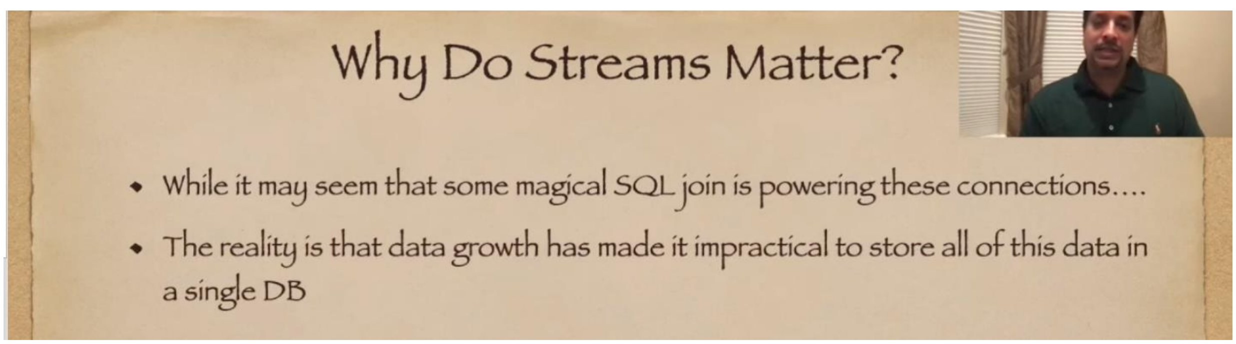
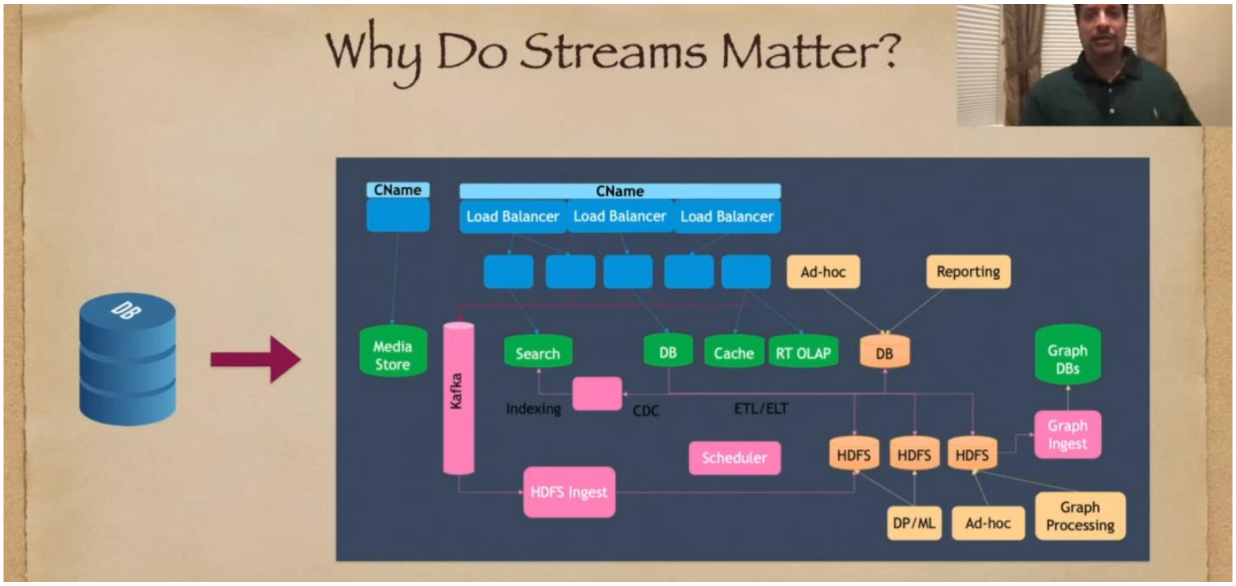
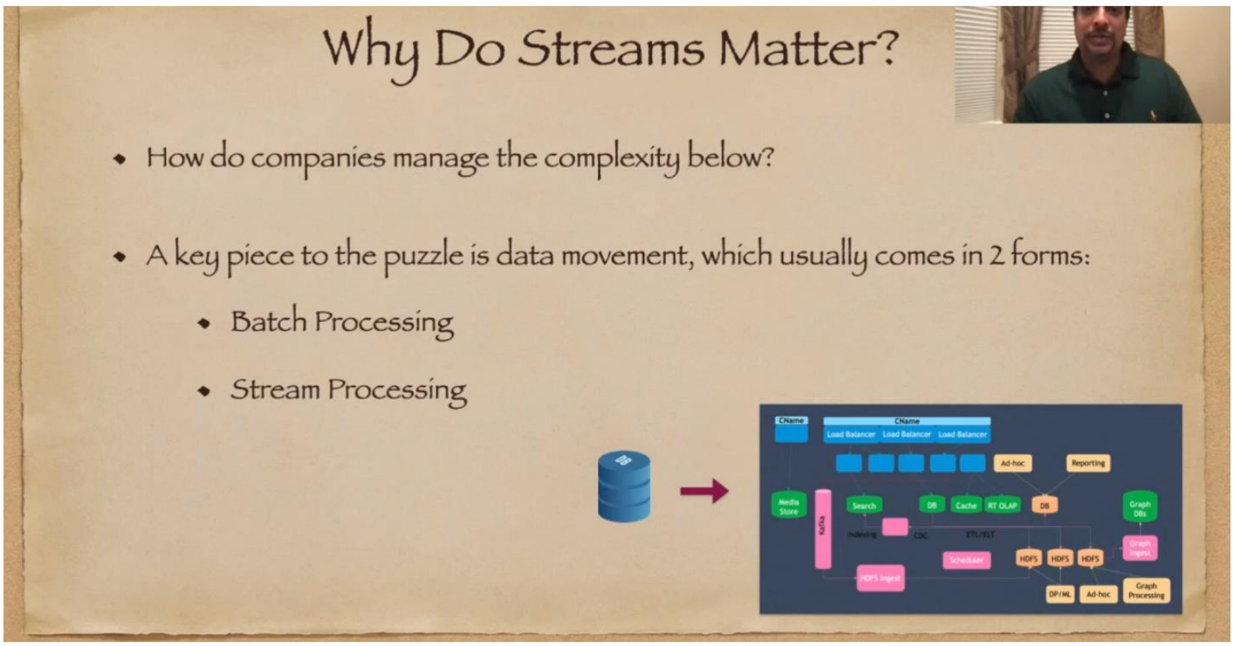
TRACK
Modern Data Architectures, Pipelines, & Streams

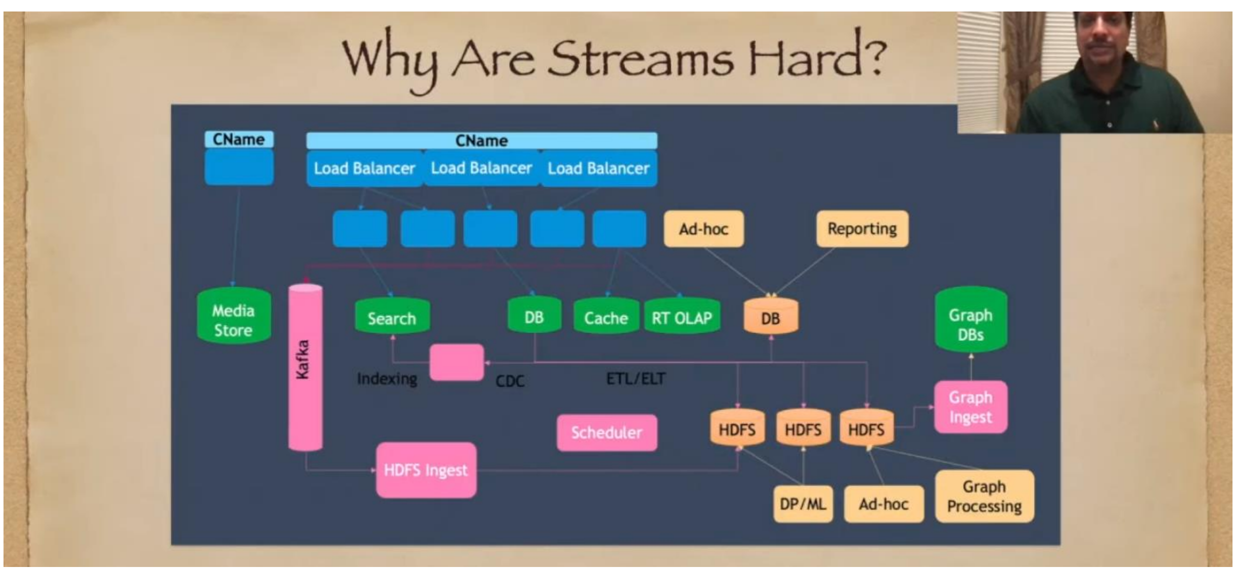
SESSION
Building & Operating High-Fidelity Data Streams

Sid Anand
Chief Architect @Datazoom, PMC @ApacheAirflow

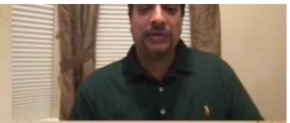



Why Do Streams Matter?
DB
CName
CName
Load Balancer
Load Balancer
Load Balancer
Ad-hoc
Reporting
Media Store
Kafka
Search
DB
Cache
RT OLAP
DB
Graph DBs
Indexing
CDC
ETL/ELT
Graph Ingest
Scheduler
HDFS
HDFS
HDFS
HDFS Ingest
DP/ML
Ad-hoc
Graph Processing


Why Do Streams Matter?
How do companies manage the complexity below?
A key piece to the puzzle is data movement, which usually comes in 2 forms:
Batch Processing
Stream Processing

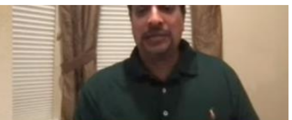

Why Are Streams Hard?
CName
CName
Load Balancer
Load Balancer
Load Balancer
Ad-hoc
Reporting
Media Store
Kafka
Search
DB
Cache
RT OLAP
DB
Graph DBs
Indexing
CDC
ETL/ELT
Graph Ingest
Scheduler
HDFS
HDFS
HDFS
HDFS Ingest
DP/ML
Ad-hoc
Graph Processing

Why Are Streams Hard?

