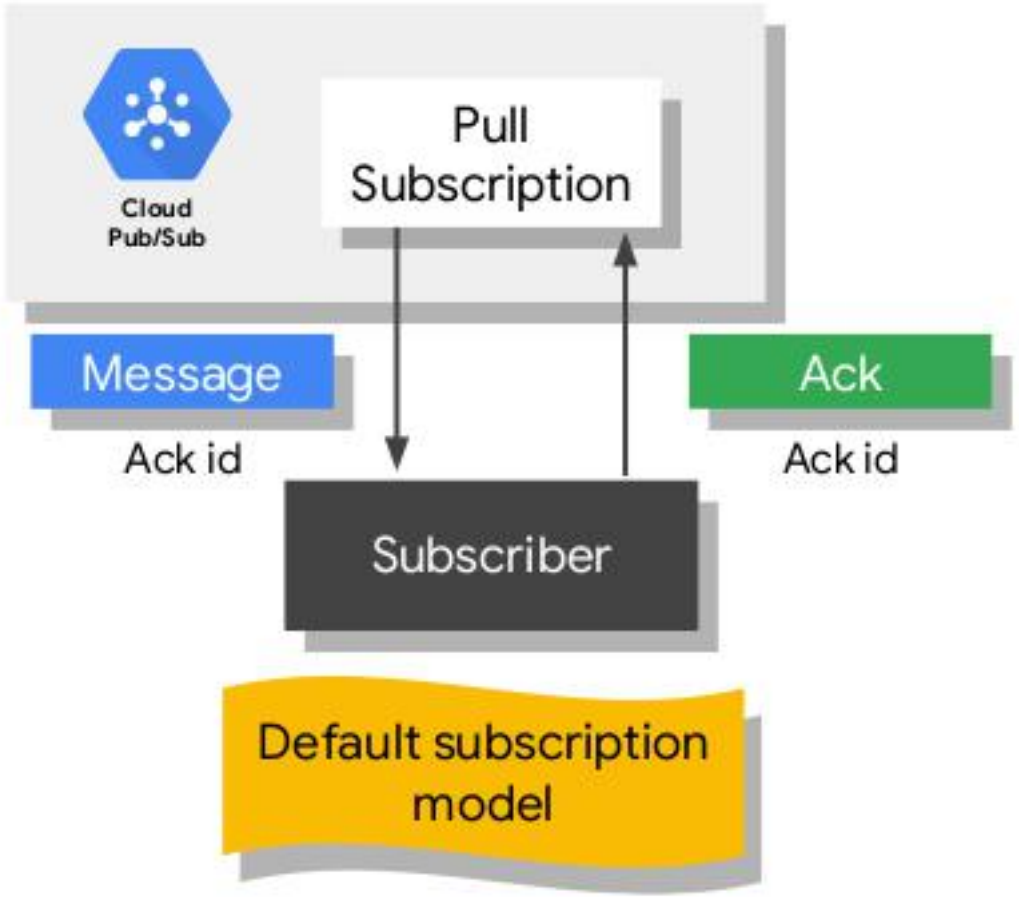
PUB/SUB

Cloud Pub/Sub is a fully managed real-time messaging architecture

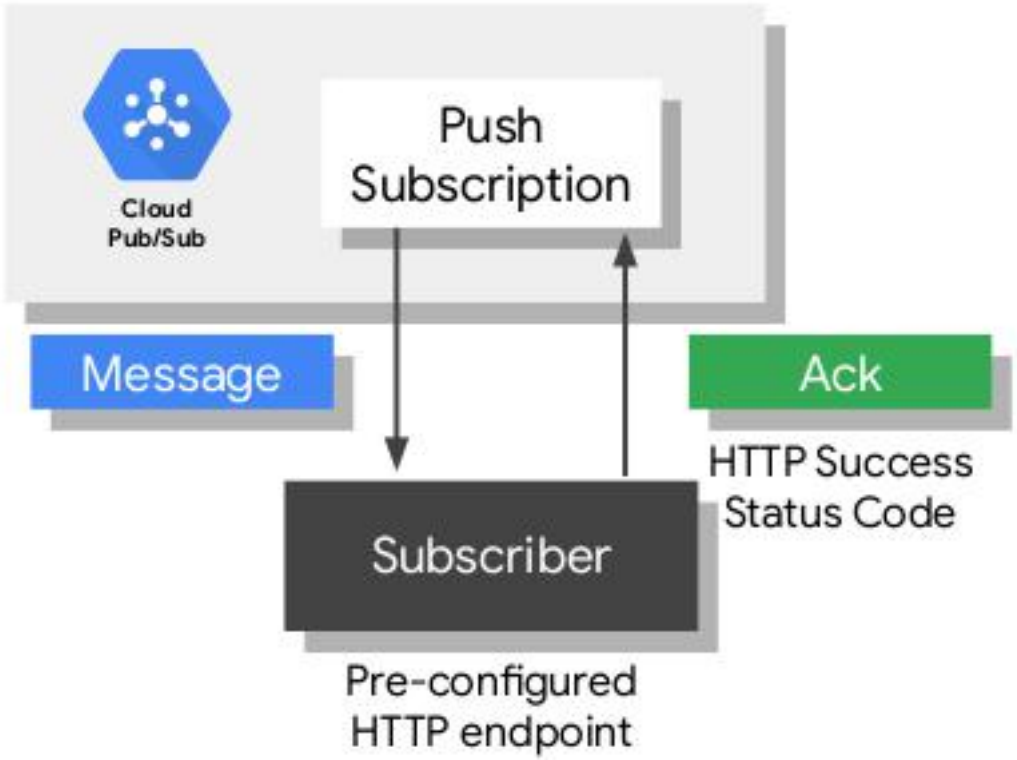
Cloud Pub/Sub delivers each message to every subscription at-least-once.



PULL AND PUSH



