Building & Operating High-Fidelity Data Streams

Why Do Streams Matter?

Why Do Streams Matter?

- In our world today, machine intelligence & personalization drive engaging experiences online

Why Do Streams Matter?

- In our world today, machine intelligence & personalization drive engaging experiences online

Google

Why Do Streams Matter?

- In our world today, machine intelligence & personalization drive engaging experiences online

Why Do Streams Matter?

- In our world today, machine intelligence & personalization drive engaging experiences online

- Disparate data is constantly being connected to drive predictions that keep us engaged!

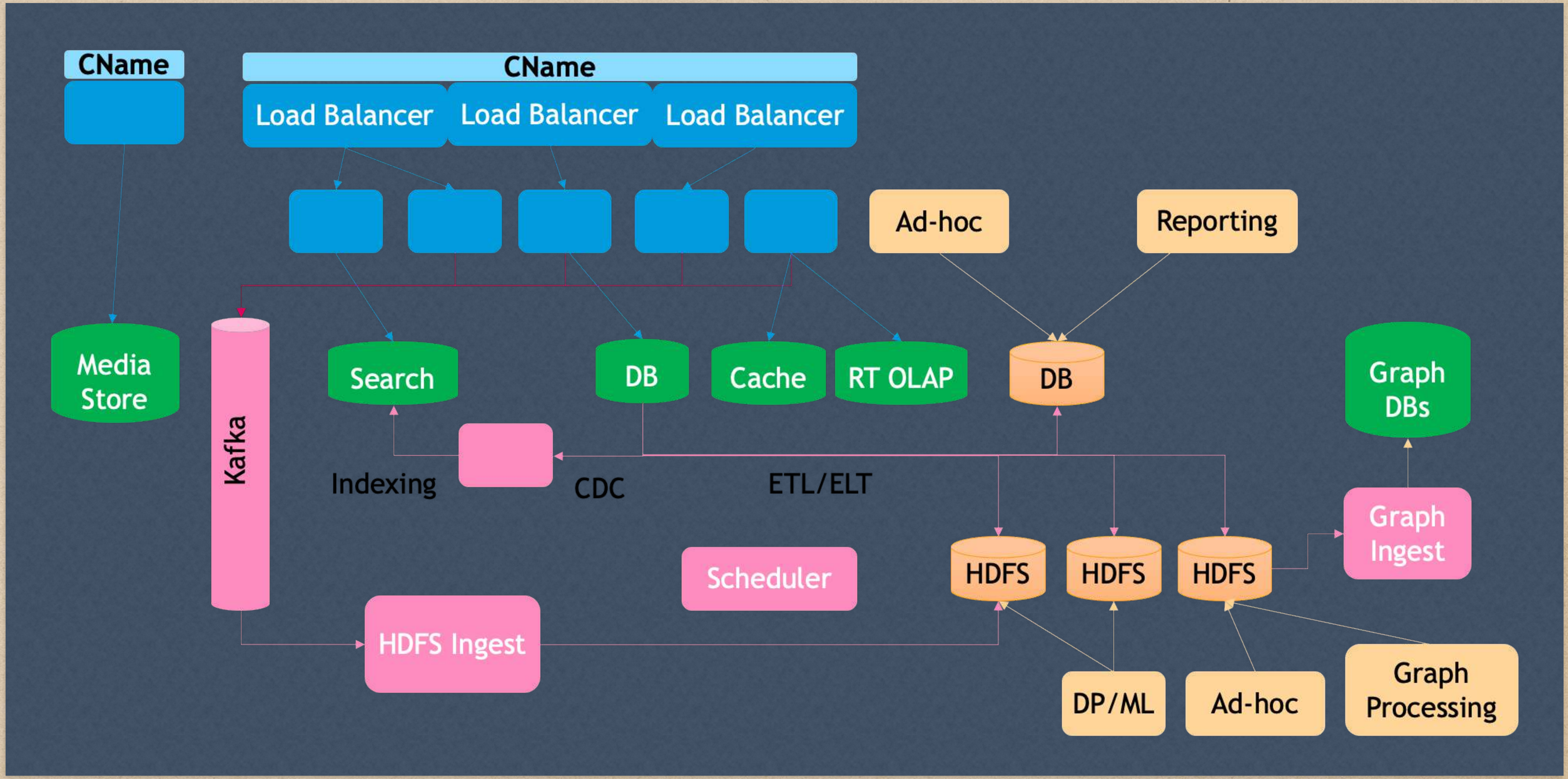
Why Do Streams Matter?

- While it may seem that some magical SQL join is powering these connections….

Why Do Streams Matter?

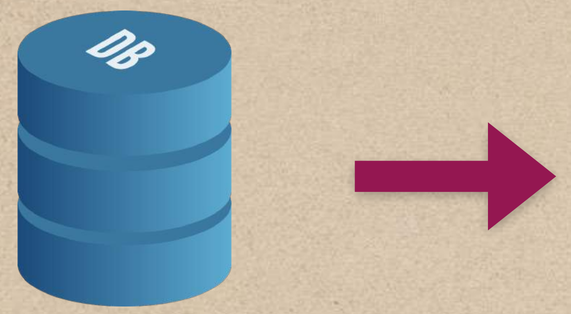
- While it may seem that some magical SQL join is powering these connections….
- The reality is that data growth has made it impractical to store all of this data in a single DB

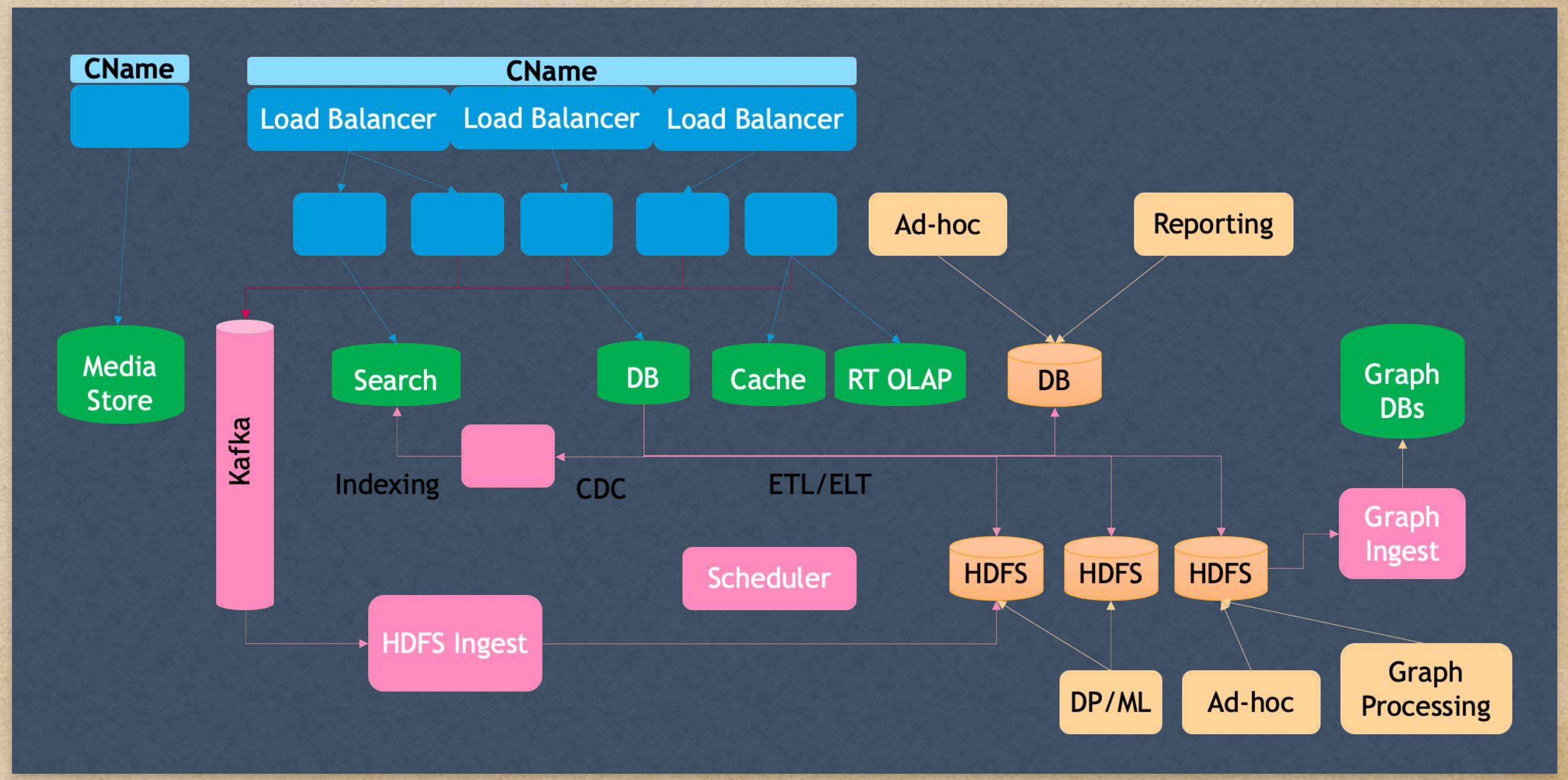
Why Do Streams Matter?



Why Do Streams Matter?

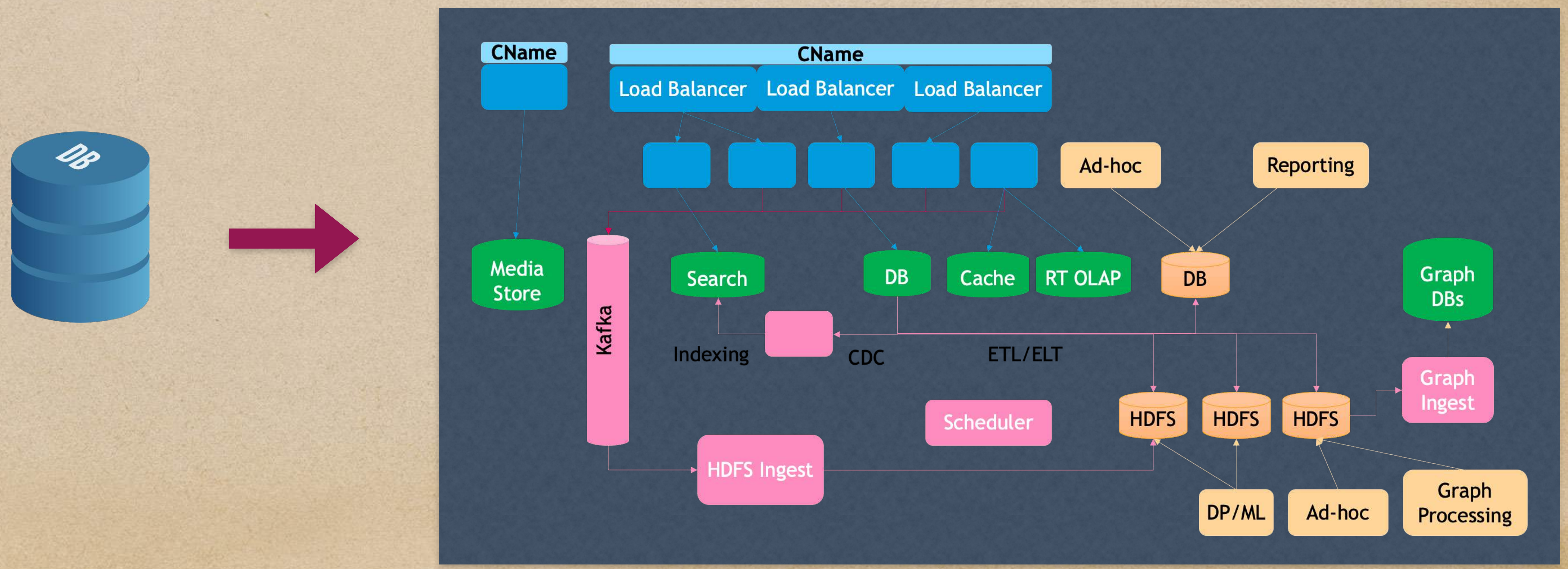
- How do companies manage the complexity below?





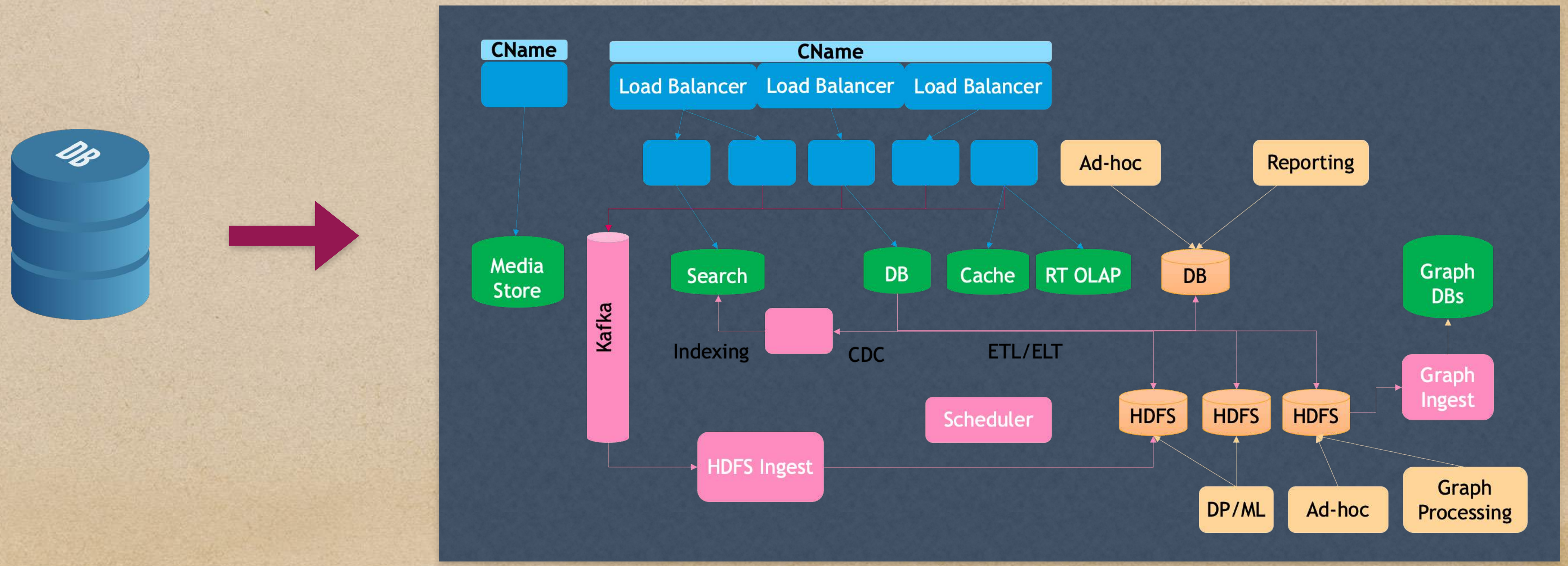
Why Do Streams Matter?