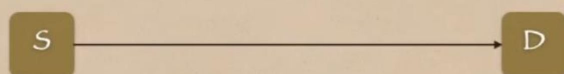


- In streaming architectures, any gaps in non-functional requirements can be unforgiving
- You end up spending a lot of your time fighting fires & keeping systems up
- If you don't build your systems with the -ilities as first class citizens, you pay an operational tax
- ... and this translates to unhappy customers and burnt-out team members!
- In this talk, we will focus on building high-fidelity streams from the ground up!

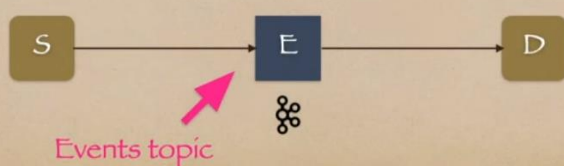
Start Simple



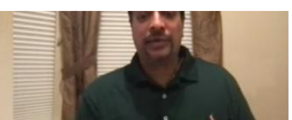
- Goal : Build a system that can deliver messages from source S to destination D

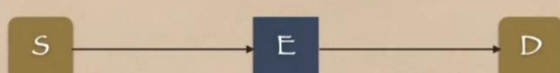


- But first, let's decouple S and D by putting messaging infrastructure between them



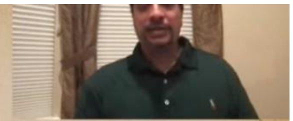
Start Simple



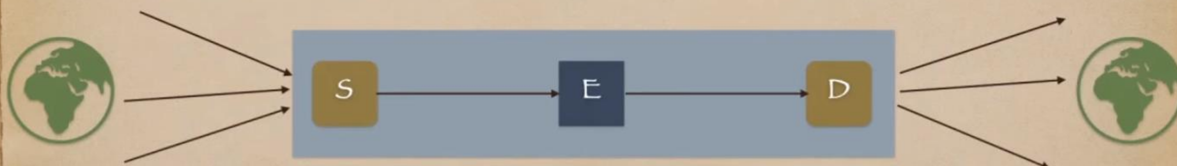


- Make a few more implementation decisions about this system
- Run our system on a cloud platform (e.g. AWS)
- Operate at low scale
  - Kafka with a single partition
  - Kafka across 3 brokers split across AZs with RF=3 (min in-sync replicas =2)
  - Run S & D on single, separate EC2 Instances

Start Simple



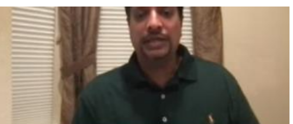
- To make things a bit more interesting, let's provide our stream as a service
  - We define our system boundary using a blue box as shown below!



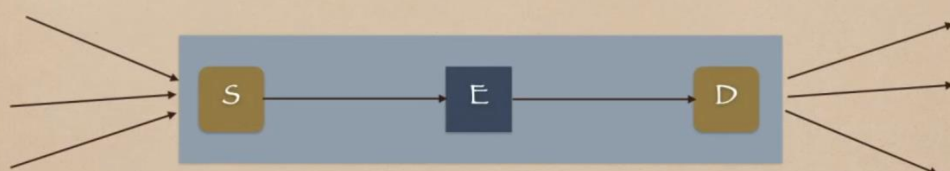
Reliability

(Is This System Reliable?)

Reliability

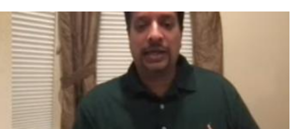


- Goal : Build a system that can deliver messages reliably from S to D

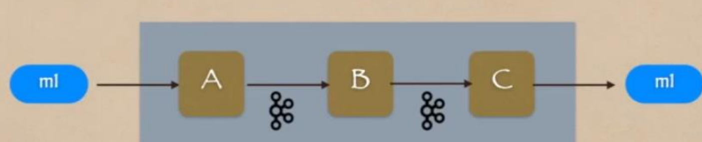


- Concrete Goal : 0 message loss
  - Once S has ACKd a message to a remote sender, D must deliver that message to a remote receiver