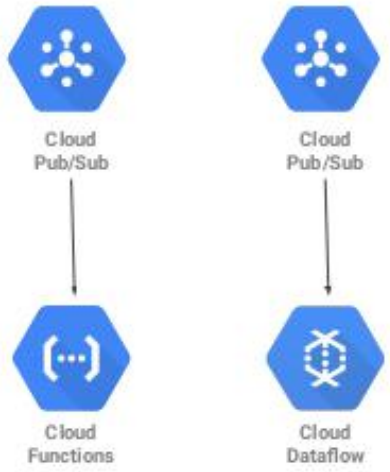
PUSH



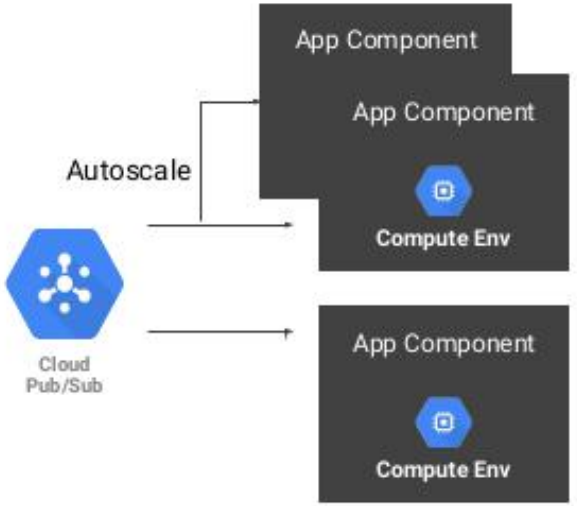
SUBSCRIBERS

You have a choice of execution environments for subscribers

Develop highly-scalable subscribers with Google Cloud Functions or Cloud Dataflow.