- How do companies manage the complexity below?
- A key piece to the puzzle is data movement, which usually comes in 2 forms:



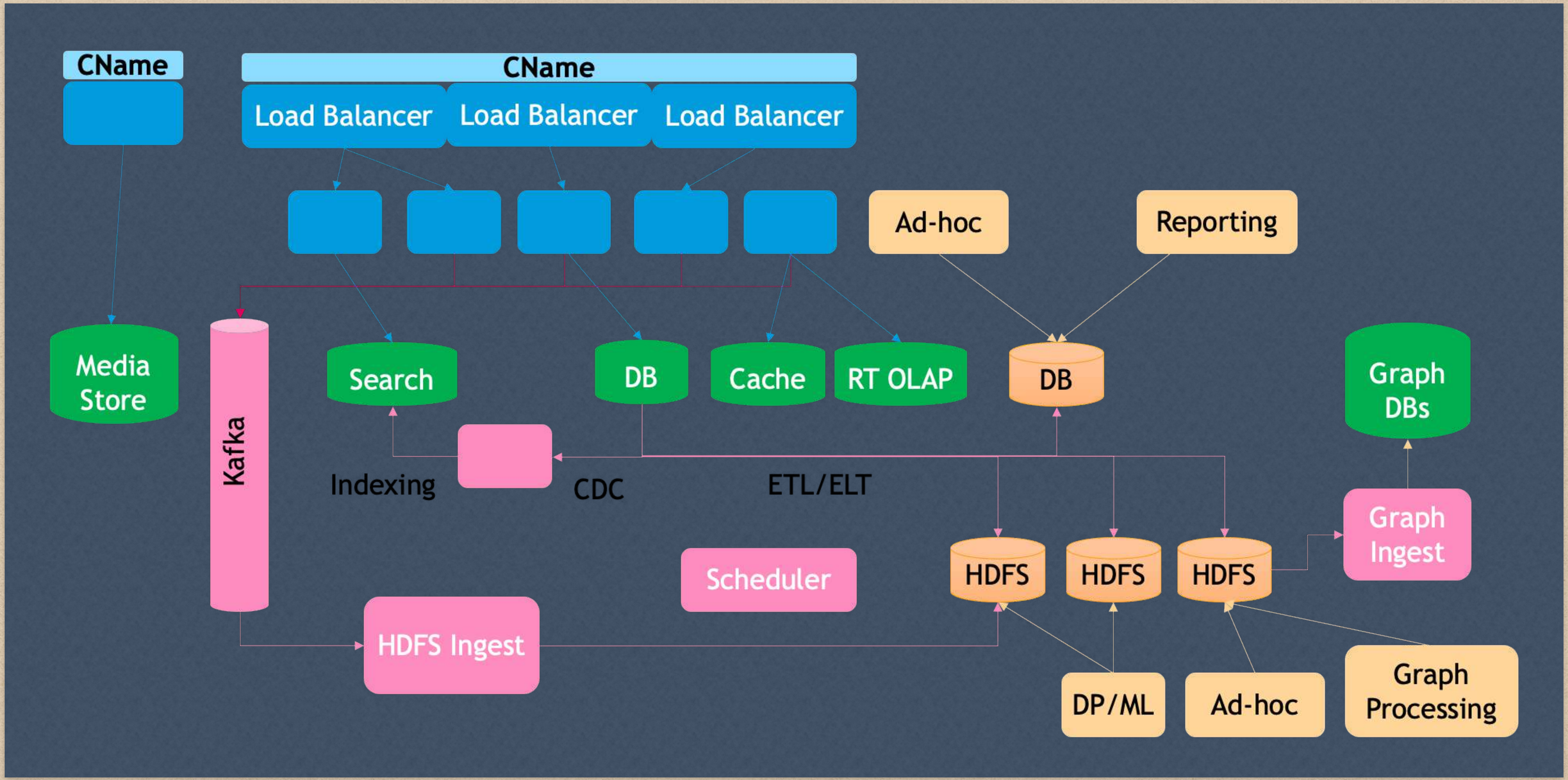
Why Do Streams Matter?

- 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



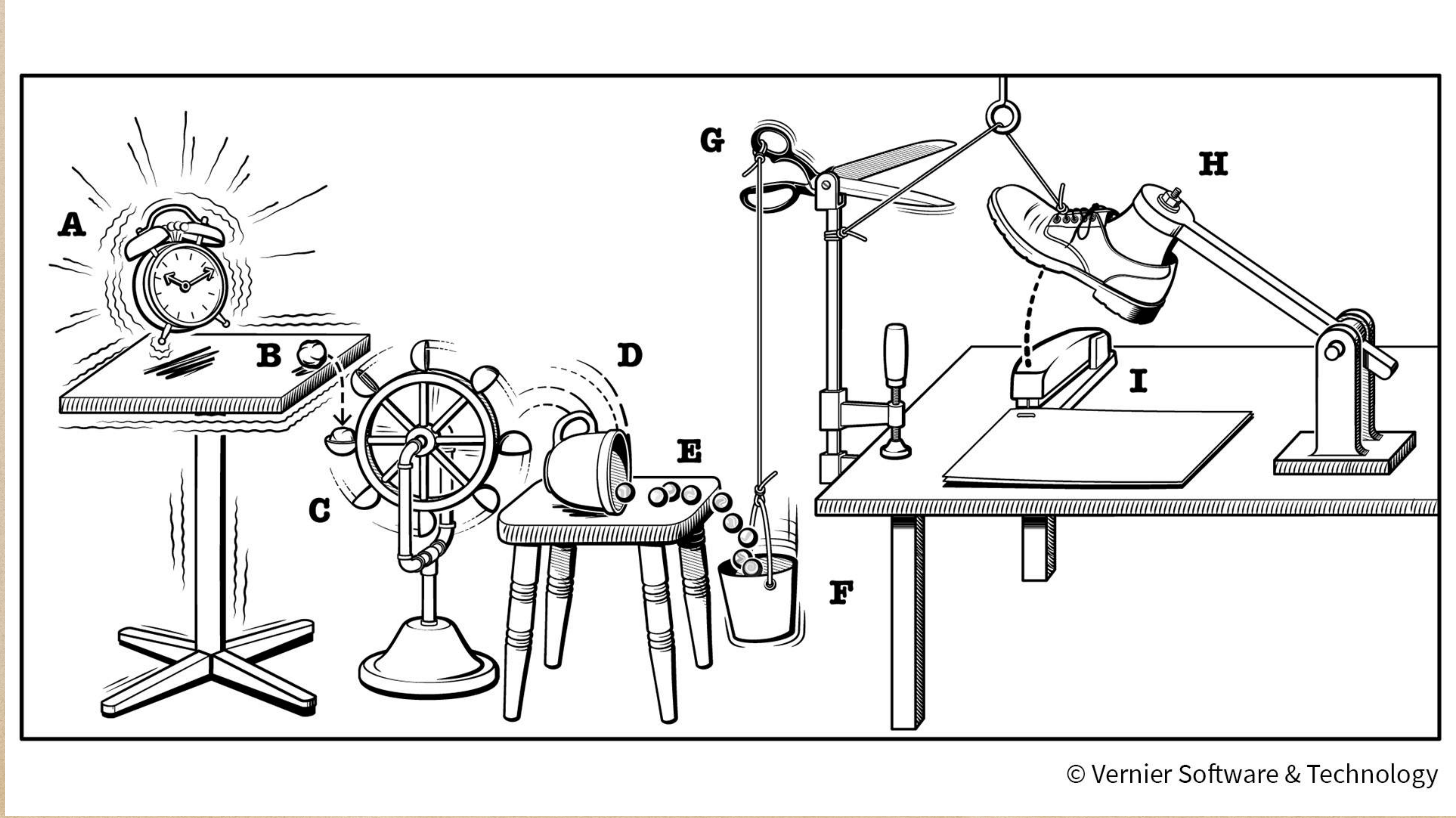
Why Are Streams Hard?

Why Are Streams Hard?



Why Are Streams Hard?

- The answer lies in the image below : complexity, lots of moving parts



Why Are Streams Hard?

- In streaming architectures, any gaps in non-functional requirements can be unforgiving

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

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

Why Are Streams Hard?

- In streaming architectures, implementation 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!

Why Are Streams Hard?

- Data Infrastructure is an iceberg
- Your customers may only see 10% of your effort — those that manifest in features
- The remaining 90% of your work goes unnoticed because it relates to keeping the lights on

Why Are Streams Hard?

- Data Infrastructure is an iceberg
- Your customers may only see 10% of your effort — those that manifest in features
- The remaining 90% of your work goes unnoticed because it relates to keeping the lights on
- In this talk, we will build high-fidelity streams-as-a-service from the ground up!


Start Simple

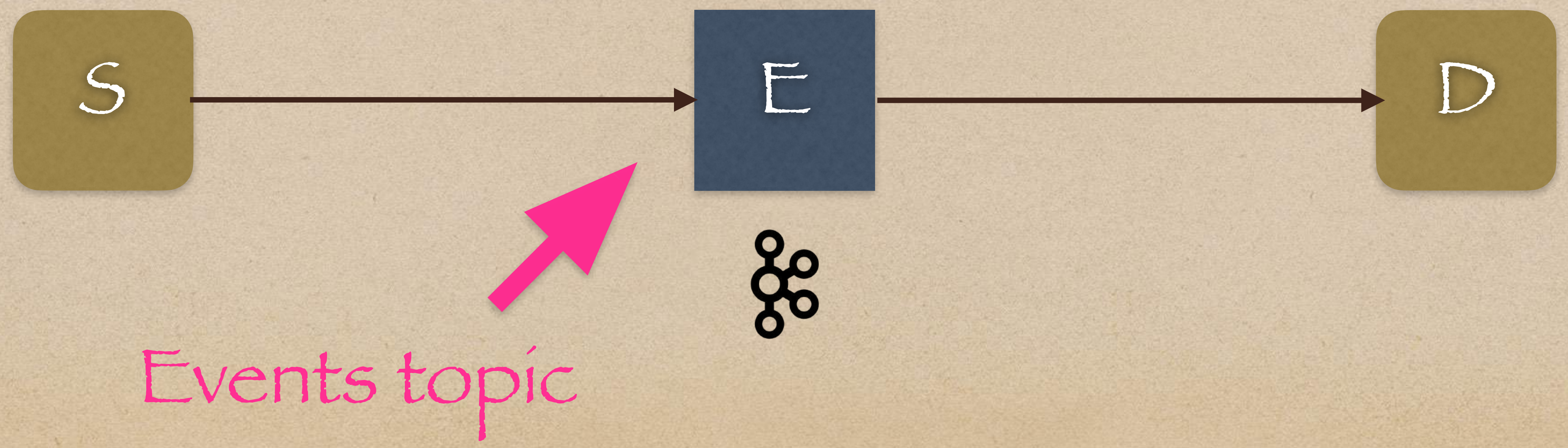
Start Simple

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

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

Start Simple

- Make a few more implementation decisions about this system
- Run our system on a cloud platform (e.g. AWS)

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

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

Start Simple

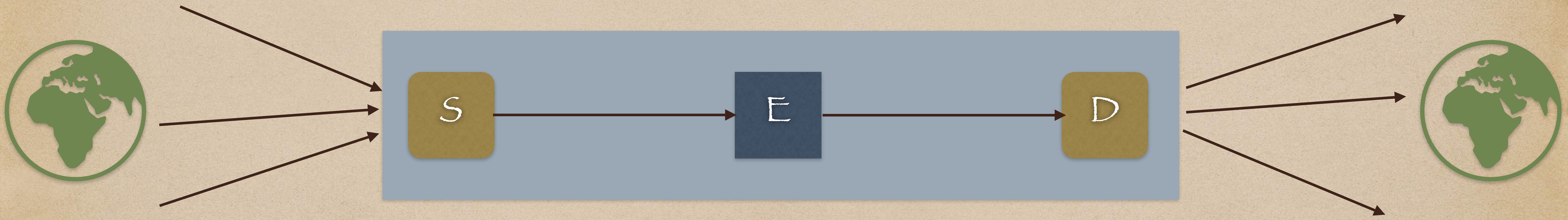
- Make a few more implementation decisions about this system
- Run our system on a cloud platform (e.g. AWS)
- Operate at low scale
    - Kafka with a single partition
    - Kafka across 3 brokers split across AZs with RF=3 (min in-sync replicas =2)

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!

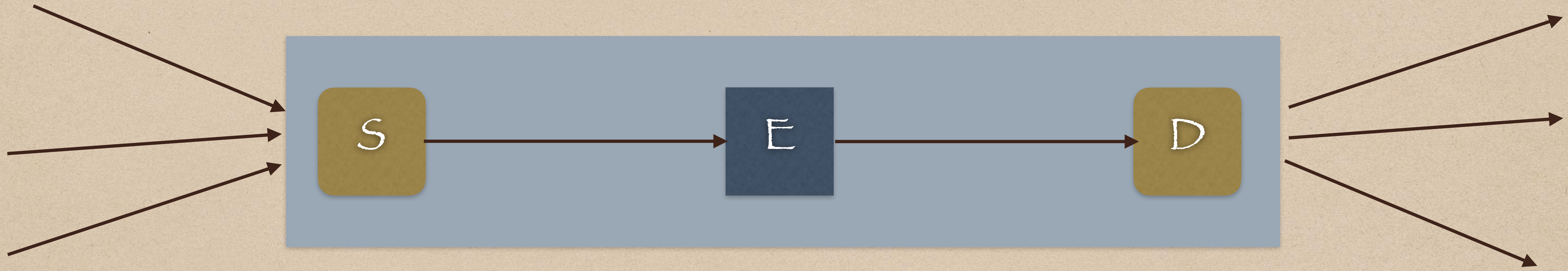


Reliability

(Is This System Reliable?)