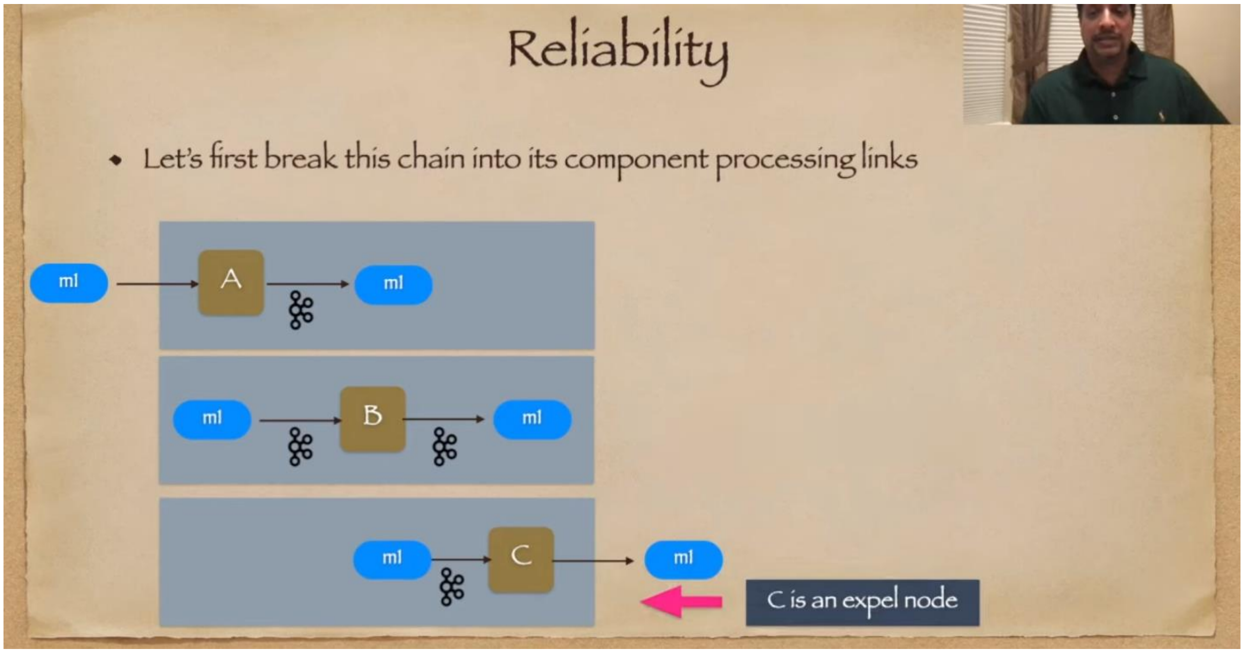
Reliability

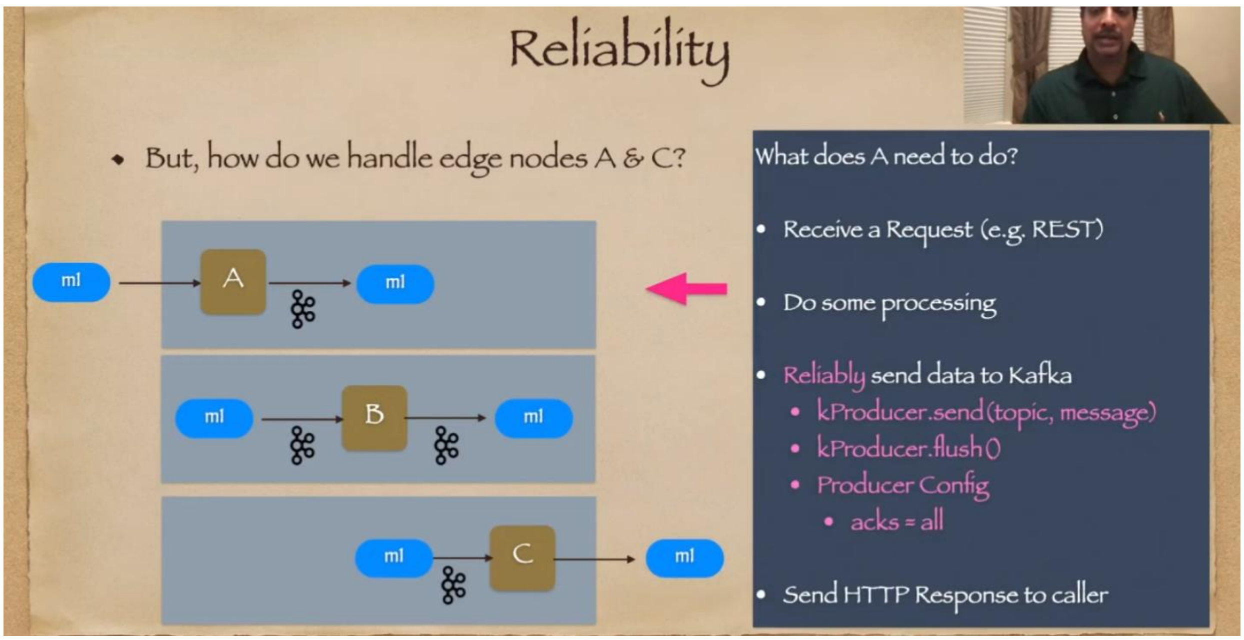


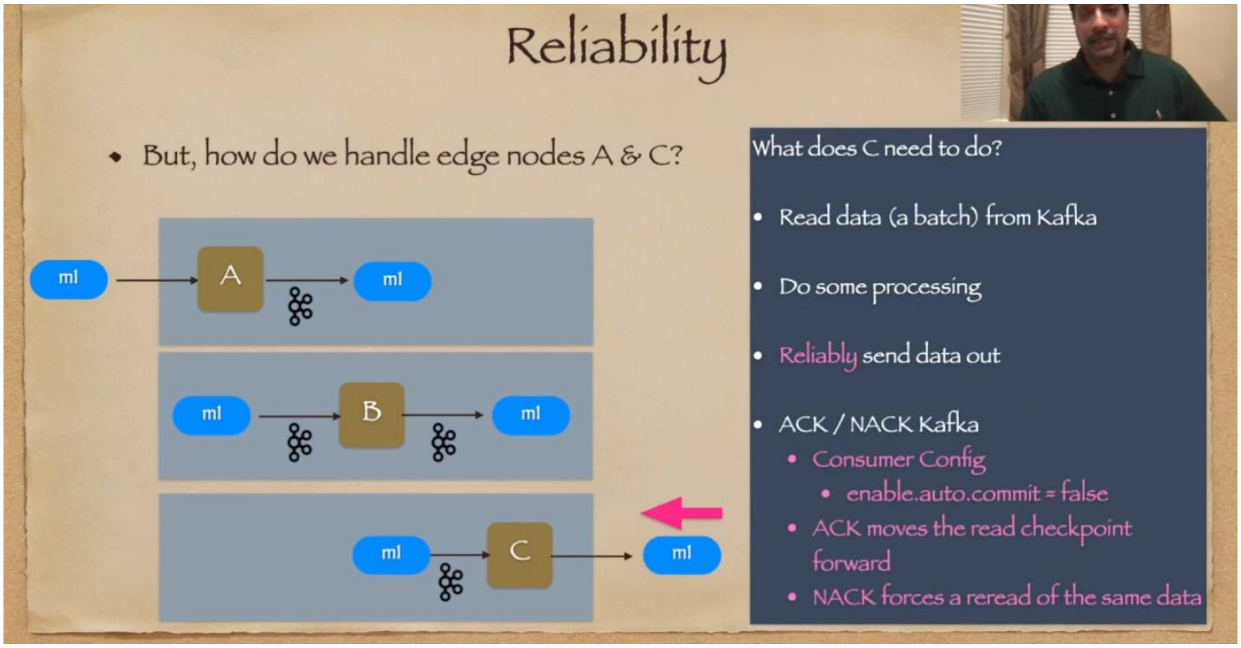
- In order to make this system reliable

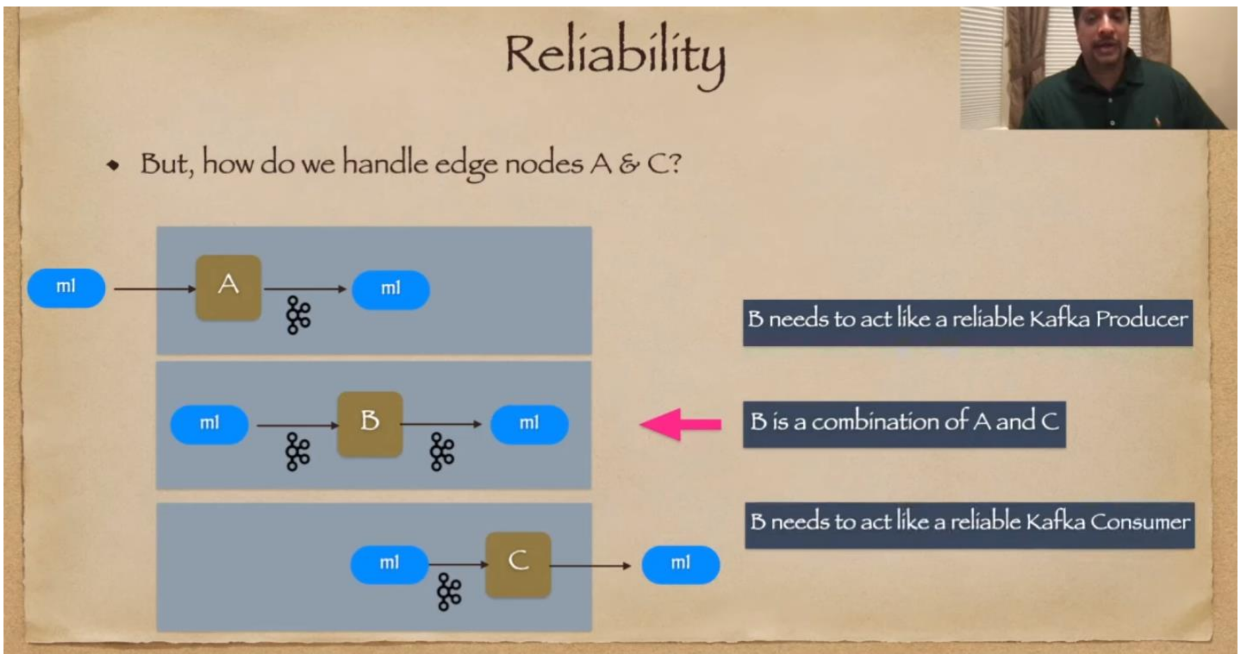


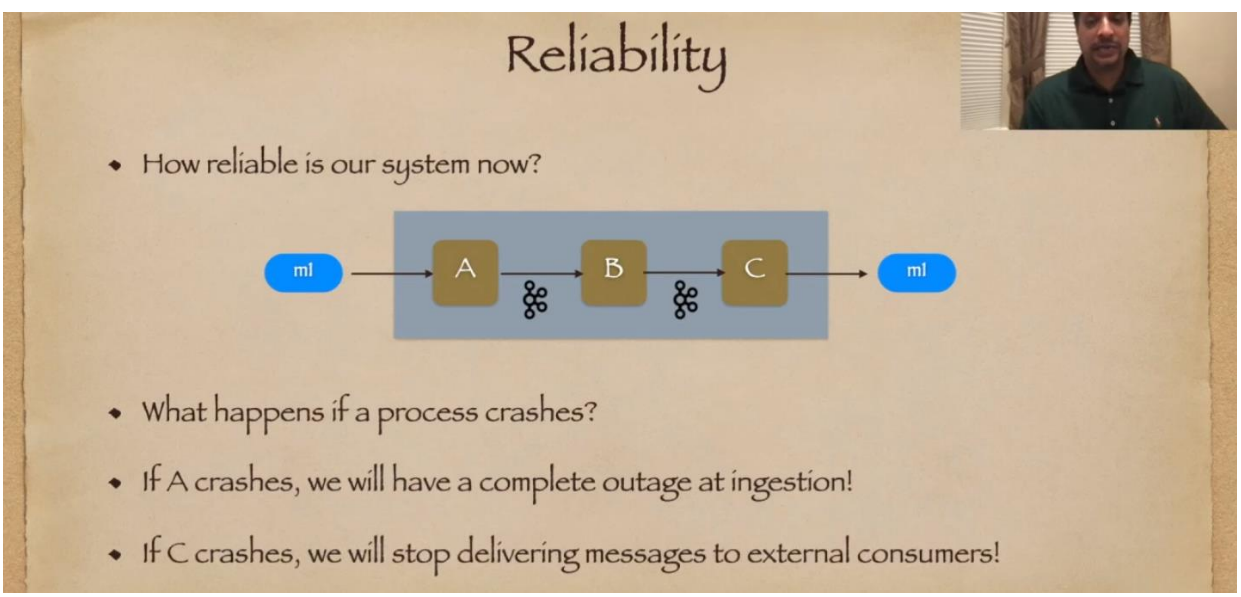
- Treat the messaging system like a chain — it's only as strong as its weakest link
- Insight : If each process/link is transactional in nature, the chain will be transactional!
- Transactionality = At least once delivery
- How do we make each link transactional?