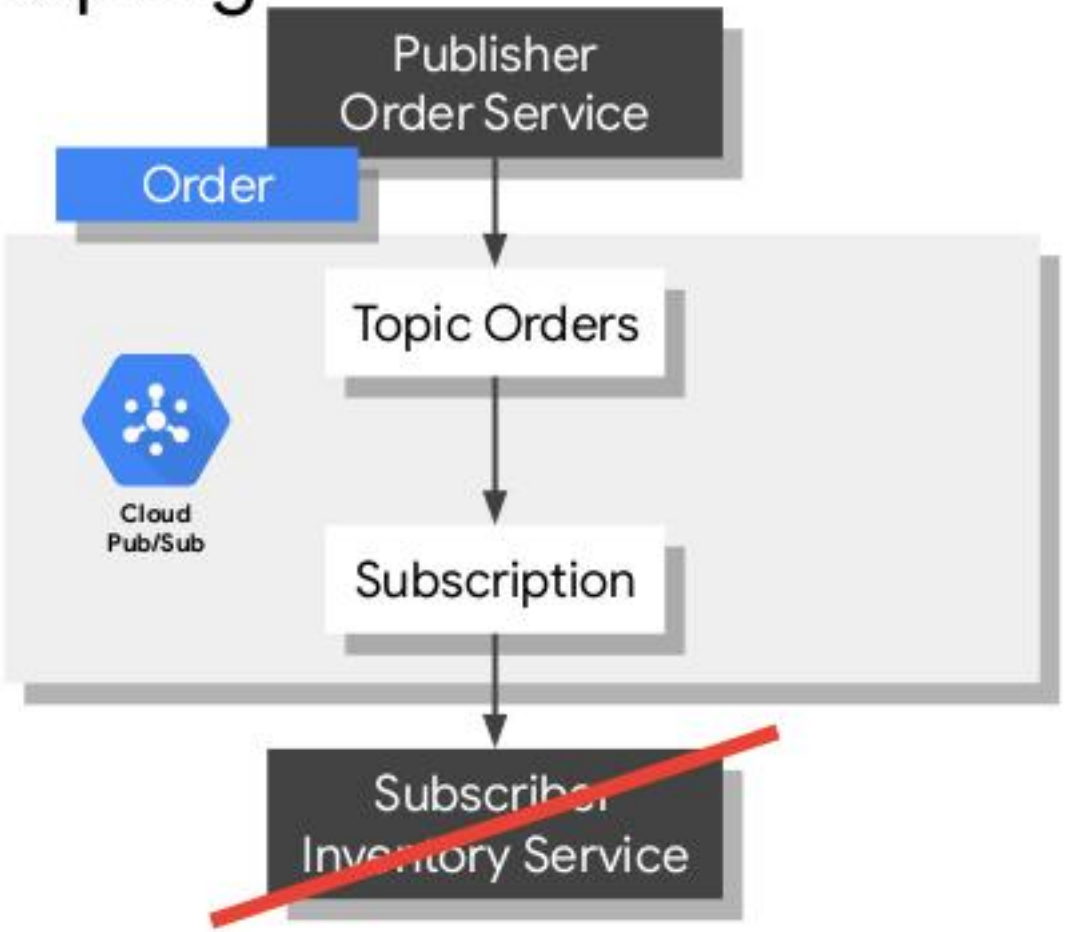


Deploy subscriber on Google Compute Engine, Google Kubernetes Engine, or Google App Engine flexible environment. Autoscale based on Google Stackdriver metrics.