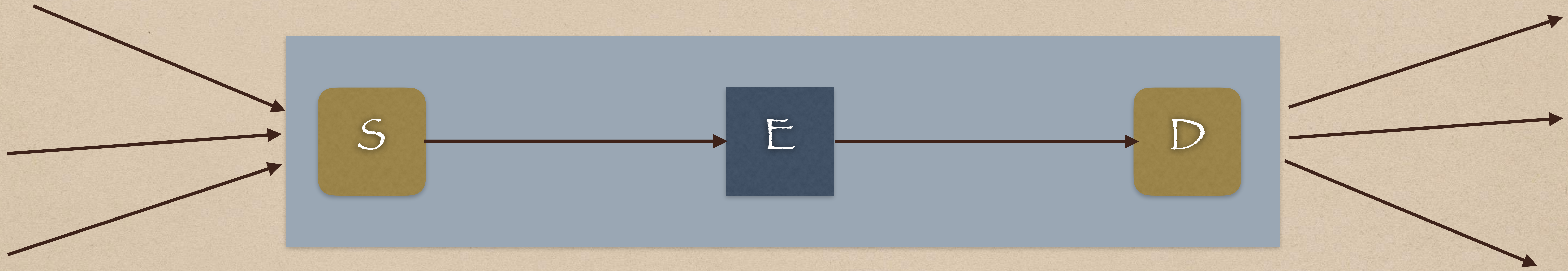
Reliability

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



Reliability

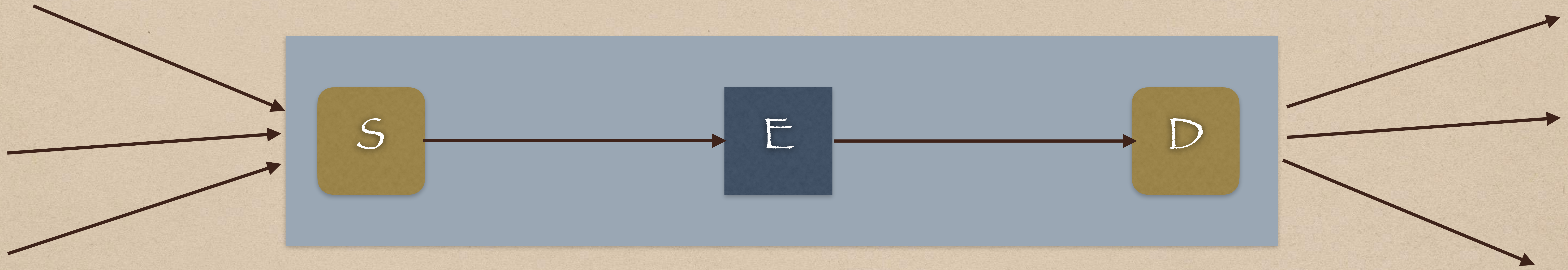
- Goal : Build a system that can deliver messages reliably from S to D



- Concrete Goal : 0 message loss

Reliability

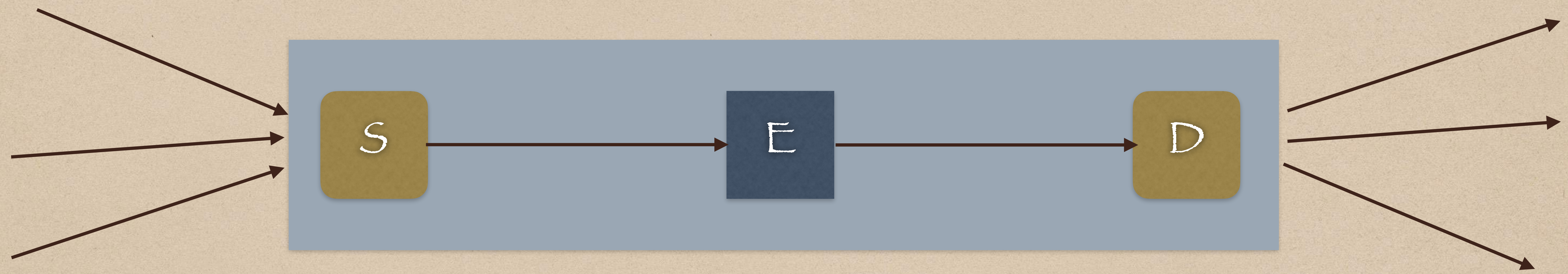
- Goal : Build a system that can deliver messages reliably from S to D



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

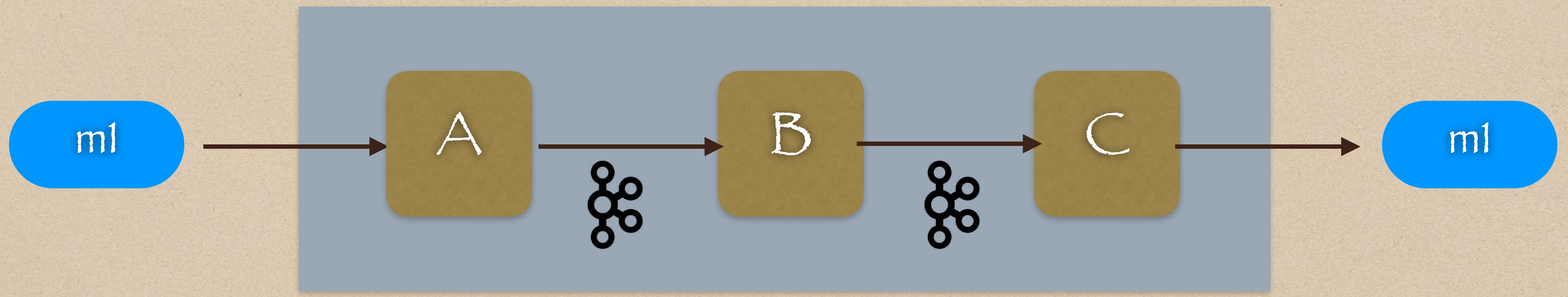
Reliability

- How do we build reliability into our system?



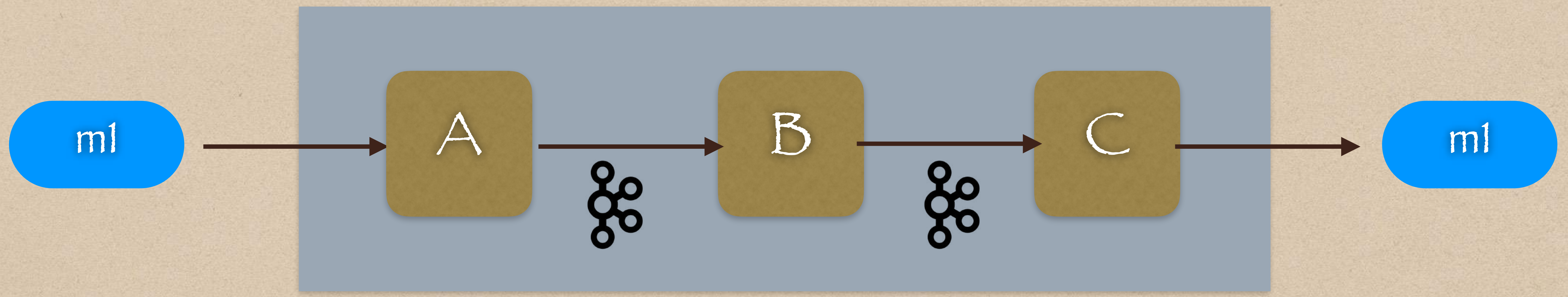
Reliability

- Let's first generalize our system!



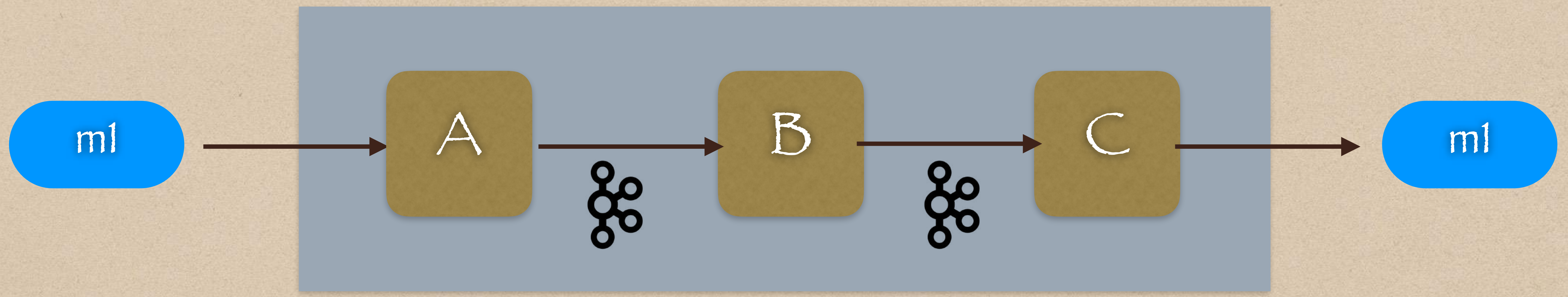
Reliability

- In order to make this system reliable



Reliability

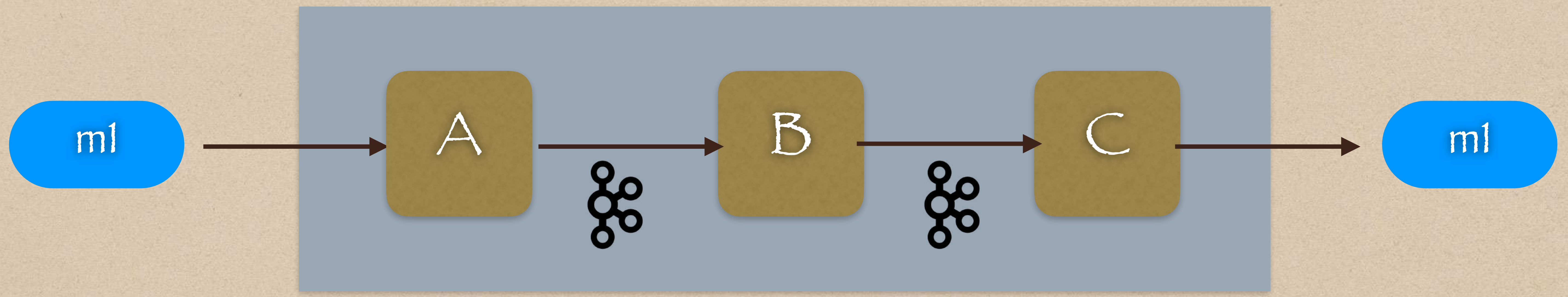
- In order to make this system reliable



- Treat the messaging system like a chain — it's only as strong as its weakest link

Reliability

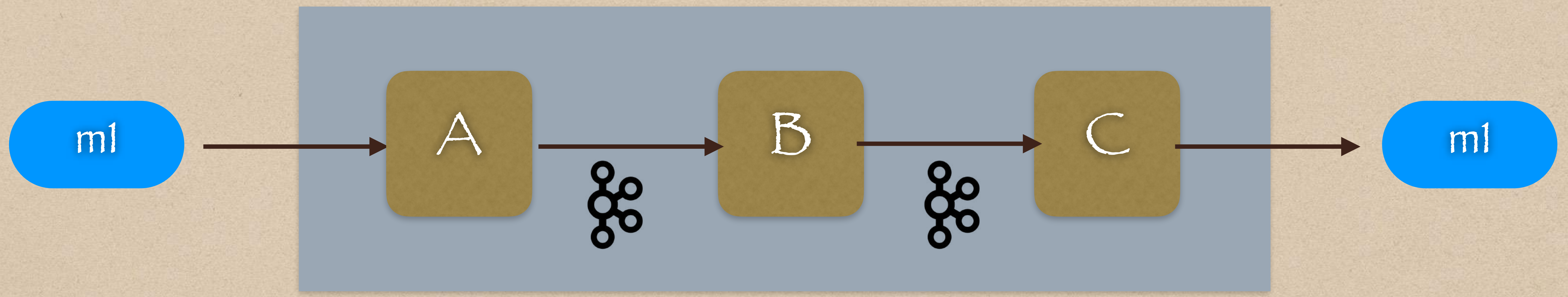
- In order to make this system reliable



- 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!

Reliability

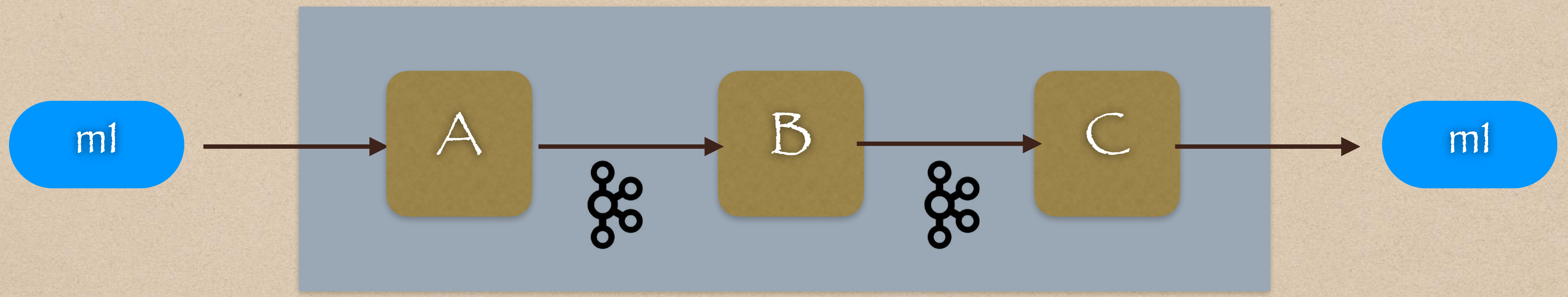
- In order to make this system reliable



- 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

Reliability

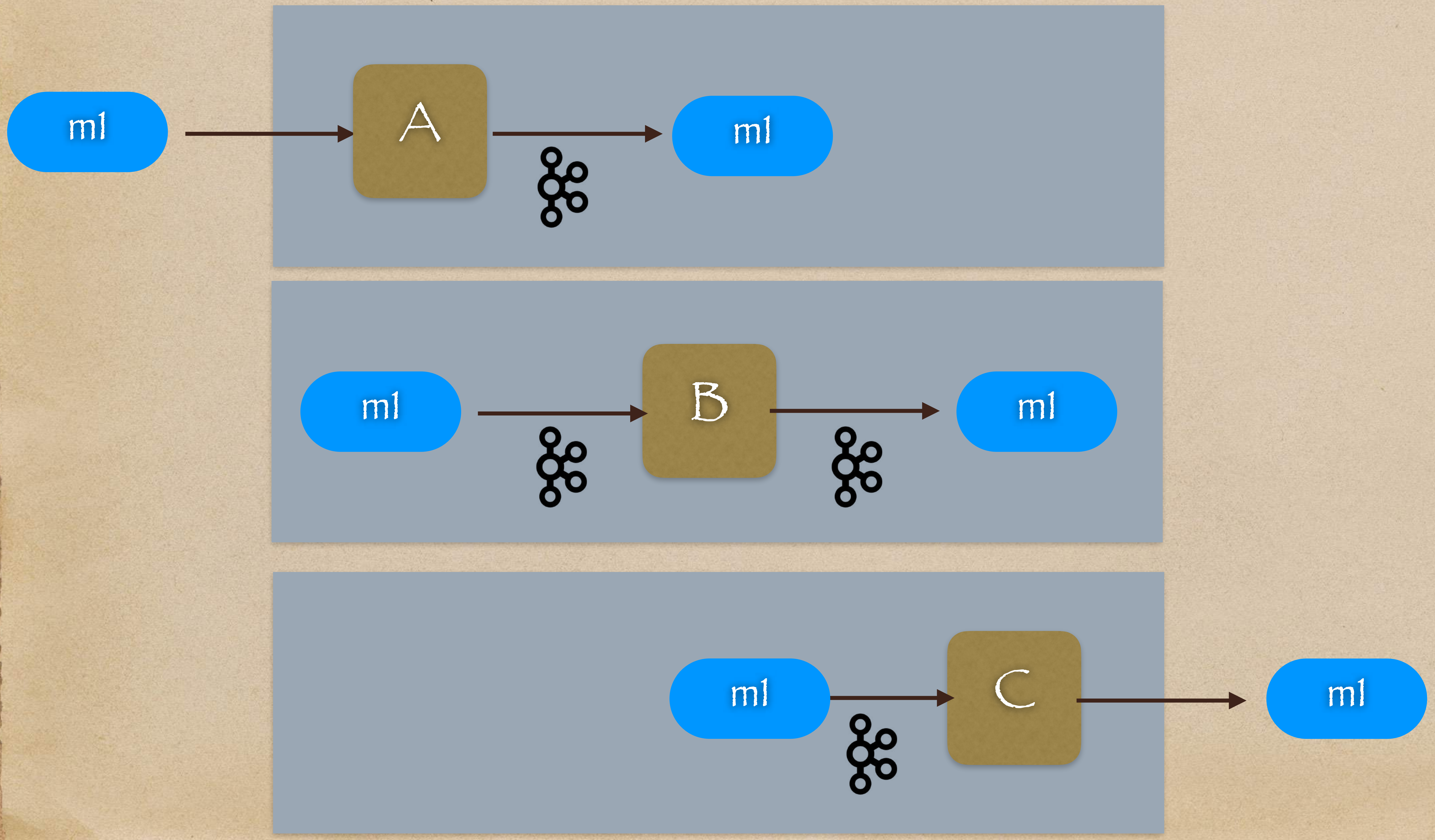
- In order to make this system reliable



- 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?