Reliability
Let's first break this chain into its component processing links
ml
A
ml
ml
B
ml
ml
C
ml
C is an expel node

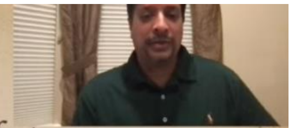

Reliability
But, how do we handle edge nodes A & C?
ml
A
ml
ml
B
ml
ml
C
ml
What does A need to do?
Receive a Request (e.g. REST)
Do some processing
Reliably send data to Kafka
kProducer.send(topic, message)
kProducer.flush()
Producer Config
acks = all
Send HTTP Response to caller


Reliability
But, how do we handle edge nodes A & C?
ml
A
ml
ml
B
ml
ml
C
ml
What does C need to do?
Read data (a batch) from Kafka
Do some processing
Reliably send data out
ACK / NACK Kafka
Consumer Config
enable.auto.commit = false
ACK moves the read checkpoint forward
NACK forces a reread of the same data

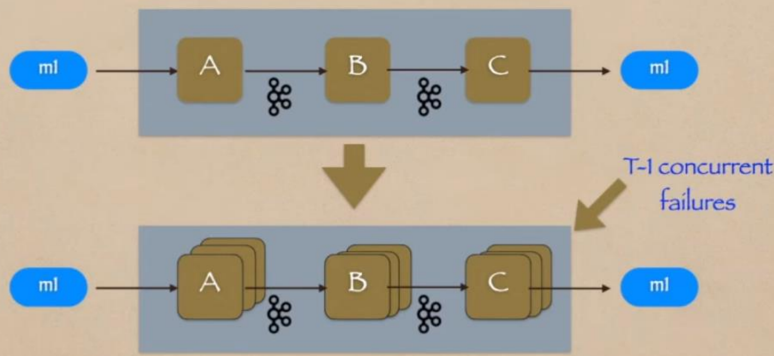

Reliability
But, how do we handle edge nodes A & C?
ml
A
ml
ml
B
ml
ml
C
ml
B needs to act like a reliable Kafka Producer
B is a combination of A and C
B needs to act like a reliable Kafka Consumer


Reliability
How reliable is our system now?
ml
A
B
C
ml
What happens if a process crashes?
If A crashes, we will have a complete outage at ingestion!
If C crashes, we will stop delivering messages to external consumers!

Reliability



Solution : Place each service in an autoscaling group of size T



- For now, we appear to have a pretty reliable data stream

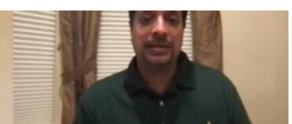
But how do we measure its reliability?

(This brings us to ...)

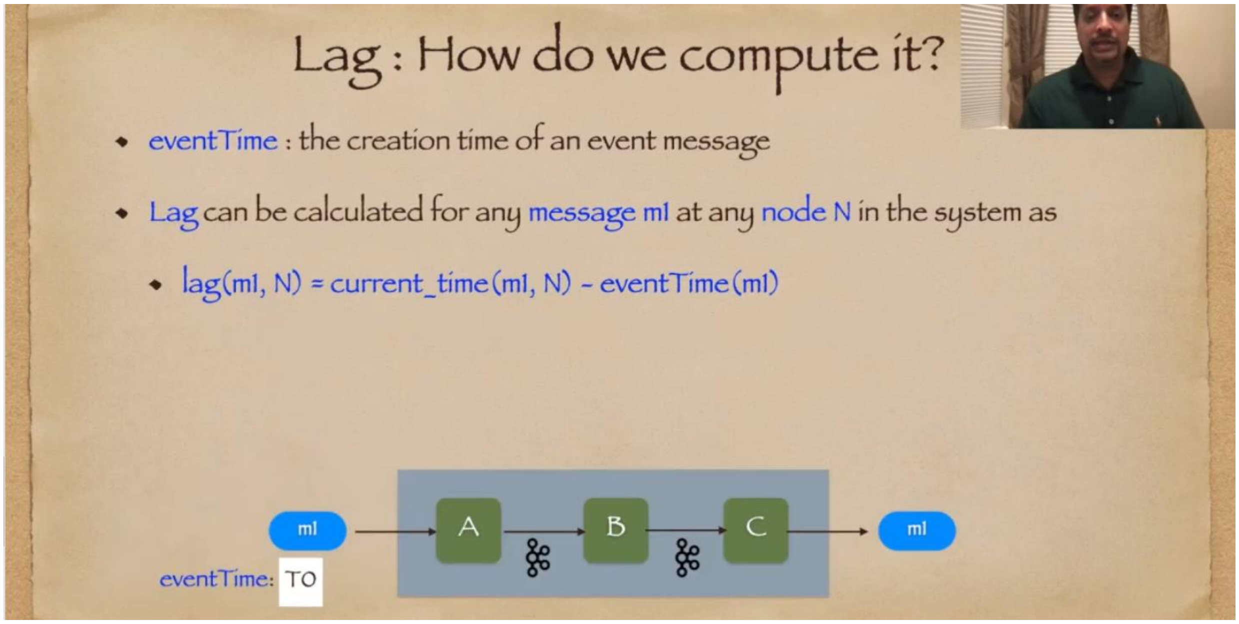
Observability

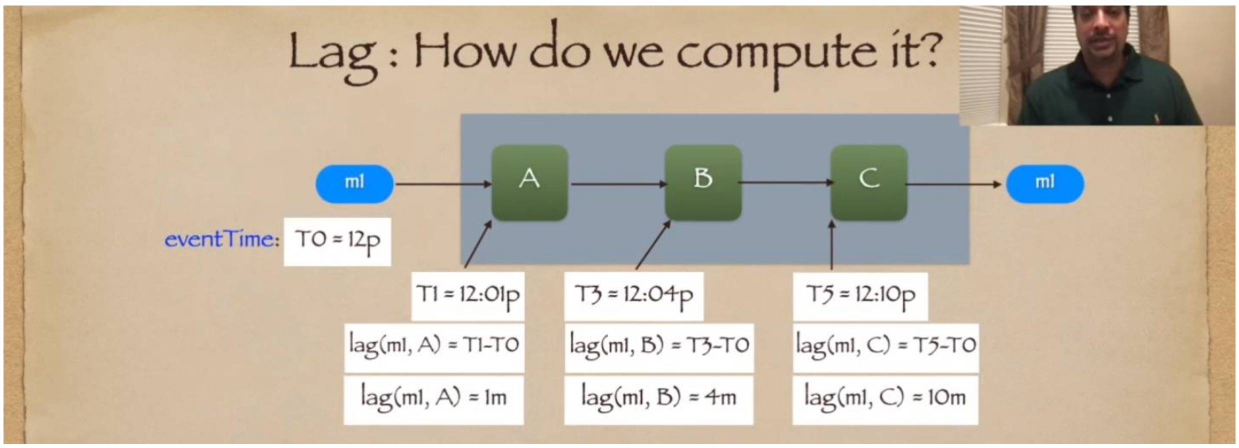
(A story about Lag & Loss Metrics)

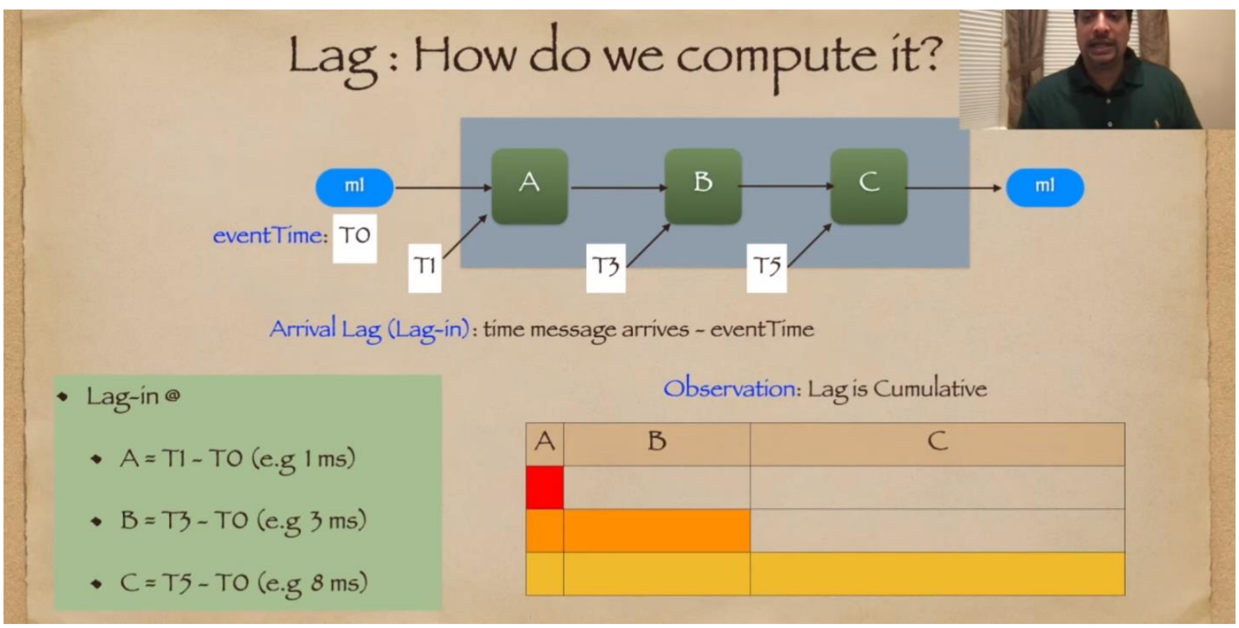
Lag : What is it?