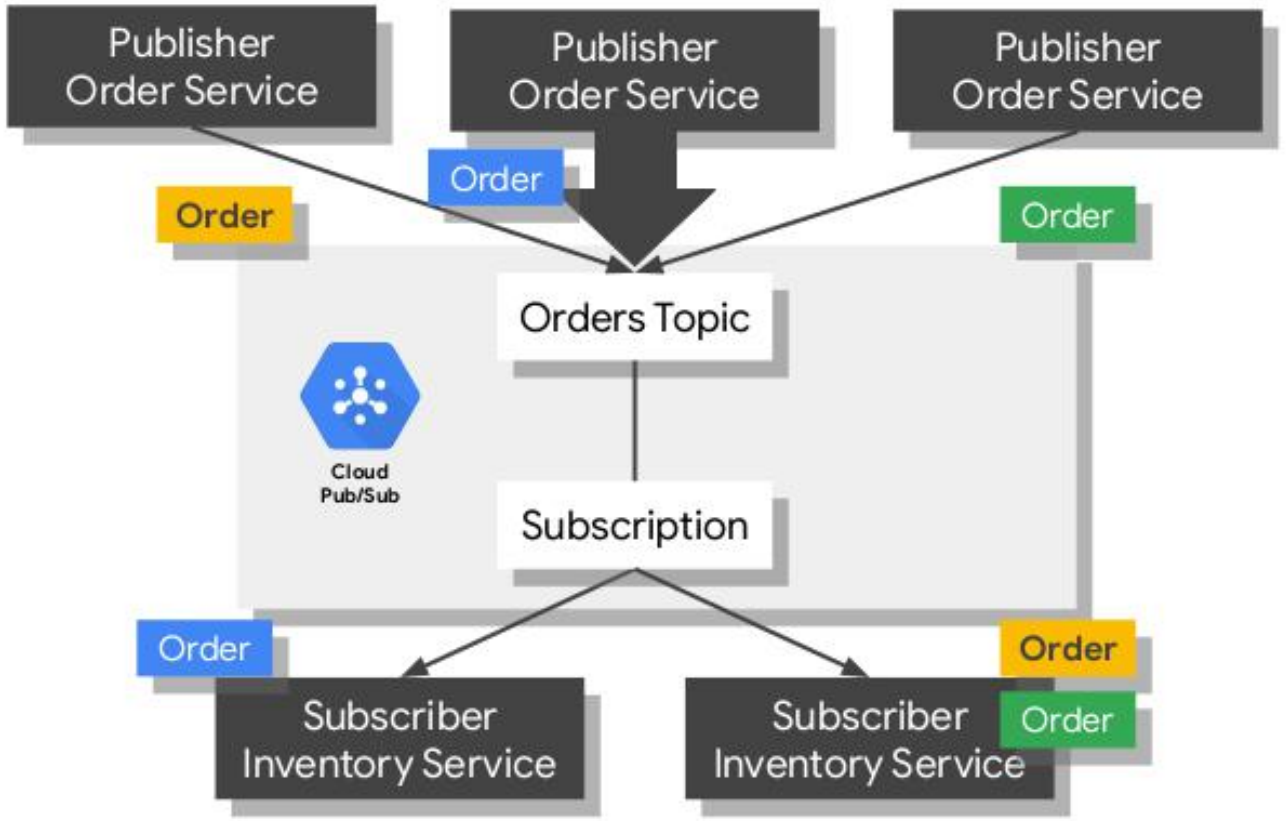
LOOSE COUPLING

Use Cloud Pub/Sub for asynchronous processing and loose coupling



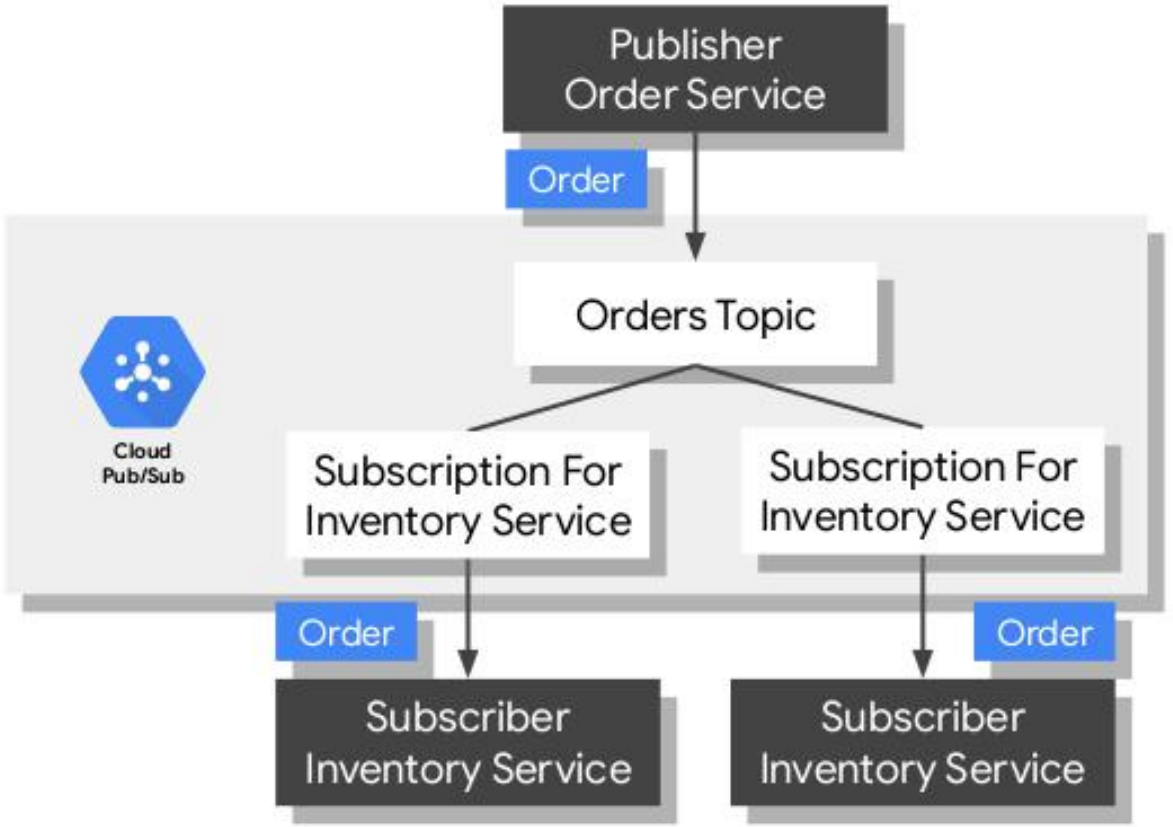
BUFFER

Use Cloud Pub/Sub to buffer incoming data



FAN OUT