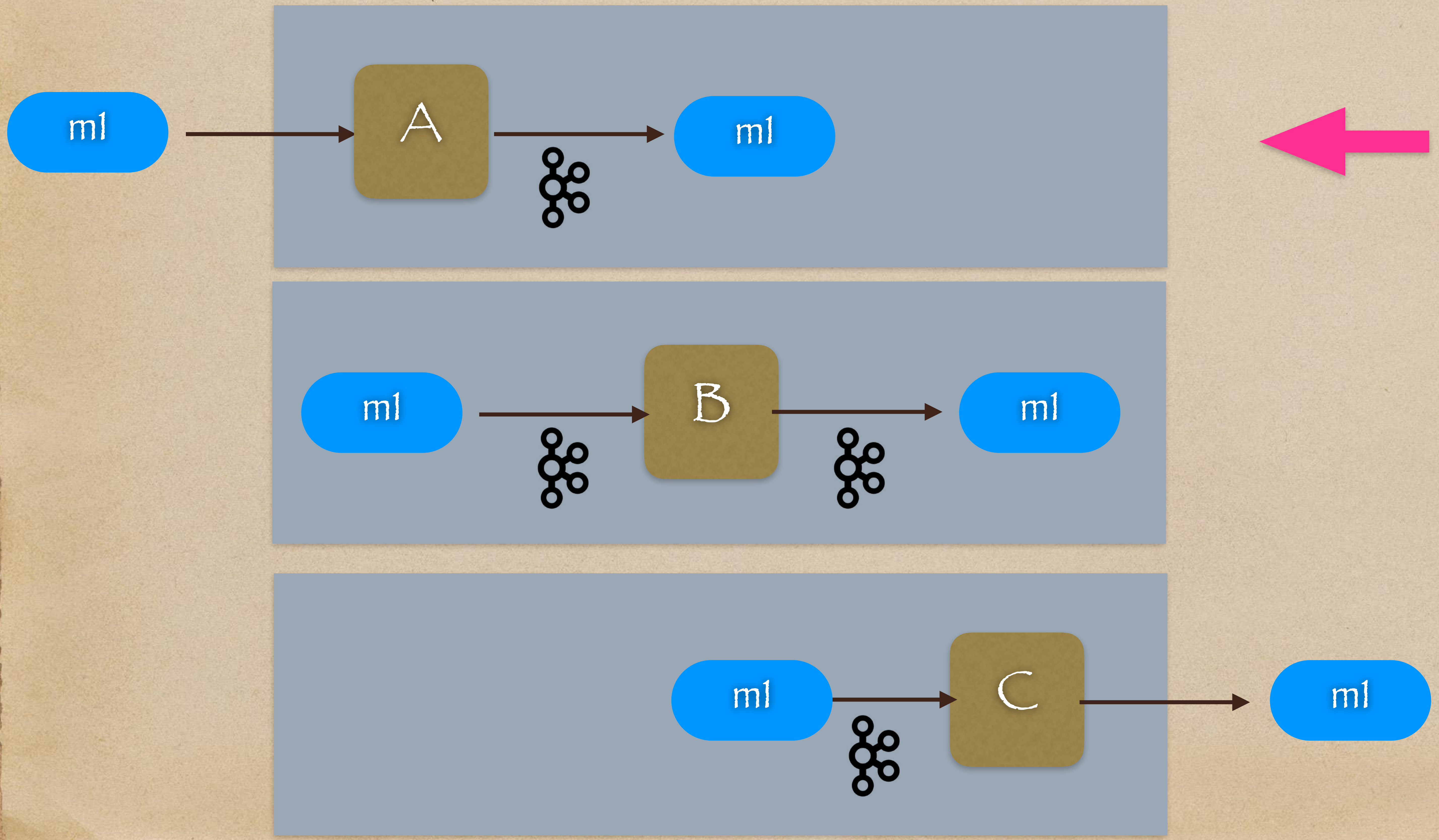
Reliability

- Let's first break this chain into its component processing links



Reliability

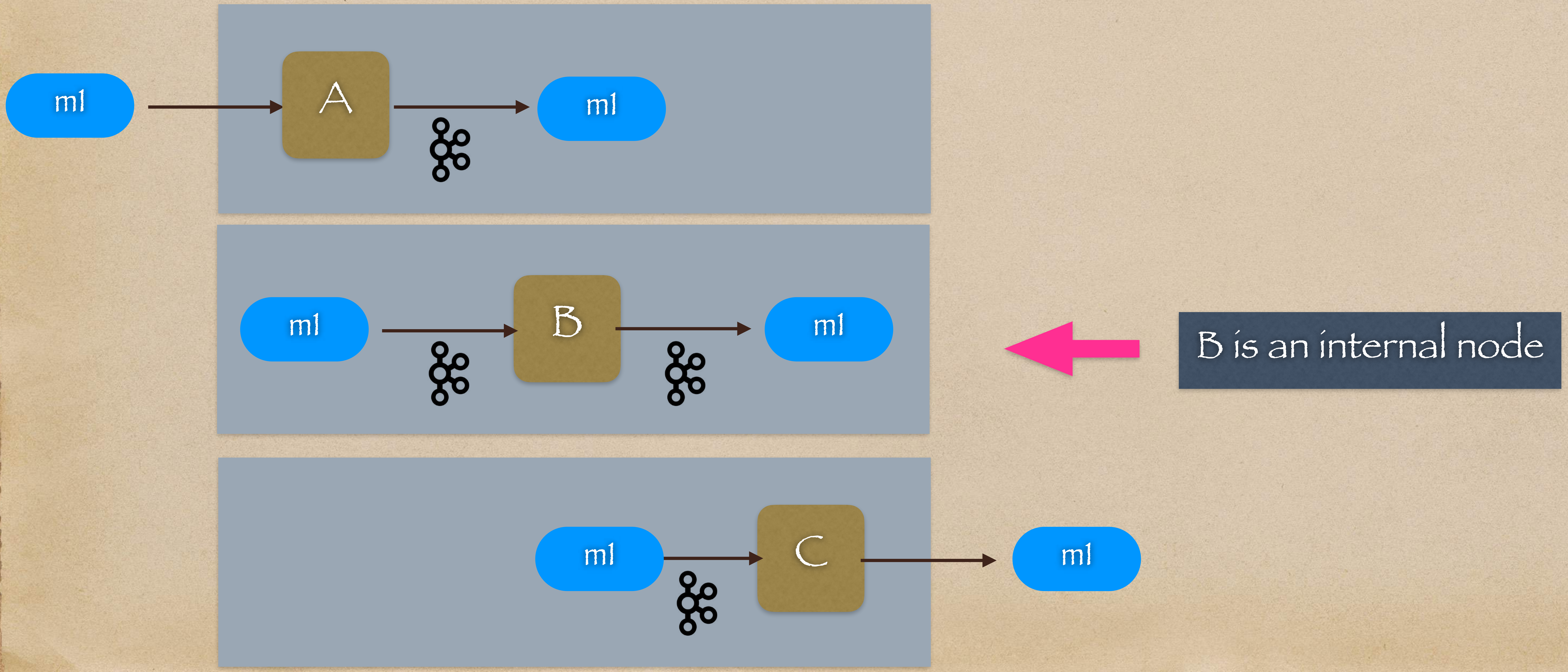
- Let's first break this chain into its component processing links



A is an ingest node

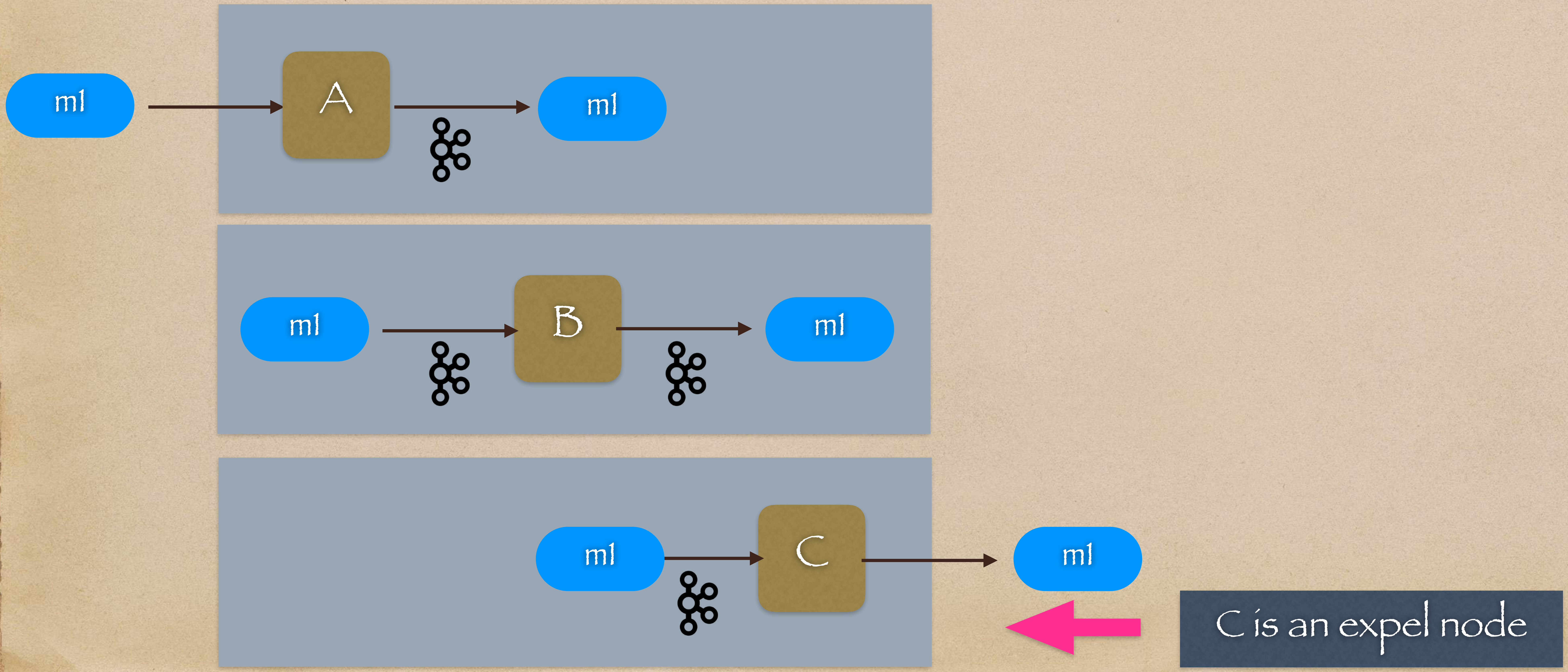
Reliability

- Let's first break this chain into its component processing links



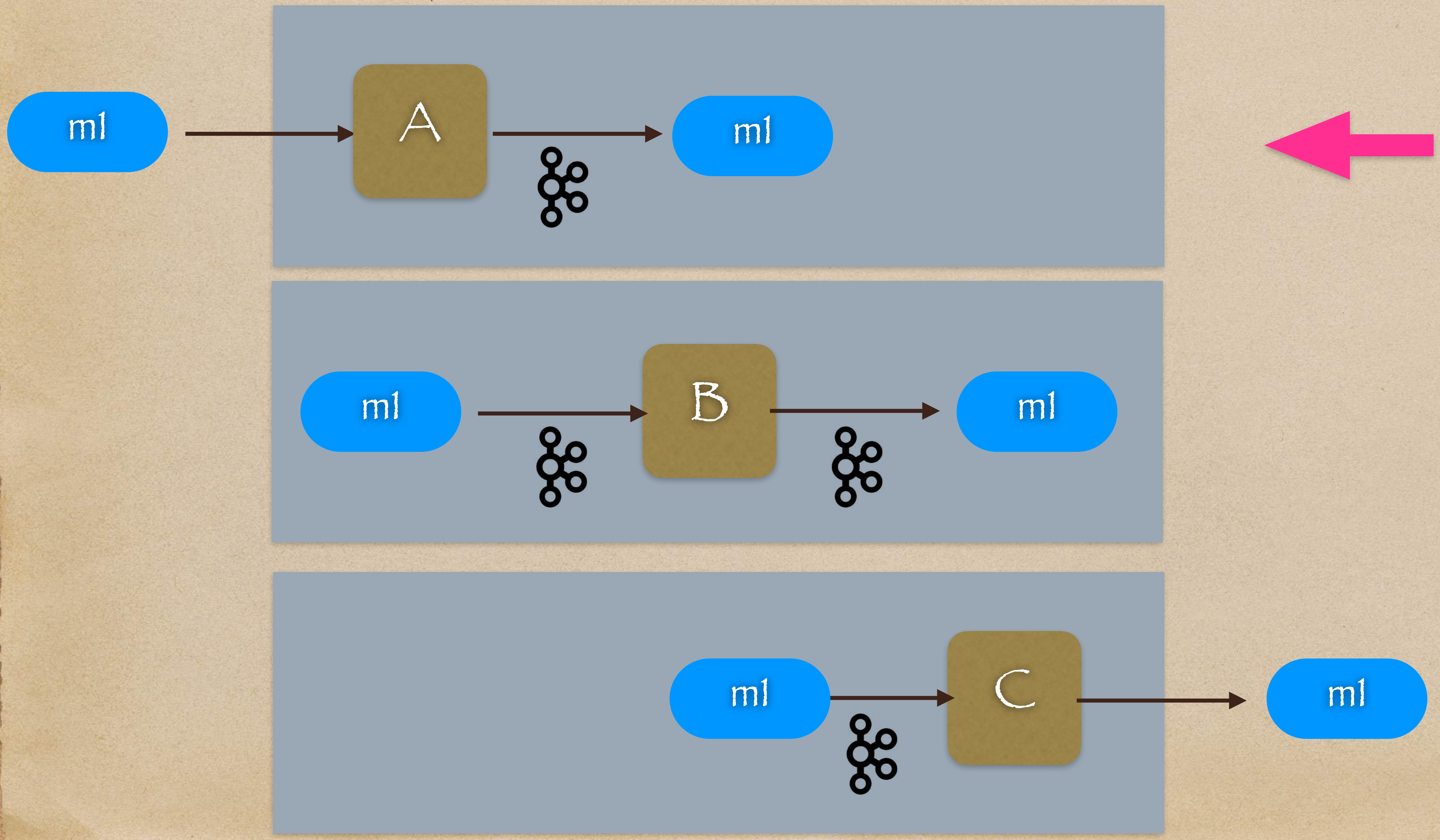
Reliability

- Let's first break this chain into its component processing links



Reliability

- But, how do we handle edge nodes A & C?