- Lag is simply a measure of message delay in a system
- The longer a message takes to transit a system, the greater its lag
- The greater the lag, the greater the impact to the business
- Hence, our goal is to minimize lag in order to deliver insights as quickly as possible

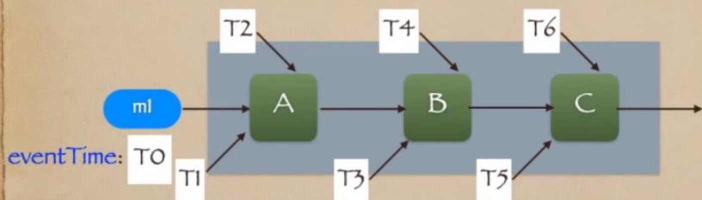

Lag : How do we compute it?
eventTime : the creation time of an event message
Lag can be calculated for any message m1 at any node N in the system as
lag(m1, N) = current_time(m1, N) - eventTime(m1)
m1
A
B
C
m1
eventTime: T0


Lag : How do we compute it?
m1
A
B
C
m1
eventTime: T0 = 12p
T1 = 12:01p
T3 = 12:04p
T5 = 12:10p
lag(m1, A) = T1-T0
lag(m1, B) = T3-T0
lag(m1, C) = T5-T0
lag(m1, A) = 1m
lag(m1, B) = 4m
lag(m1, C) = 10m


Lag : How do we compute it?
m1
A
B
C
m1
eventTime: T0
T1
T3
T5
Arrival Lag (Lag-in) : time message arrives - eventTime
Lag-in @
A = T1 - T0 (e.g 1 ms)
B = T3 - T0 (e.g 3 ms)
C = T5 - T0 (e.g 8 ms)
Observation: Lag is Cumulative
A
B
C

Lag : How do we compute it?

Departure Lag (Lag-out): time message leaves - eventTime

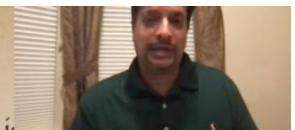


Arrival Lag (Lag-in): time message arrives - eventTime

- Lag-out @
  - A = T2 - T0 (e.g 2 ms)
  - B = T4 - T0 (e.g 4 ms)
  - C = T6 - T0 (e.g 10 ms)
- Lag-in @
  - A = T1 - T0 (e.g 1 ms)
  - B = T3 - T0 (e.g 3 ms)
  - C = T5 - T0 (e.g 8 ms)

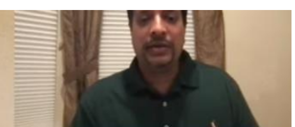
The most important metric for Lag in any streaming system is E2E Lag: the total time a message spent in the system

Lag : How do we compute it?

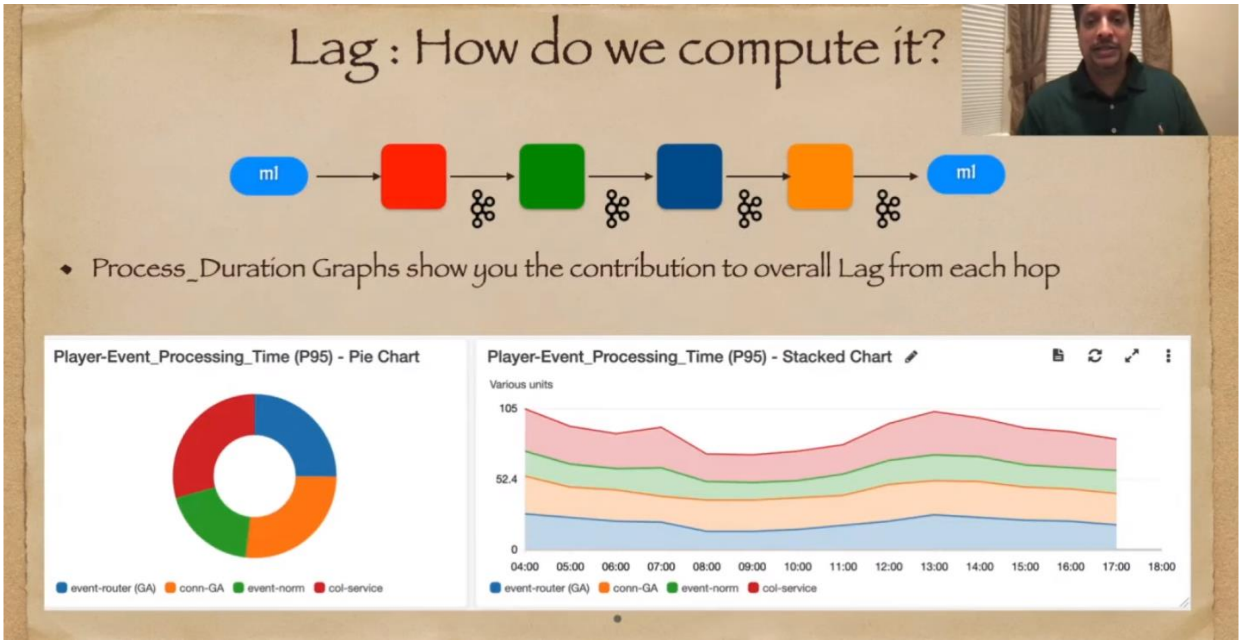


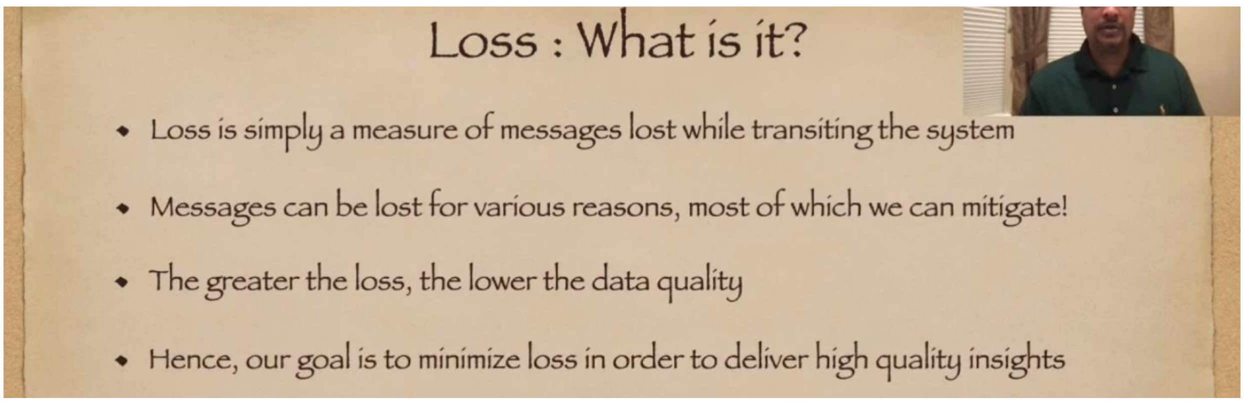
- While it is interesting to know the lag for a particular message ml, it is [illegible] since we typically deal with millions of messages
- Instead, we prefer statistics (e.g. P95) to capture population behavior

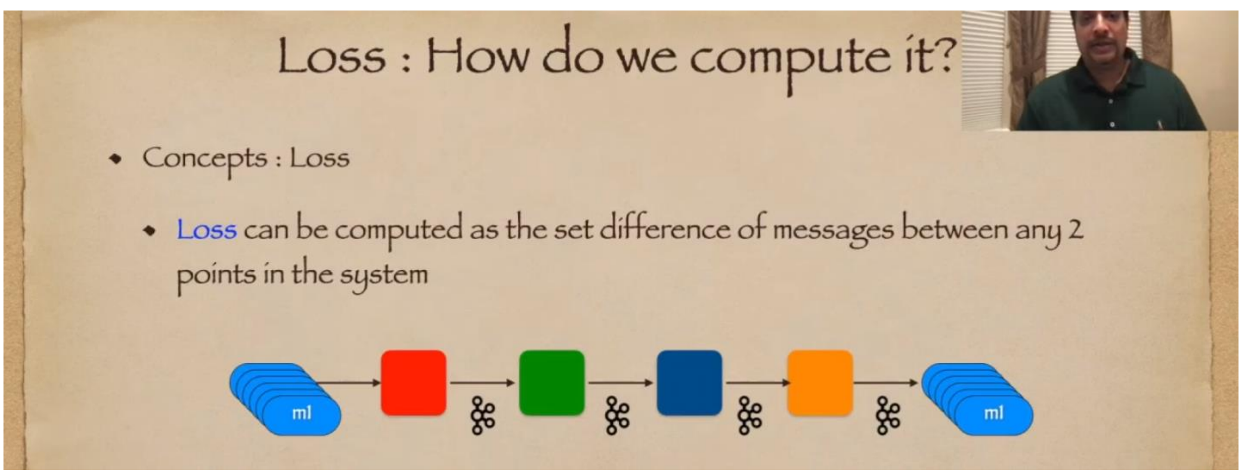
Lag : How do we compute it?