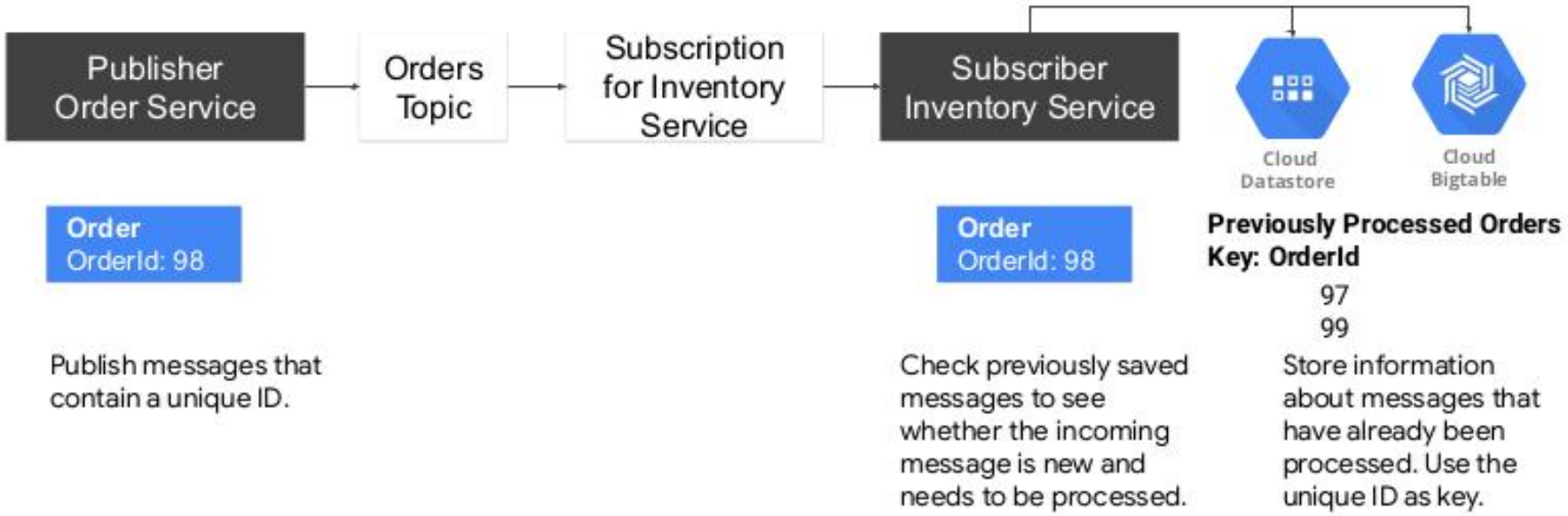
Use Cloud Pub/Sub to fan out messages



DUPLICATE MESSAGES

Handle duplicate messages

Ensure that messages contain identifying attributes that subscribers can use to perform idempotent operations.