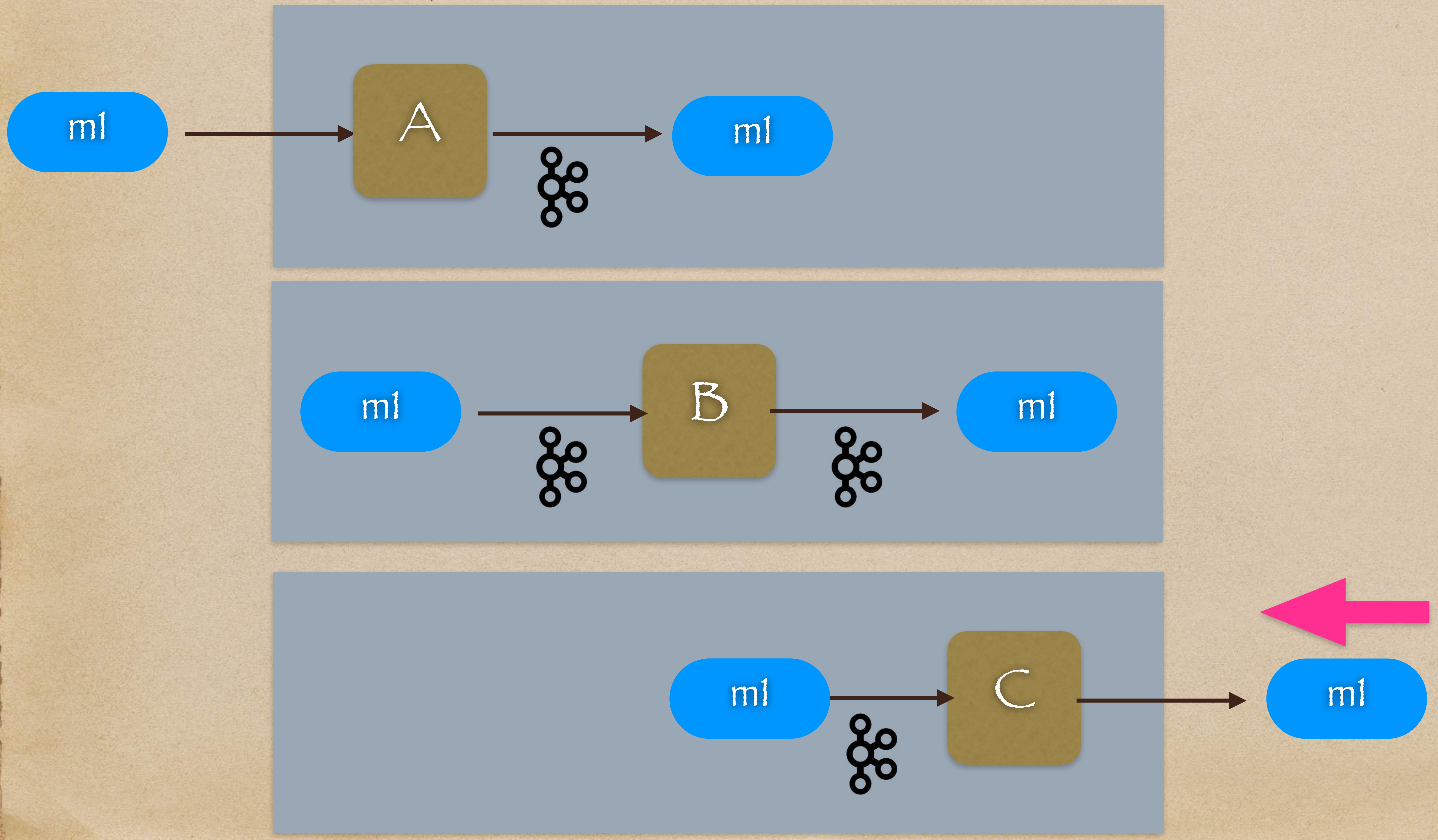


What does A need to do?

- 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?

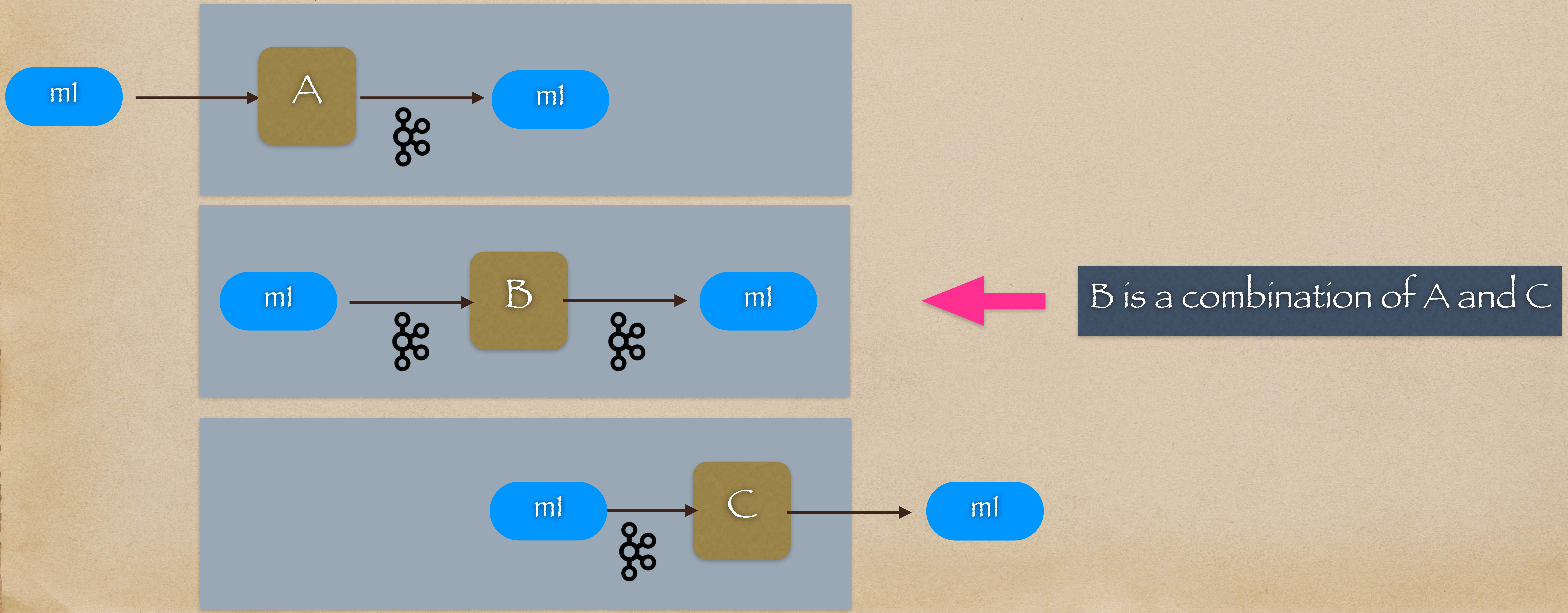


What does C need to do?

- 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

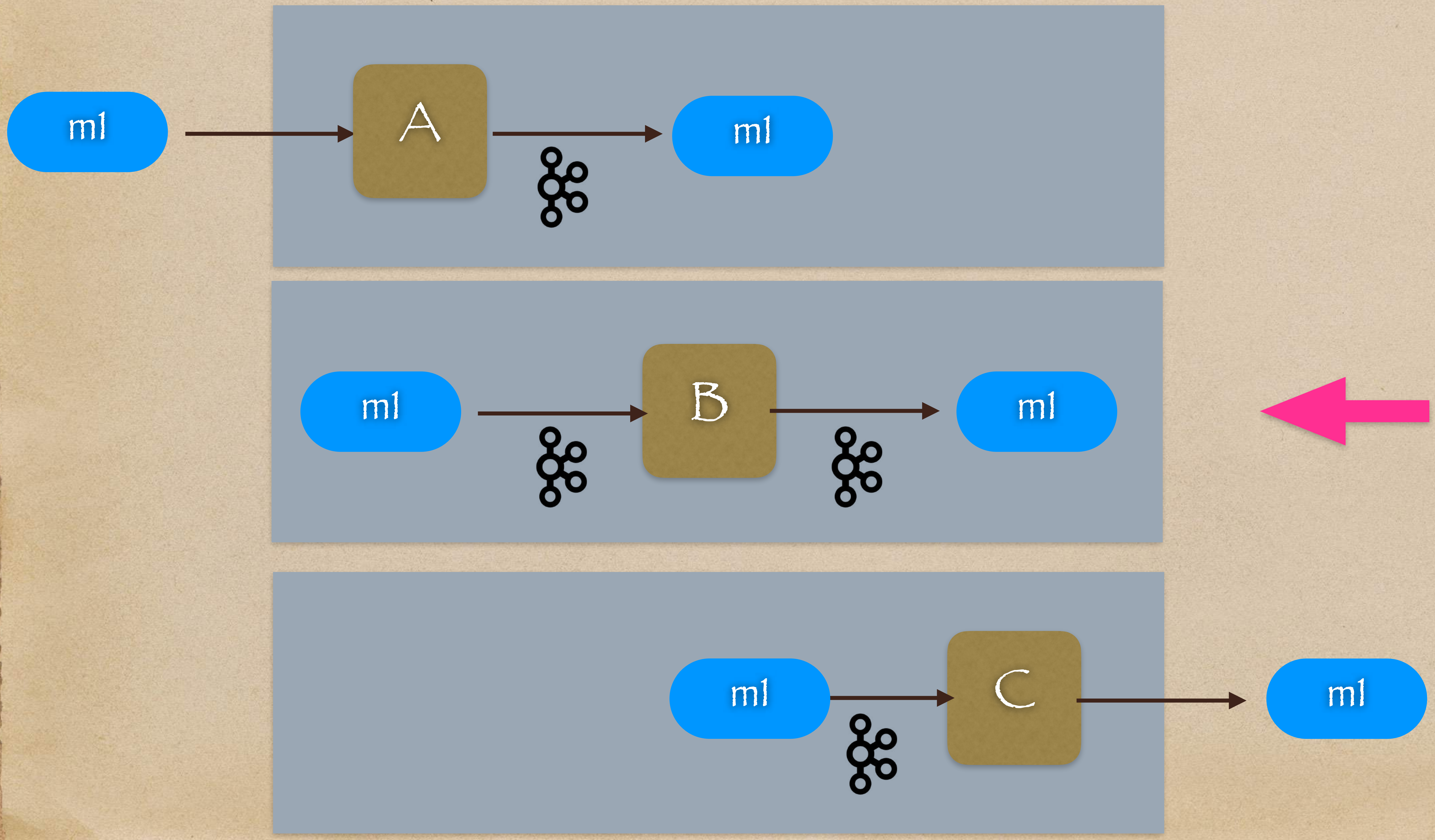
Reliability

- But, how do we handle edge nodes A & C?



Reliability

- But, how do we handle edge nodes A & C?

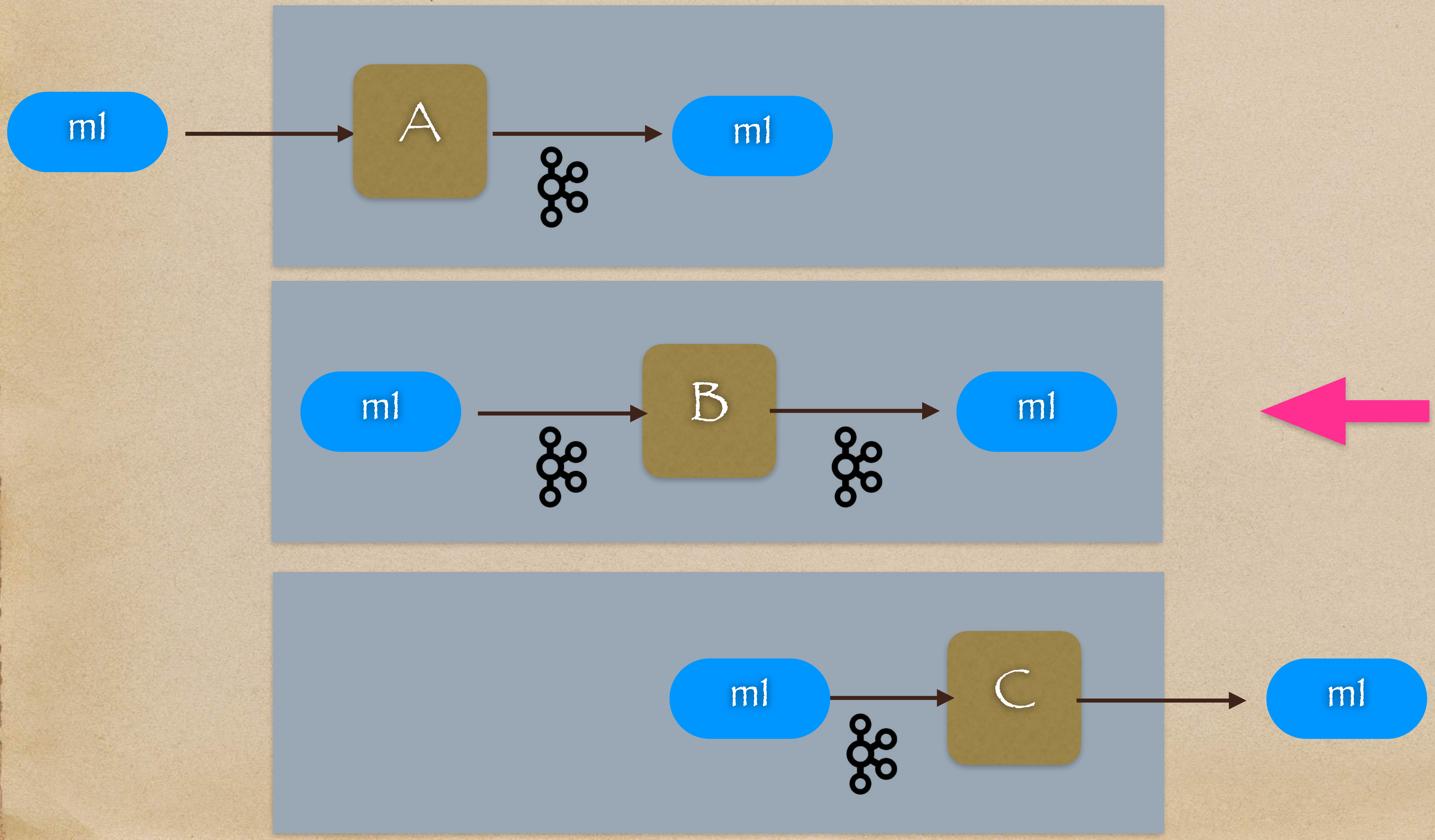


B needs to act like a reliable Kafka Producer

B is a combination of A and C

Reliability

- But, how do we handle edge nodes A & C?



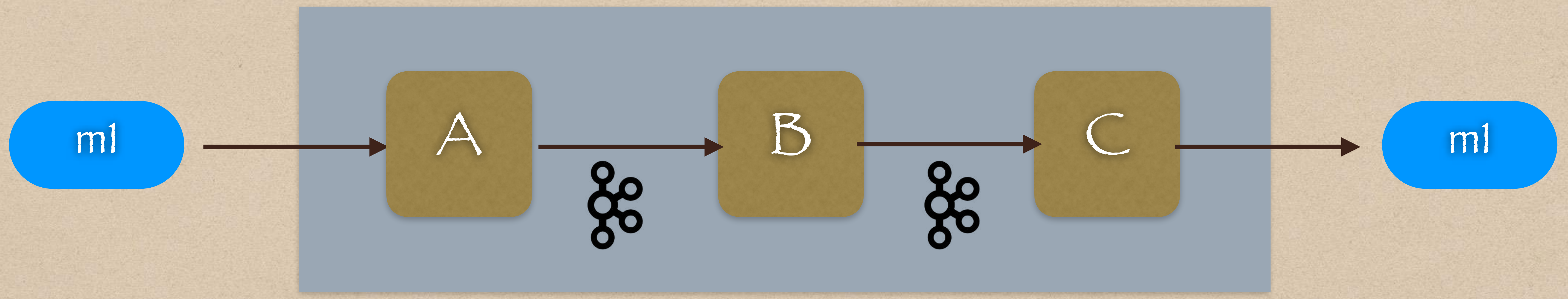
B needs to act like a reliable Kafka Producer

B is a combination of A and C

B needs to act like a reliable Kafka Consumer

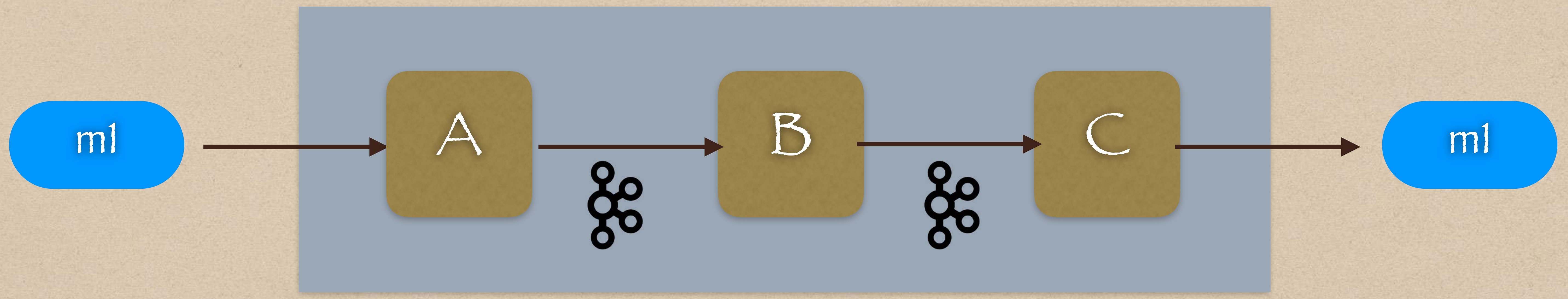
Reliability

- How reliable is our system now?