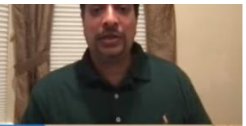
- Some useful Lag statistics are:
  - E2E Lag (p95) : 95th percentile time of messages spent in the system
  - Lag_[in|out] (N, p95): P95 Lag_in or Lag_out at any Node N
  - Process_Duration(N, p95) : Time Spent at any node in the chain!
    - Lag_out(N, p95) - Lag_in(N, p95)

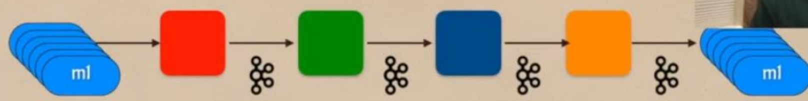

Lag : How do we compute it?
ml
ml
Process_Duration Graphs show you the contribution to overall Lag from each hop
Player-Event_Processing_Time (P95) - Pie Chart
event-router (GA)
conn-GA
event-norm
col-service
Player-Event_Processing_Time (P95) - Stacked Chart
Various units
105
52.4
0
04:00
05:00
06:00
07:00
08:00
09:00
10:00
11:00
12:00
13:00
14:00
15:00
16:00
17:00
18:00
event-router (GA)
conn-GA
event-norm
col-service


Loss : What is it?
Loss is simply a measure of messages lost while transiting the system
Messages can be lost for various reasons, most of which we can mitigate!
The greater the loss, the lower the data quality
Hence, our goal is to minimize loss in order to deliver high quality insights


Loss : How do we compute it?
Concepts : Loss
Loss can be computed as the set difference of messages between any 2 points in the system
ml
ml

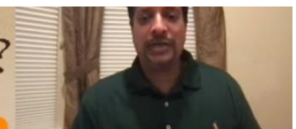
Loss : How do we compute it?

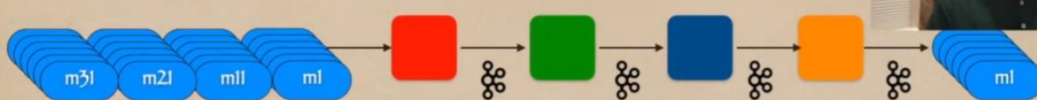




Message Id					E2E Loss	E2E Loss %
m1	1	1	1	1		
m2	1	1	1	1		
m3	1	0	0	0		
...	...	...	...	...		
m10	1	1	0	0		
Count	**10**	**9**	**7**	**5**		
Per Node Loss(N)	**0**	**1**	**2**	**2**	**5**	**50%**

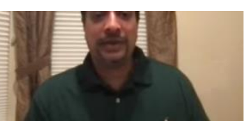
Loss : How do we compute it?

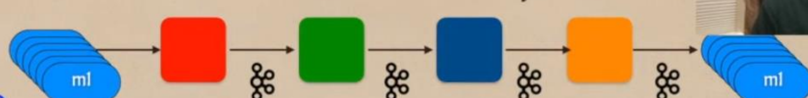




- In a streaming data system, messages never stop flowing. So, how do we know when to count?
- Solution
  - Allocate messages to 1-minute wide time buckets using message eventTime

Loss : How do we compute it?

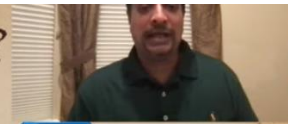


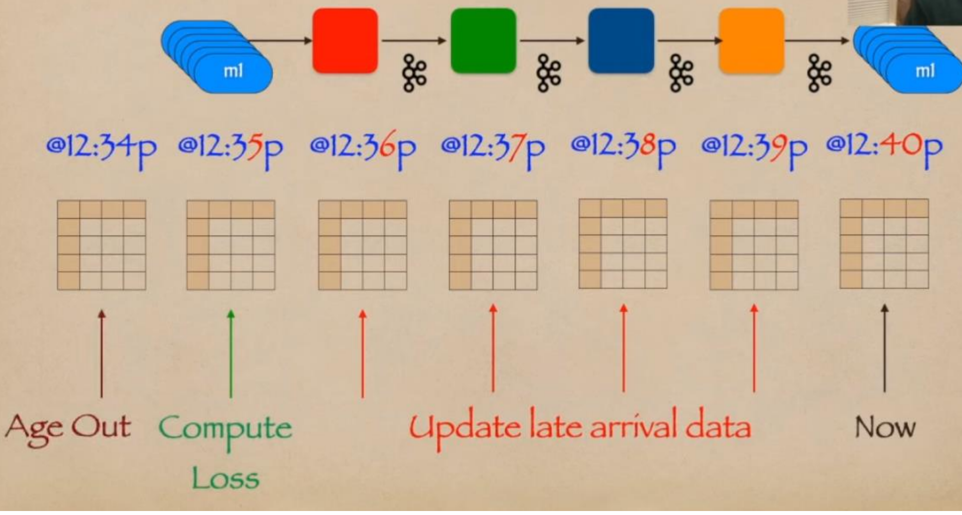


@12:34p

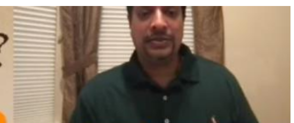
Message Id					E2E Loss	E2E Loss %
m1	1	1	1	1		
m2	1	1	1	1		
m3	1	0	0	0		
...	...	...	...	...		
m10	1	1	0	0		
Count	**10**	**9**	**7**	**5**		
Per Node Loss(N)	**0**	**1**	**2**	**2**	**5**	**50%**

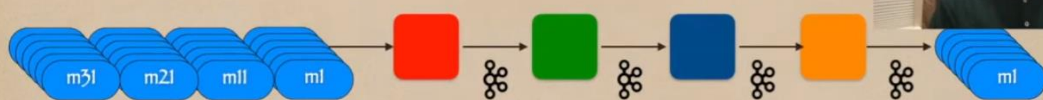
Loss : How do we compute it?





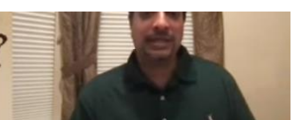
Loss : How do we compute it?





- In a streaming data system, messages never stop flowing. So, how do we know when to count?
- Solution
  - Allocate messages to 1-minute wide time buckets using message eventTime
  - Wait a few minutes for messages to transit, then compute loss (e.g. 12:35p loss table)
  - Raise alarms if loss occurs over a configured threshold (e.g. > 1%)

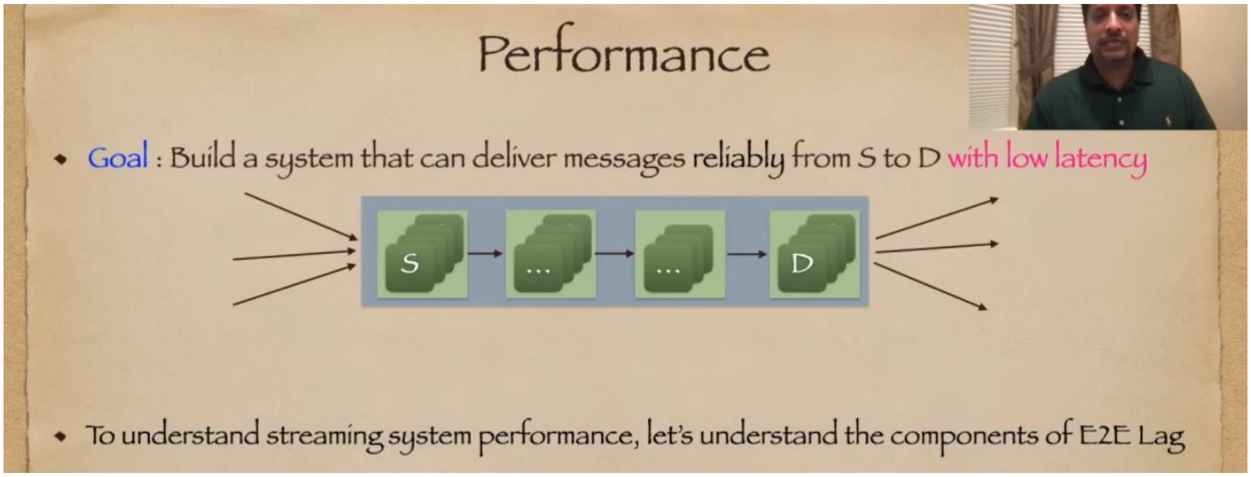
Loss : How do we compute it?