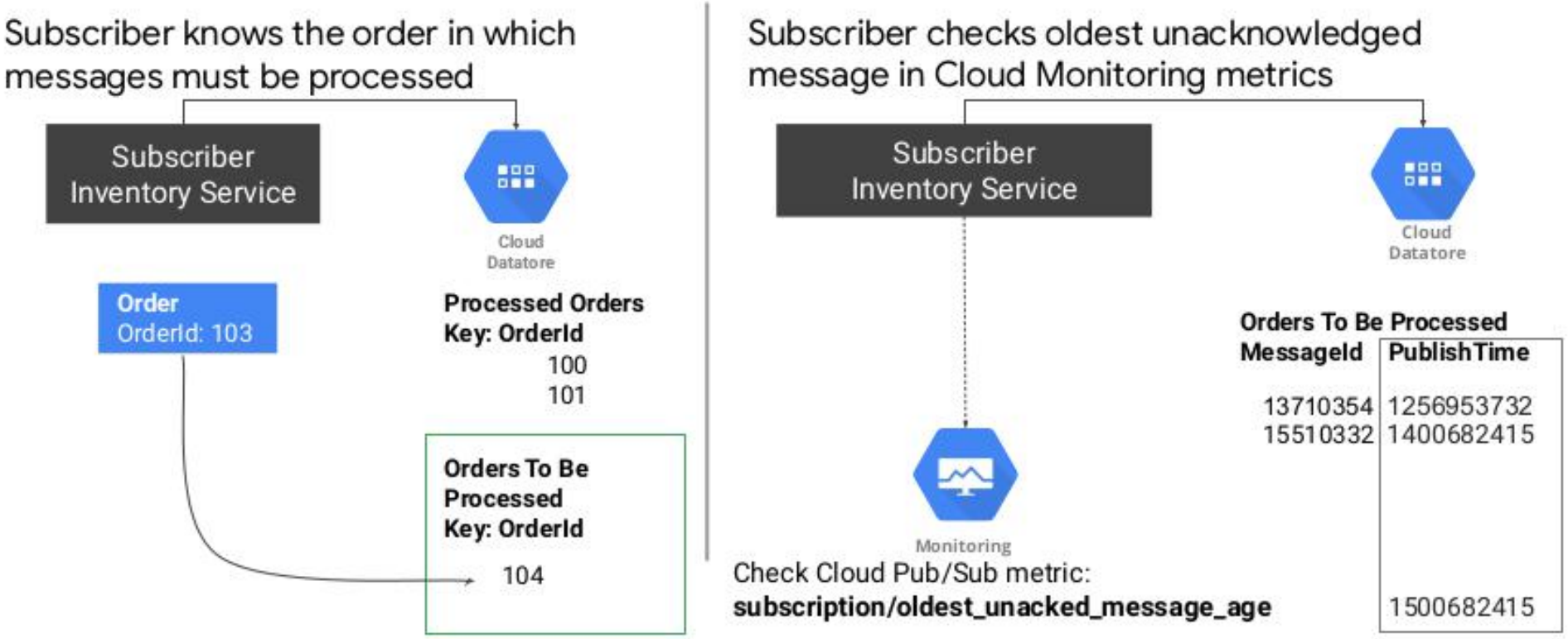
SCALABILITY

For scalability, reduce, or eliminate dependencies on message ordering

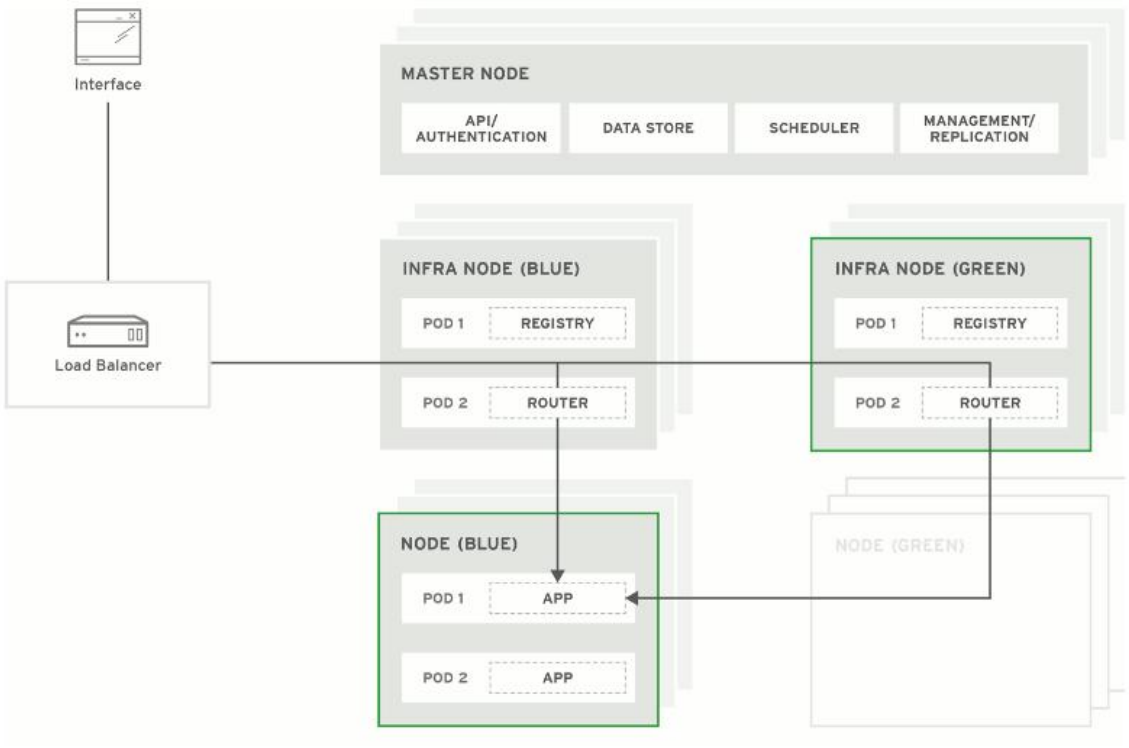
Scenario: Ordering is irrelevant	Scenario: Final order is important; processed message order is not	Scenario: Processed message order is important
Examples: • Collection of statistics about events • Notification when somebody comes online	Examples: • Log messages	Examples: • Transactional data such as financial transactions • Multi-player network games