Reliability

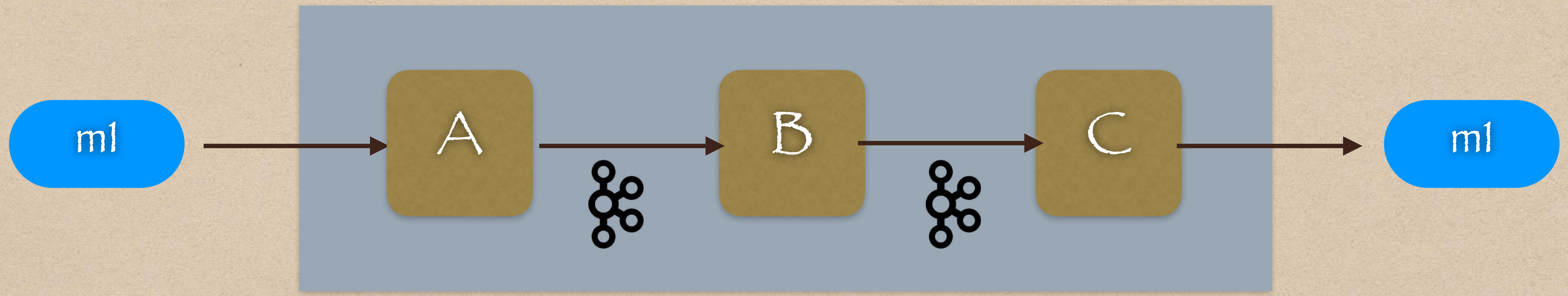
- How reliable is our system now?



- What happens if a process crashes?

Reliability

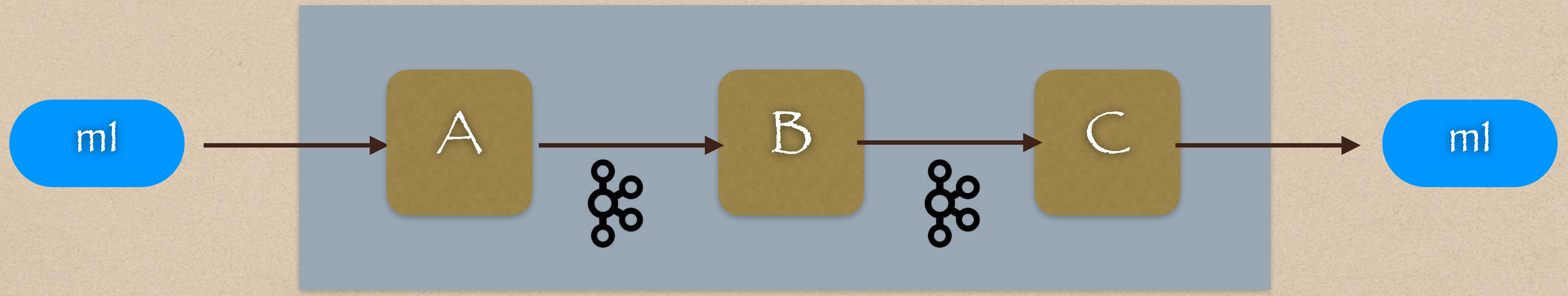
- How reliable is our system now?



- What happens if a process crashes?
- If A crashes, we will have a complete outage at ingestion!

Reliability

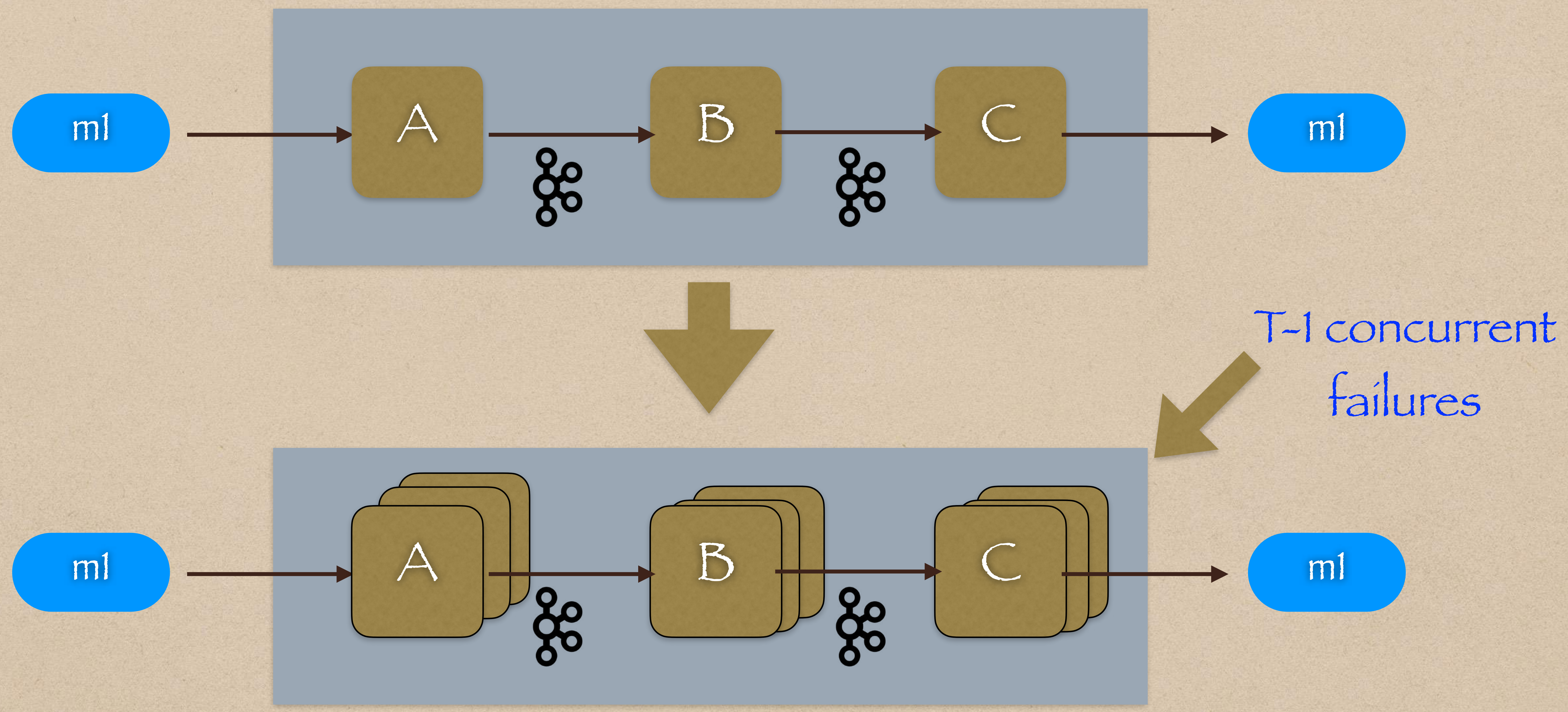
- How reliable is our system now?



- 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!

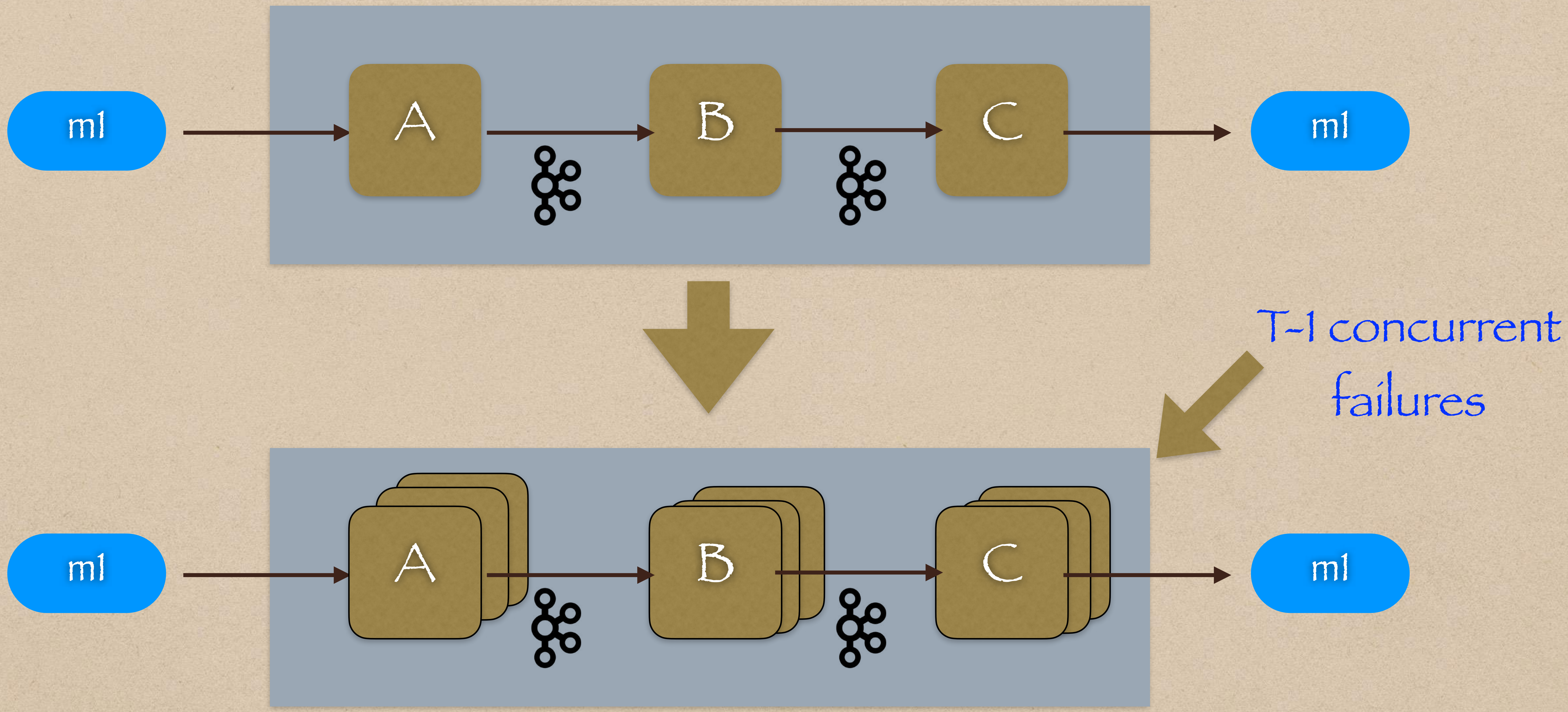
Reliability

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



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 : What is it?

- Lag is simply a measure of message delay in a system

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

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

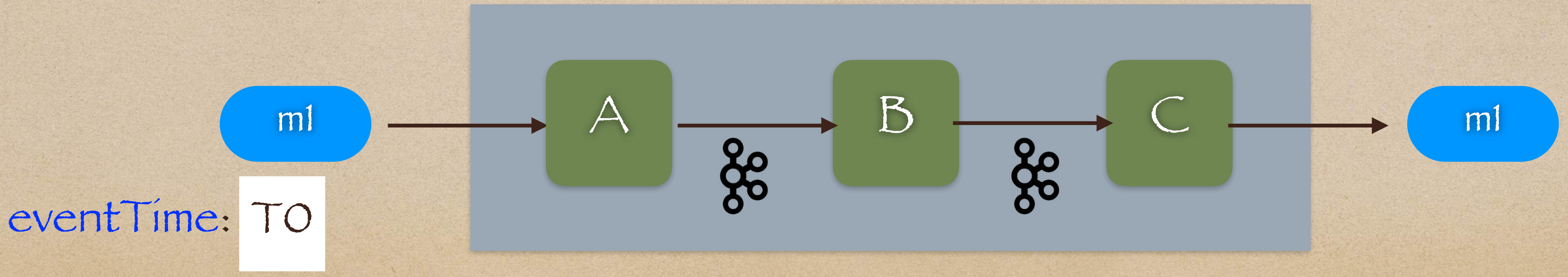
Lag : What is it?

- 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

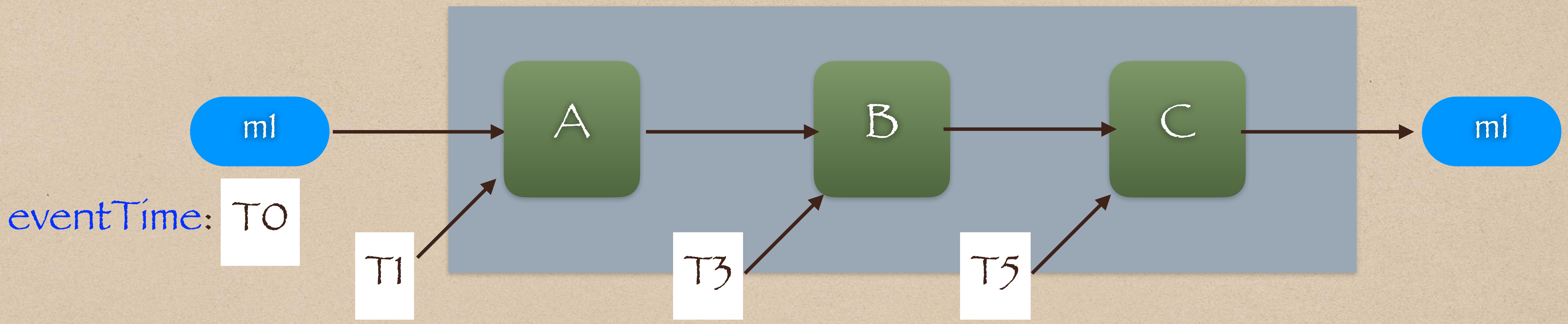
Lag : How do we compute it?