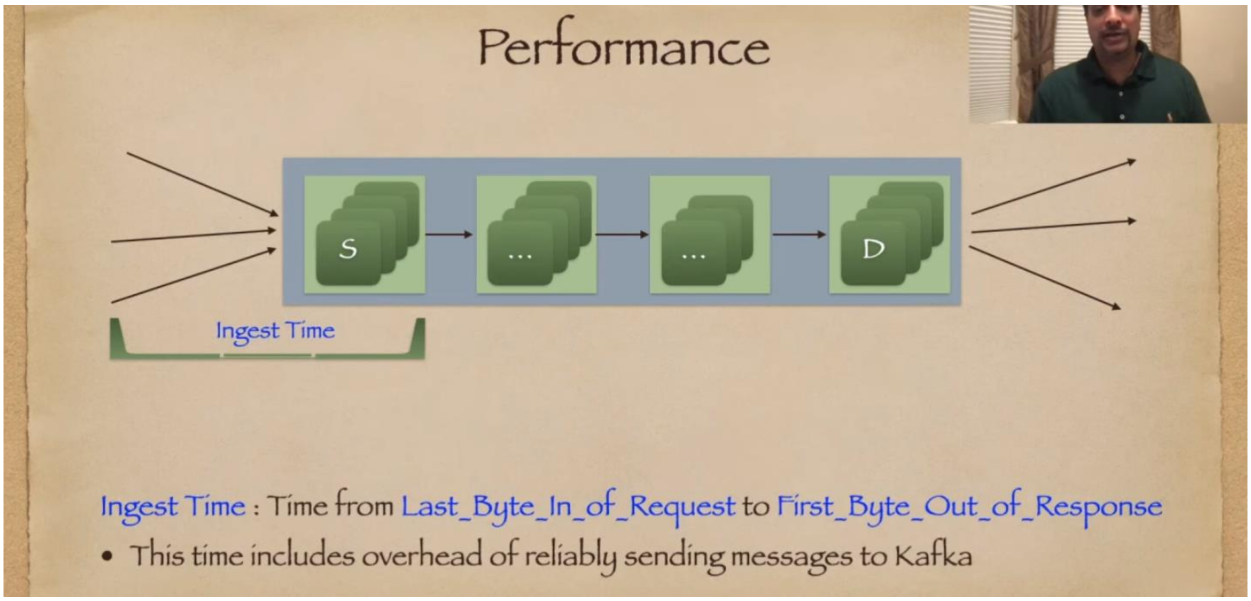


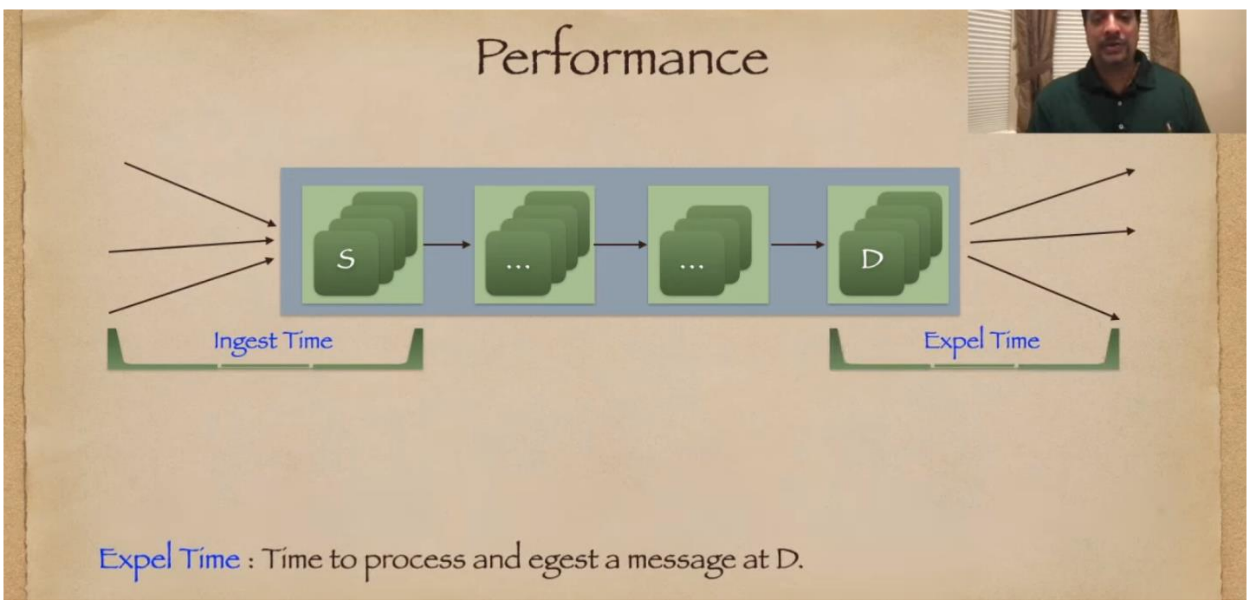
- We now have a way to measure the reliability (via Loss metrics) and latency (via Lag metrics) of our system.
- But wait…

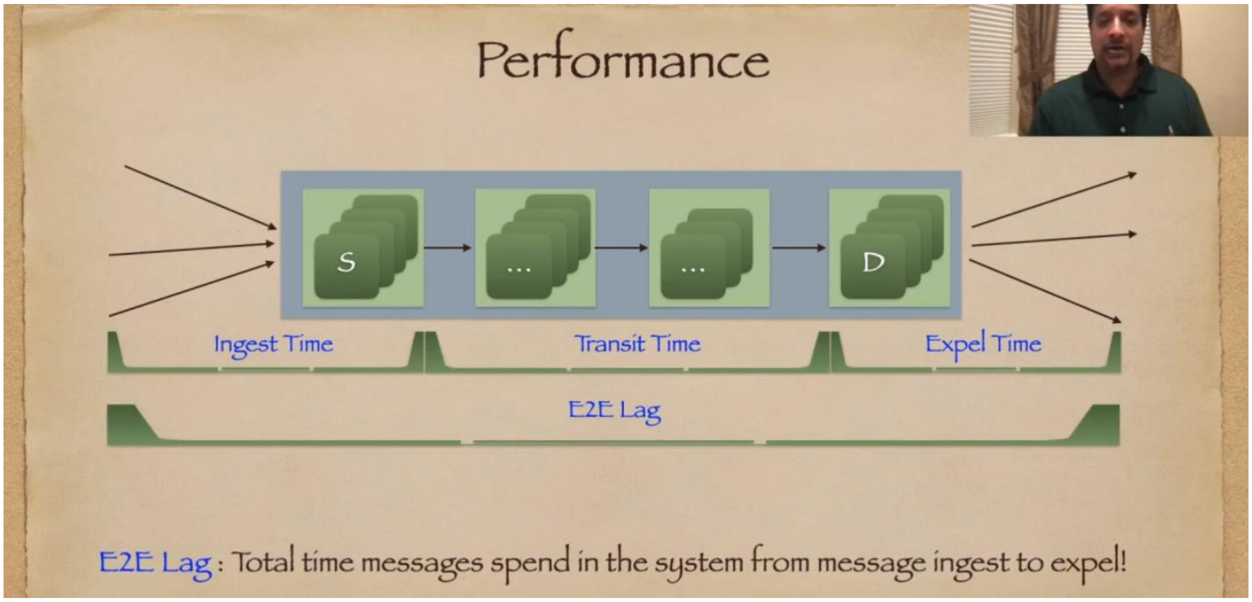
Performance

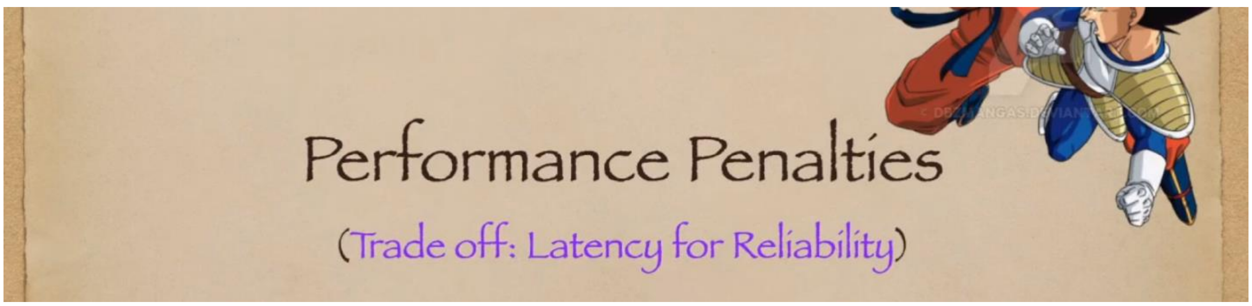
(have we tuned our system for performance yet??)

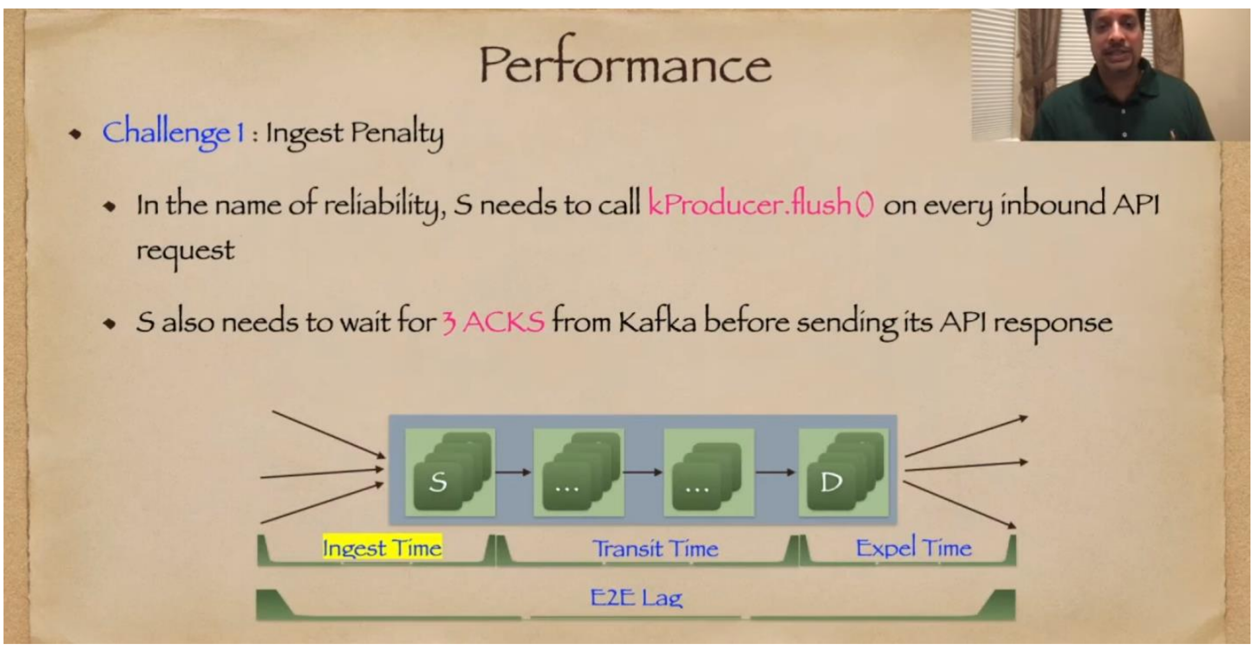

Performance
• Goal : Build a system that can deliver messages reliably from S to D with low latency
S
...
...
D
• To understand streaming system performance, let's understand the components of E2E Lag