TRANSACTIONS

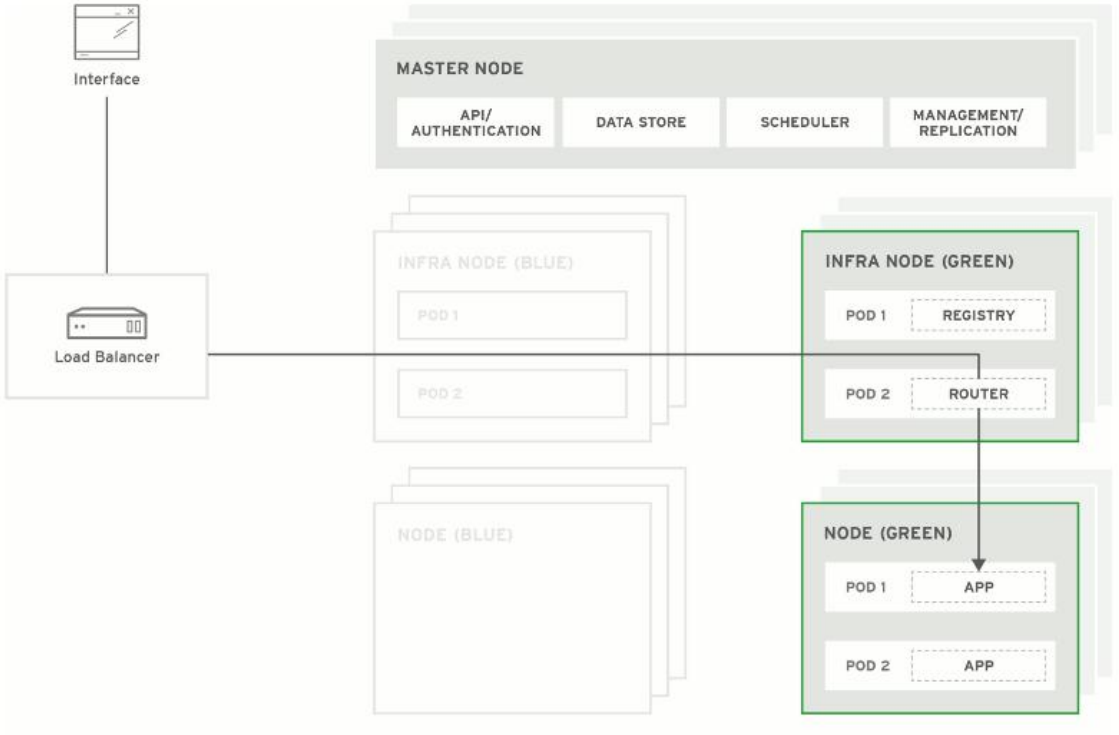
Handle message ordering for transactional data



EXAMPLE: BLUE GREEN DEPLOYMENT



EXAMPLE: BLUE GREEN DEPLOYMENT



MICROSERVICES

LEARN FROM KAFKA
AMAZON KINESIS
AZURE EVENT HUBS
GOOGLE PUB/SUB
MICROSERVICES

MONOLITHIC SERVICES

Traditional Services are defined as monolithic

Monolithic Service Does all of the following:

- Presentation
- Business Logic
- DB Access
- Application Integration

DISADVANTAGES TO MONOLITHIC SERVICES

Complexity to fully understand

Difficult to Scale

Reliability

Difficult to Test

Tight Coupling.

Violation of Single Concern

Difficult to Re-use components.

CONWAY'S LAW

Melvin Conway (1967) stated Conway's Law:

- "Organizations which design systems -- are constrained to produce designs which are copies of the communication structures of those organizations"

What does this mean?

- it means that are architecture will naturally match our own business structure
- 3 business units means likely three components

What's wrong with that? - Should we match our *own* business structure?
- What about the business structure of our users??