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)



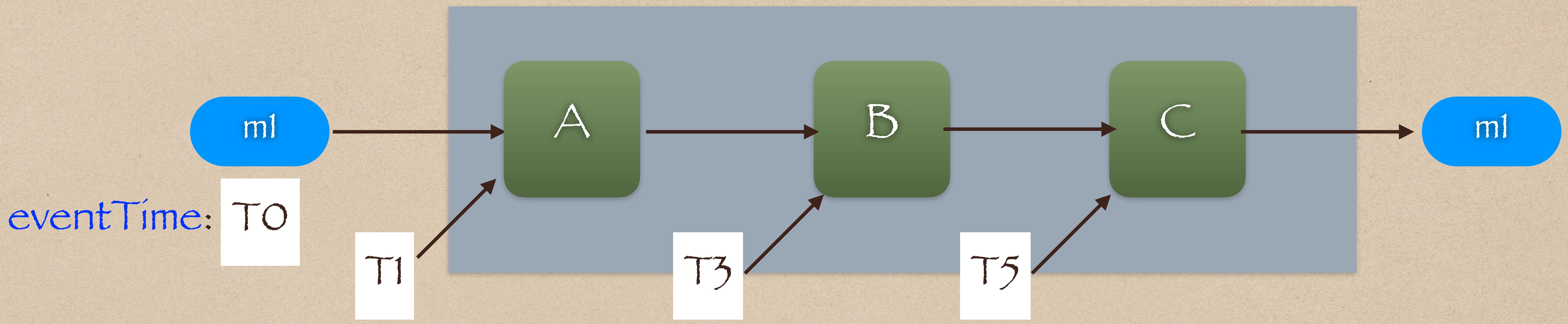
Lag : How do we compute it?



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

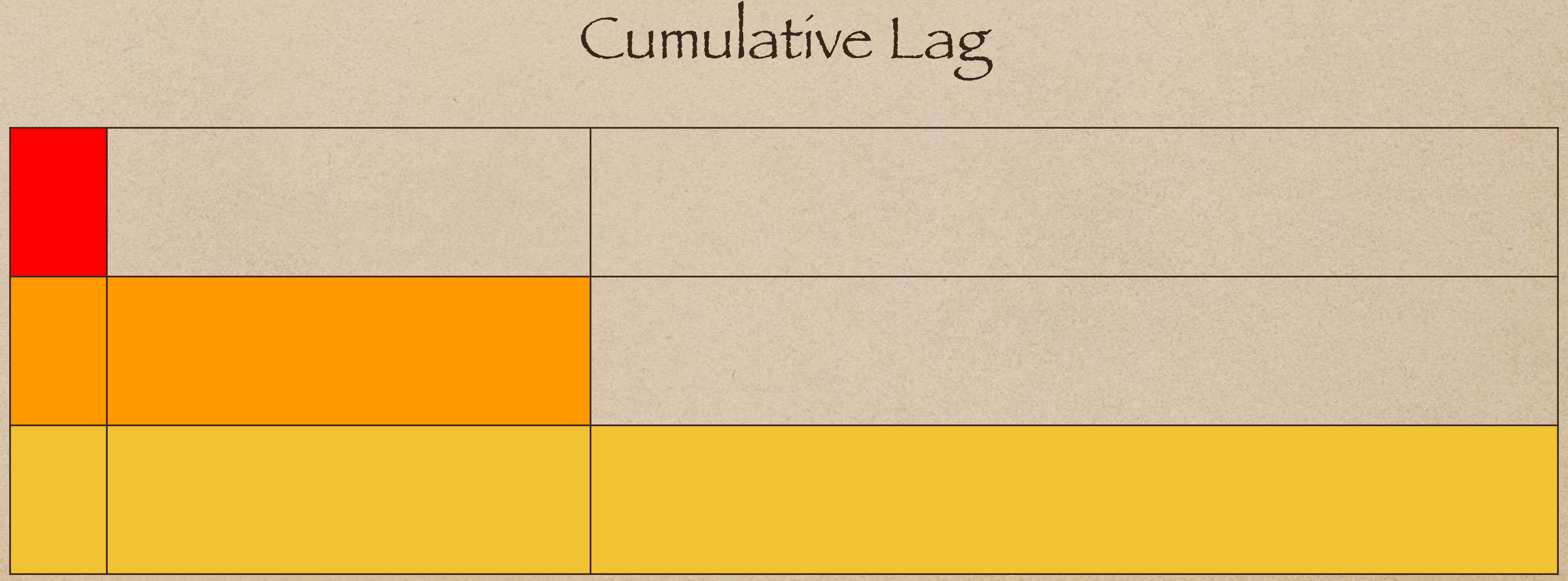
- Lag-in @
  - A = T1 - T0 (e.g 1 ms)
  - B = T3 - T0 (e.g 5 ms)
  - C = T5 - T0 (e.g 10 ms)

Lag : How do we compute it?



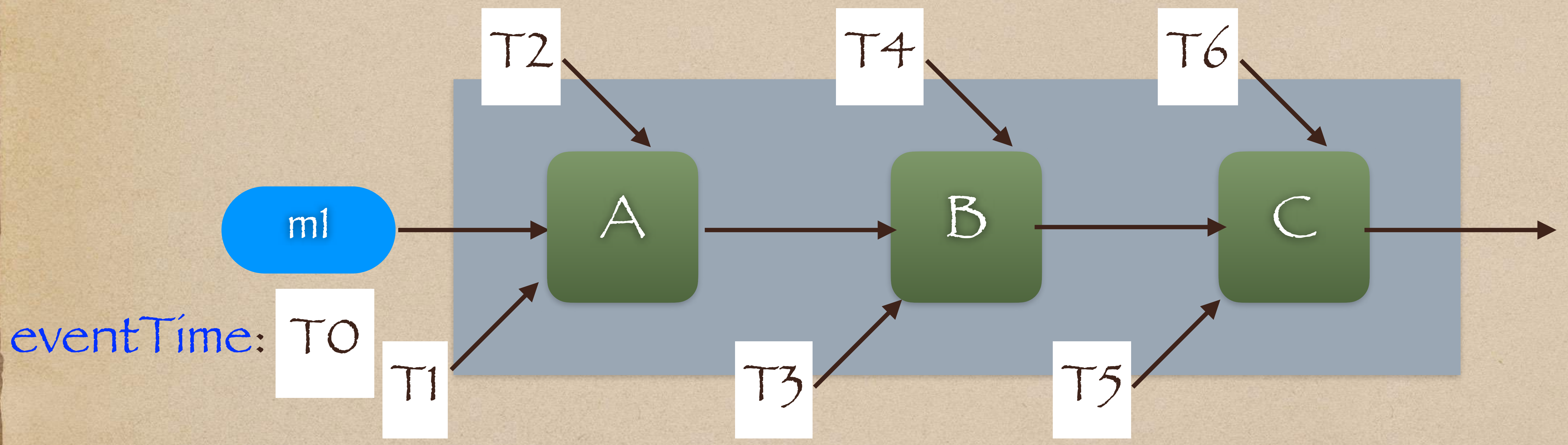
Arrival Lag (Lag-in) : time message arrives - eventTime

- Lag-in @
  - A = T1 - T0 (e.g 1 ms)
  - B = T3 - T0 (e.g 5 ms)
  - C = T5 - T0 (e.g 10 ms)



Lag : How do we compute it?

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

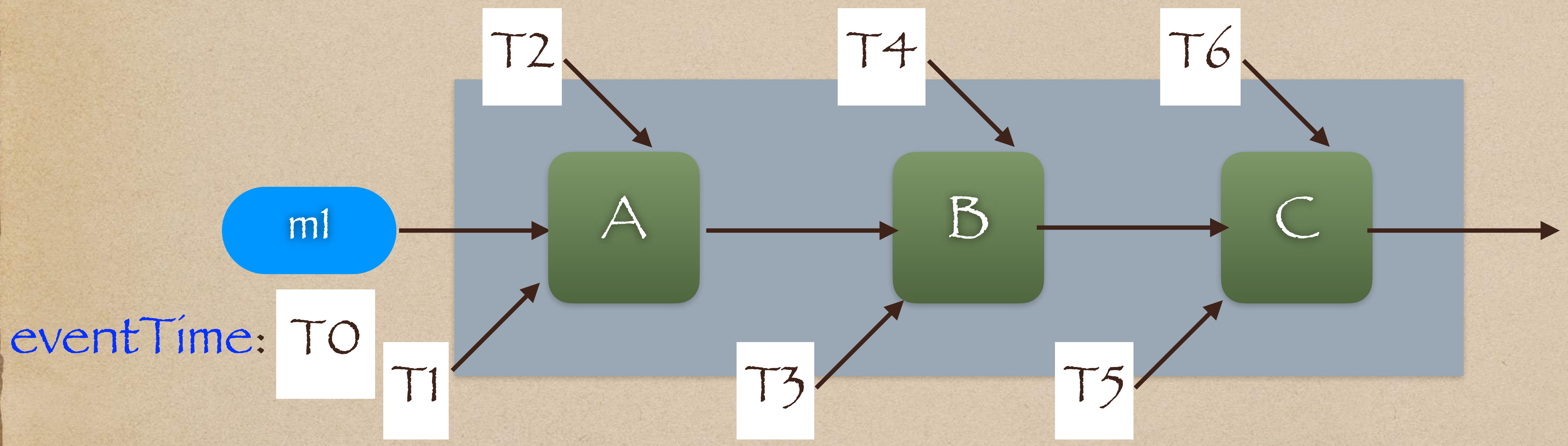


Arrival Lag (Lag-in): time message arrives - eventTime

- Lag-out @
  - A = T2 - T0 (e.g 3 ms)
  - B = T4 - T0 (e.g 8 ms)
  - C = T6 - T0 (e.g 12 ms)
- Lag-in @
  - A = T1 - T0 (e.g 1 ms)
  - B = T3 - T0 (e.g 5 ms)
  - C = T5 - T0 (e.g 10 ms)

Lag : How do we compute it?

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



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

E2E Lag is the total time a message spent in the system

- Lag-out @
  - A = T2 - T0 (e.g 3 ms)
  - B = T4 - T0 (e.g 8 ms)
  - C = T6 - T0 (e.g 12 ms) E2E Lag
- Lag-in @
  - A = T1 - T0 (e.g 1 ms)
  - B = T3 - T0 (e.g 5 ms)
  - C = T5 - T0 (e.g 10 ms)

Lag : How do we compute it?

- While it is interesting to know the lag for a particular message m1, it is of little use since we typically deal with millions of messages

Lag : How do we compute it?

- While it is interesting to know the lag for a particular message m1, it is of little use since we typically deal with millions of messages

- Instead, we prefer statistics (e.g. P95) to capture population behavior

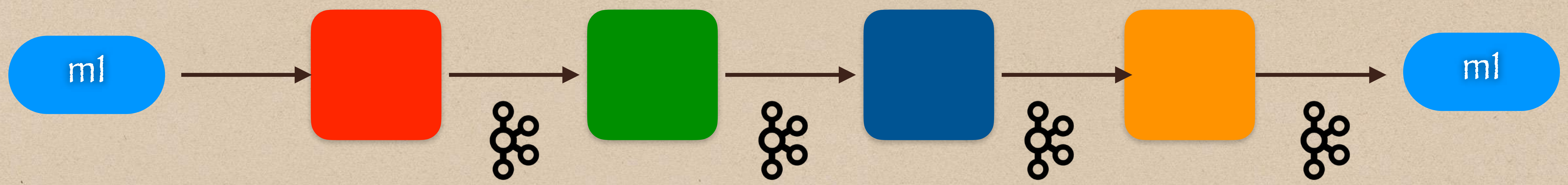
Lag : How do we compute it?

- 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

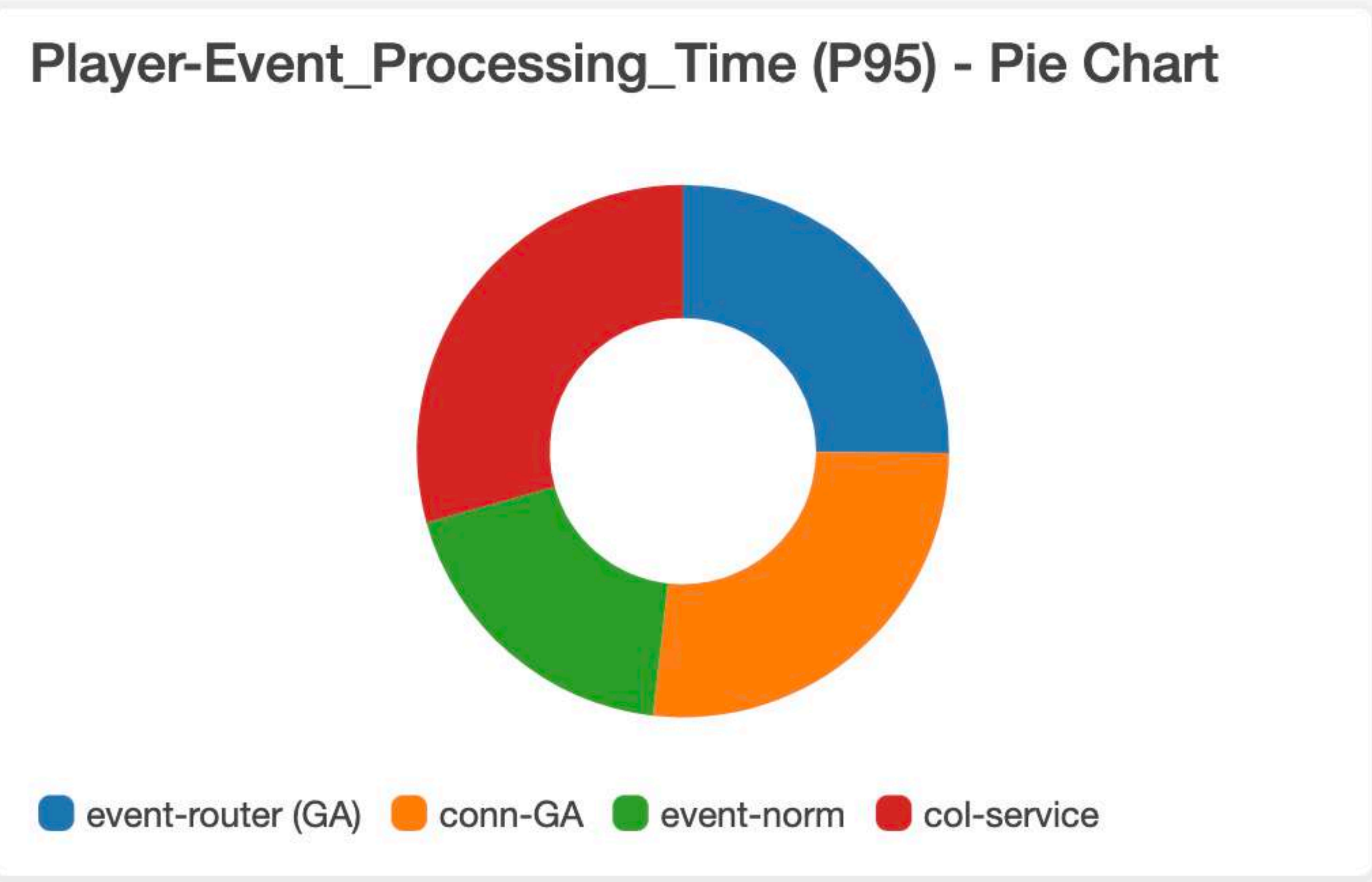
Lag : How do we compute it?