Performance
S
...
...
D
Ingest Time
Ingest Time : Time from Last_Byte_In_of_Request to First_Byte_Out_of_Response
• This time includes overhead of reliably sending messages to Kafka


Performance
S
...
...
D
Ingest Time
Expel Time
Expel Time : Time to process and egest a message at D.

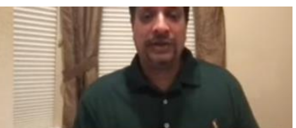

Performance
S
...
...
D
Ingest Time
Transit Time
Expel Time
E2E Lag
E2E Lag : Total time messages spend in the system from message ingest to expel!


Performance Penalties
(Trade off: Latency for Reliability)


Performance
Challenge 1 : Ingest Penalty
In the name of reliability, S needs to call kProducer.flush() on every inbound API request
S also needs to wait for 3 ACKS from Kafka before sending its API response
S
...
...
D
Ingest Time
Transit Time
Expel Time
E2E Lag

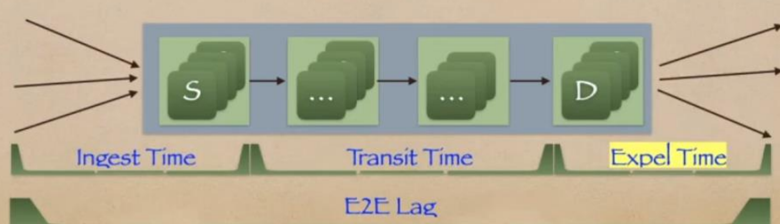
Performance

- Challenge 1 : Ingest Penalty
  - Approach : Amortization
    - Support Batch APIs (i.e. multiple messages per web request) to amortize the ingest penalty

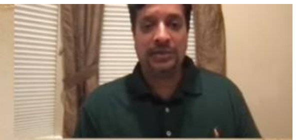


Performance

- Challenge 2 : Expel Penalty
  - Observations
    - Kafka is very fast — many orders of magnitude faster than HTTP RTTs
    - The majority of the expel time is the HTTP RTT

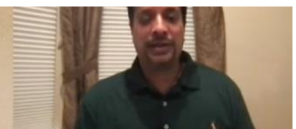


Performance



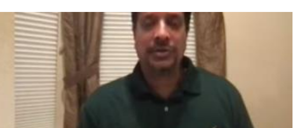
- Challenge 2 : Expel Penalty
  - Approach : Amortization
    - In each D node, add batch + parallelism

Performance



- Challenge 3 : Retry Penalty (@ D)
  - Concepts
    - In order to run a zero-loss pipeline, we need to retry messages @ D that will succeed given enough attempts
      - We call these Recoverable Failures
    - In contrast, we should never retry a message that has 0 chance of success!
      - We call these Non-Recoverable Failures
        - E.g. Any 4xx HTTP response code, except for 429 (Too Many Requests)

Performance



- Challenge 3 : Retry Penalty
  - Approach
    - We pay a latency penalty on retry, so we need to smart about
      - What we retry — Don't retry any non-recoverable failures
      - How we retry — One Idea : Tiered Retries

Performance - Tiered Retries

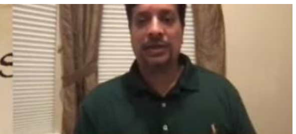
Local Retries

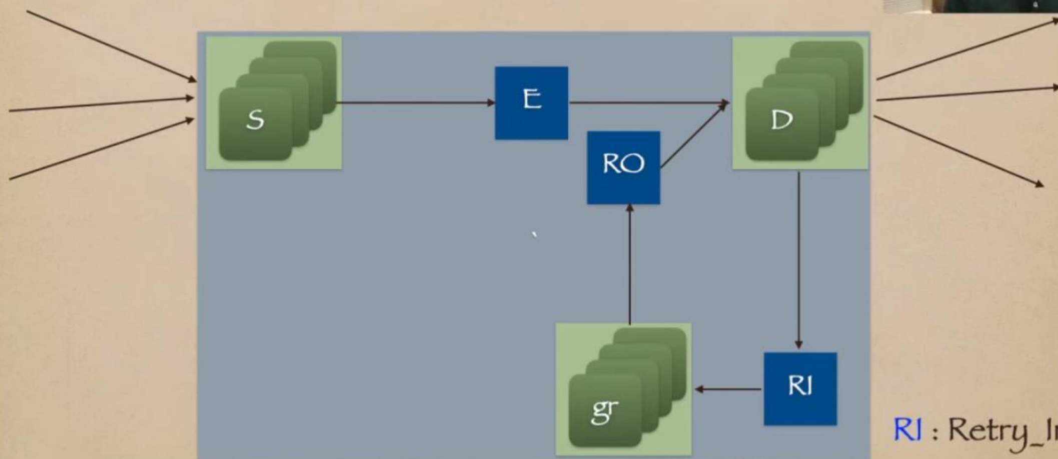
- Try to send message a configurable number of times @ D
- If we exhaust local retries, D transfers the message to a Global Retrier

Global Retries

- The Global Retrier than retries the message over a longer span of time

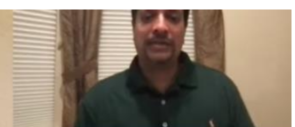
Performance - 2 Tiered Retries



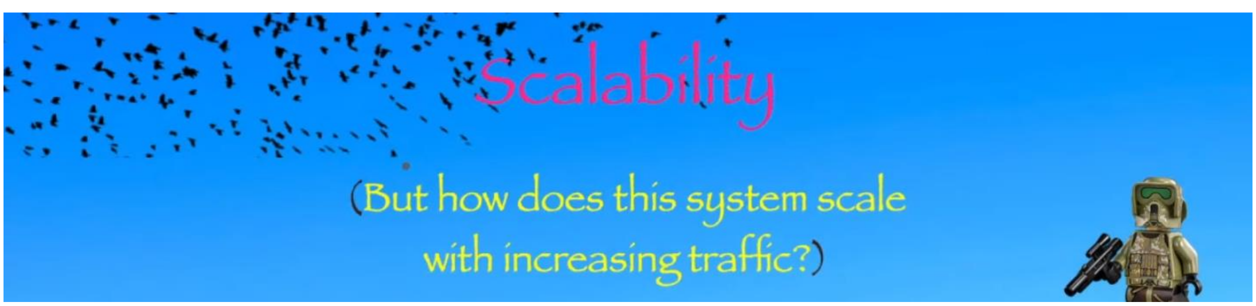


RI : Retry_In
RO : Retry_Out

Performance



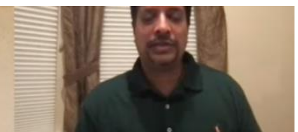
- At this point, we have a system that works well at low scale



Scalability

- First, Let's dispel a myth!
  - There is no such thing as a system that can handle infinite scale
  - Each system is traffic-rated
  - The traffic rating comes from running load tests
  - We only achieve higher scale by iteratively running load tests & removing bottlenecks

Scalability - Autoscaling



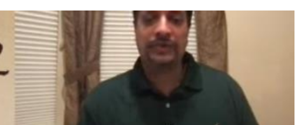
Autoscaling Goals (for data streams):

- Goal 1: Automatically scale out to maintain low latency (e.g. E2E Lag)
- Goal 2: Automatically scale in to minimize cost

Autoscaling Considerations

What can autoscale? What can't autoscale?

Scalability - Autoscaling EC2



- Pick a metric that
  - Preserves low latency
  - Goes up as traffic increases
  - Goes down as the microservice scales out

E.g.

- Average CPU

What to be wary of

- Any locks/code synchronization & IO Waits
  - Otherwise … As traffic increases, CPU will plateau, auto-scale-out will stop, and latency (i.e. E2E Lag) will increase

Conclusion

- We now have a system with the Non-functional Requirements (NFRs) that we desire!
- While we've covered many key elements, a few areas will be covered in future talks (e.g. Isolation, Containerization, Caching).
- These will be covered in upcoming blogs! Follow for updates on (@r39132)