ERIC RAYMOND'S HACKER DICTIONARY

Eric Raymond re-states Conway's Law.

- "If you have four groups working on a compiler; you will get a four-stage compiler"

The organization of the *product* is congruent with the organization of the organization

CONWAYS LAWS:

Law 1: Communication dictates design

The mode of organizational communication is expressed through system design

Law 2: There is never enough time to do something right, but there is always enough time to do it over

A task can never be done perfectly, even with unlimited time, but there is always time to complete a task

Law 3: There is a homomorphism from the linear graph of a system to the linear graph of its design organization

Homomorphism exists between linear systems and linear organizational structures

Law 4: The structures of large systems tend to disintegrate during development, qualitatively more so than small systems

A large system organization is easier to decompose than a smaller one

CONWAY'S FIRST LAW

"Human beings are complex social animals."

For a complex system, design topics always involve communication between human beings.

"The Mythical Man-Month".

“Adding manpower to a late software project makes it later” — Fred Brooks, (1975)

Why?

Communication cost increases exponentially with the number of people in a project:

- Communication Cost: `n(n-1)/2` , or `O(n²)`
- 5 members team: `5*(5-1)/2 = 10` channels
- 15 member team: `15*(15-1)/2 = 105` channels
- 50 member team: `50*(50-1)/2 = 1,225` channels

Dunbar Number: (Robin Dunbar)

- Human brains seem wired to have about 150 friends (5 of whom are intimate friends)

CONWAY'S SECOND LAW

"Rome was not built in a day. Address the issues that can be addressed first."

“Problem too complicated? Ignore details. Not enough resources? Give up features.” – Erik Hollnagel (2009)

Solution:

- Reslient
- Fault-Tolerant

CONWAY'S THIRD LAW

"Create independent subsystems to reduce the communication cost."

Business Boundaries create small systems

"Inter-operate, don't integrate"

- Interoperate: Define System Boundareis and interface
- Full stack to entire team
- Complete Autonomy

WHAT ARE MICROSERVICES?

"Do One Thing and Do It Well"

- Each microservice does minimal level of useful functionality

Application Composed of interconnecting services

- Services communicate via RPC or Messaging
- Services do their own persistence

Services are Testable

- Services have their own testing strategy
- Functionally

UNIX PHILOSOPHY

Small is beautiful.

Make each program do one thing well.

Build a prototype as soon as possible.

Choose portability over efficiency.

Store data in flat text files.

Use software leverage to your advantage.

Use shell scripts to increase leverage and portability.

Avoid captive user interfaces.

Make every program a filter.

DISTRIBUTED SYSTEMS

Microservices are the "UNIX Philosophy"

- in distributed systems
- in services

Does this mean that I can only do this on Linux?

- NO!
- "UNIX Philosophy" can be used on Windows too!
- It's just an idea

MICROSERVICE ADVANTAGES

Scalability

- Functionality already distributed
- Easy to Break it up as deployment

Testability

- Each portion is minimal so easy to test
- TDD

Re-usability

- Components have to focus on doing one task well.
- We avoid cluttering components with application specific code.

MICROSERVICE ARCHITECTURE PRINCIPLES

Elastic

Resilient

Composable

Minimal

Complete

PRINCIPLE 1: ELASTIC

Must be able to scale up or down

Multiple Stateless Instances of Service

Routing and Load Balancing

Registration, Naming, and Discovery

PRINCIPLE 2: RESILIENT

Service will have multiple Instances

High Availability

- Redundancy
- Fault Tolerance

No Single Point of Failure

- No one service is indispensable
- Load/Risk Distribution

Service Instance Dynamic

PRINCIPLE 3: COMPOSABLE

Common, Uniform Interface

REST Principles

Composition Patterns:

- aggregation
- linking
- caching
- proxies
- gateways

PRINCIPLE 4: MINIMAL

The "Micro" in Microservices!

Entities should be cohesive

SRP (Single Responsibility Principle

- **One** business function

Not necessarily small in size

- (See next slide)
- But as as small as possible.

PRINCIPLE 5: COMPLETE

Minimize coupling With Other Services

- Tight Coupling limits re-usability.

Should fully accomplish business function

- Don't "split" services for the sake of it
- Each module is as big as it needs to be
- "Micro" doesn't **always** mean small

MICROSERVICES AND CONTAINERS

Container frameworks facilitate microservices.

Kubernetes Pods

- Allows groups of containers to run together.
- facilitates microservices.

CONGRATS ON COMPLETION