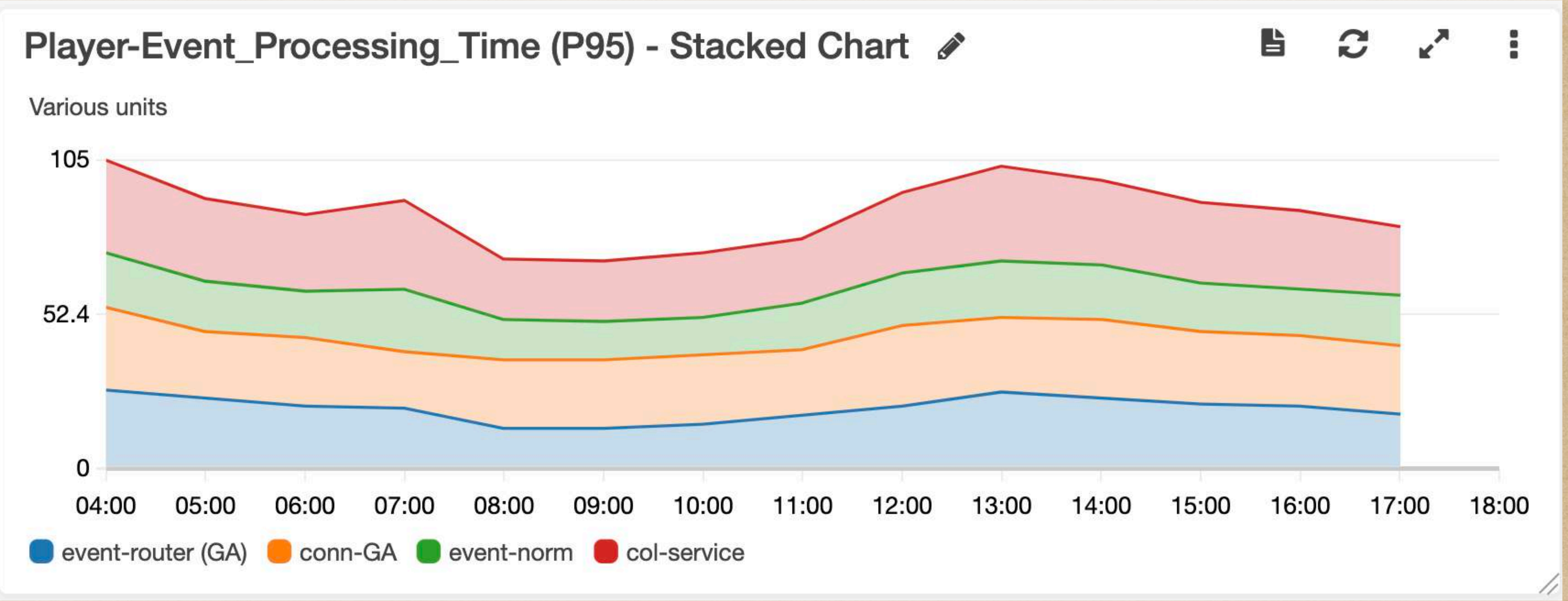
- Some useful Lag statistics are:
  - E2E Lag (p95) : 95th percentile time of messages spent in the system
  - Lag_[in|out] (N, p95): P95 Lag_in or Lag_out at any Node N
  - Process_Duration(N, p95) : Lag_out(N, p95) - Lag_in(N, p95)

Lag : How do we compute it?



- Process_Duration Graphs show you the contribution to overall Lag from each hop





Loss : What is it?

Loss : What is it?

- Loss is simply a measure of messages lost while transiting the system

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!

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

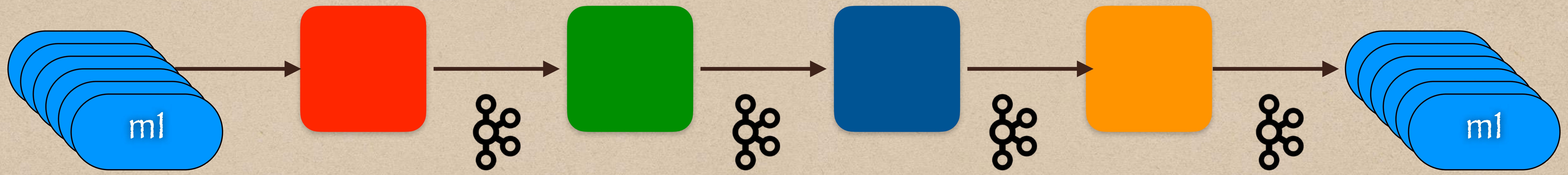
Loss : What is it?

- 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

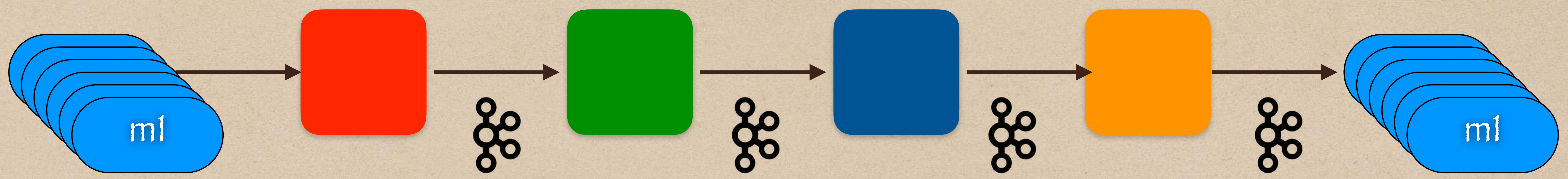
Loss : How do we compute it?

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

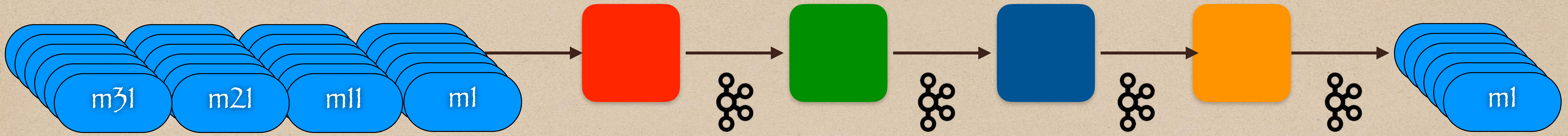


Loss : How do we compute it?



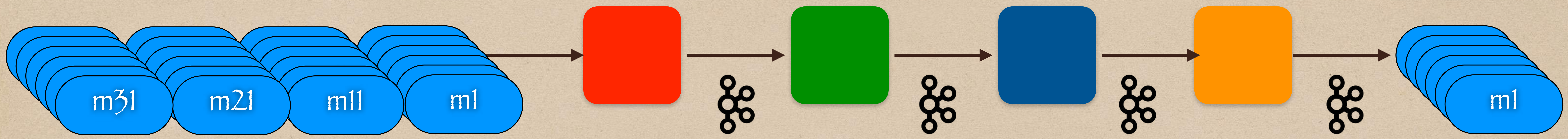
Message Id					E2E Loss	E2E Loss %
m1	1	1	1	1		
m2	1	1	1	1		
m3	1	0	0	0		
...	...	...	...	...		
m10	1	1	0	0		
Count	**10**	**9**	**7**	**5**		
Per Node Loss(N)	**0**	**1**	**2**	**2**	**5**	**50%**

Loss : How do we compute it?



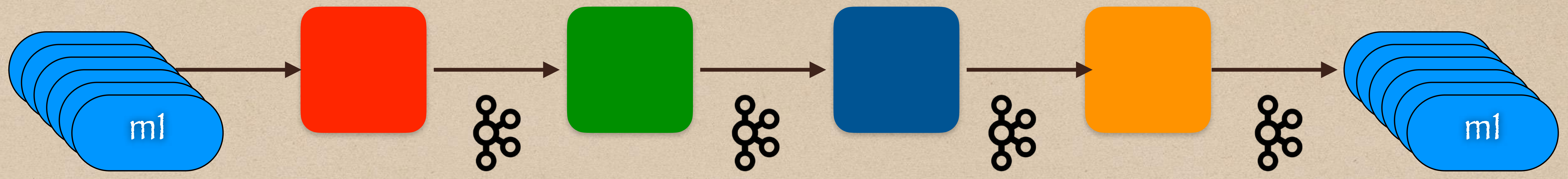
- In a streaming data system, messages never stop flowing. So, how do we know when to count?

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

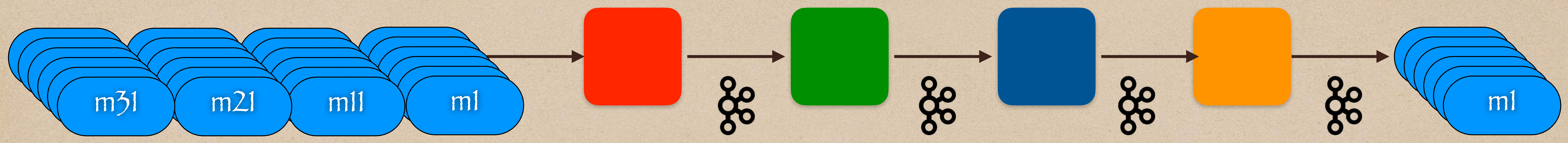
Loss : How do we compute it?



@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?



- 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
  - 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.

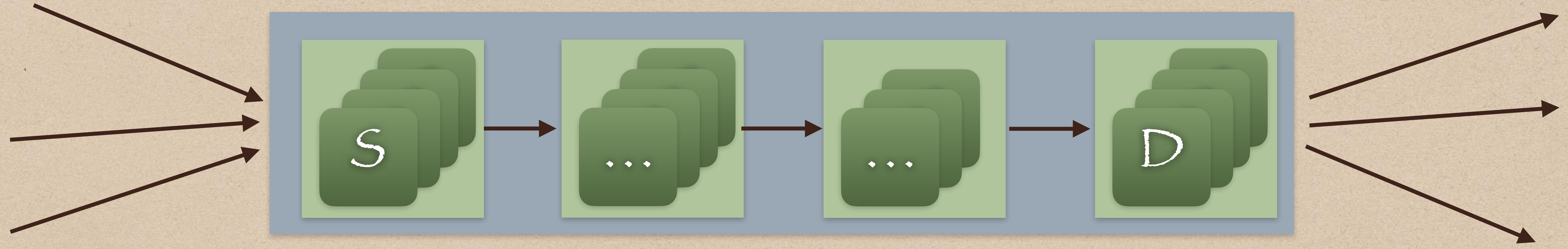
- But wait…

Performance

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

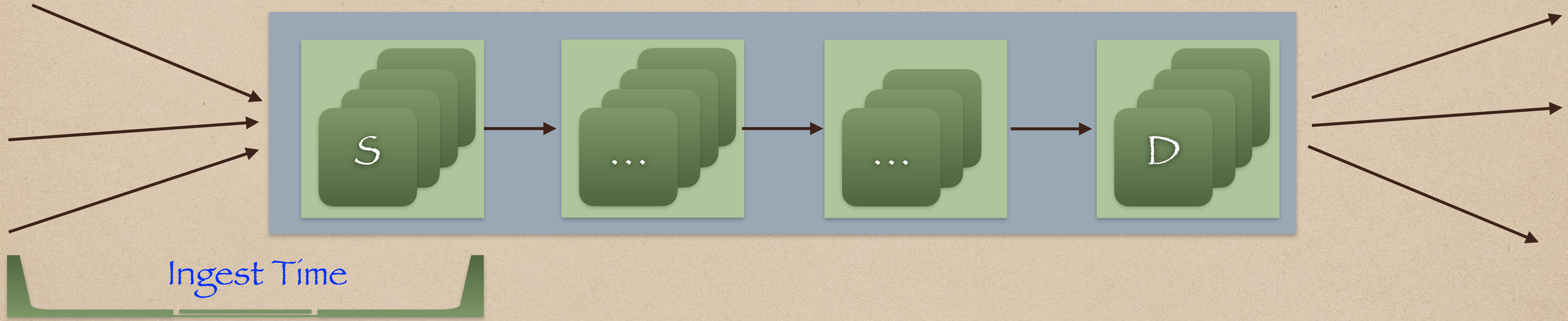
Performance

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



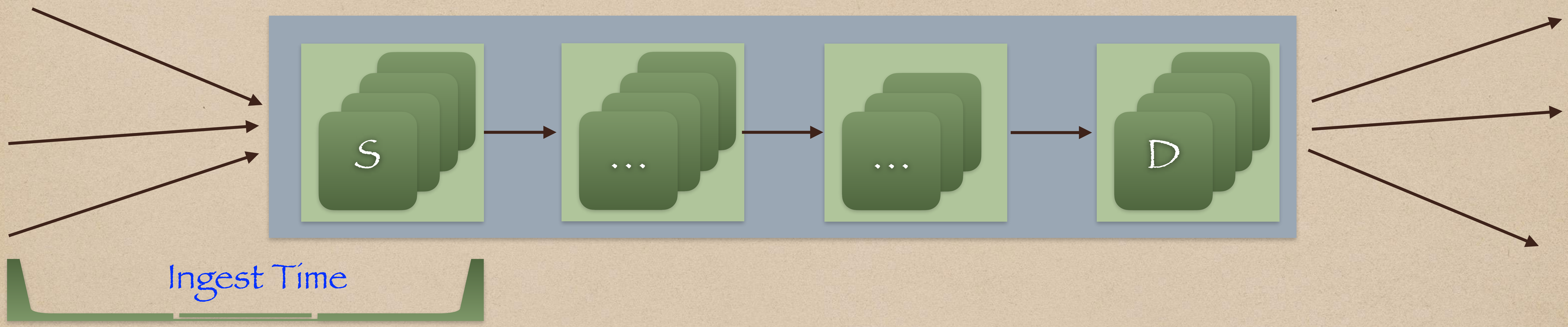
- To understand streaming system performance, let's understand the components of E2E Lag

Performance



Ingest Time : Time from Last_Byte_In_of_Request to First_Byte_Out_of_Response

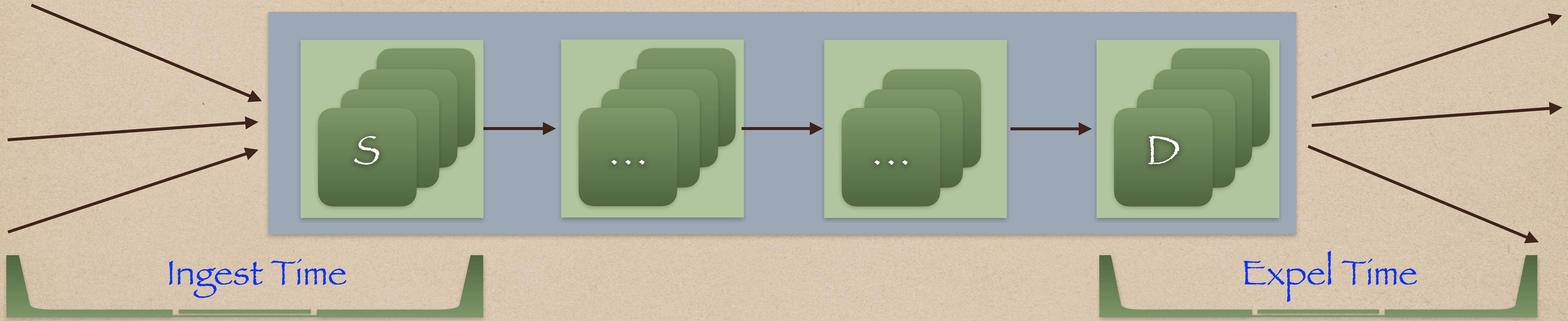
Performance



Ingest Time : Time from Last_Byte_In_of_Request to First_Byte_Out_of_Response

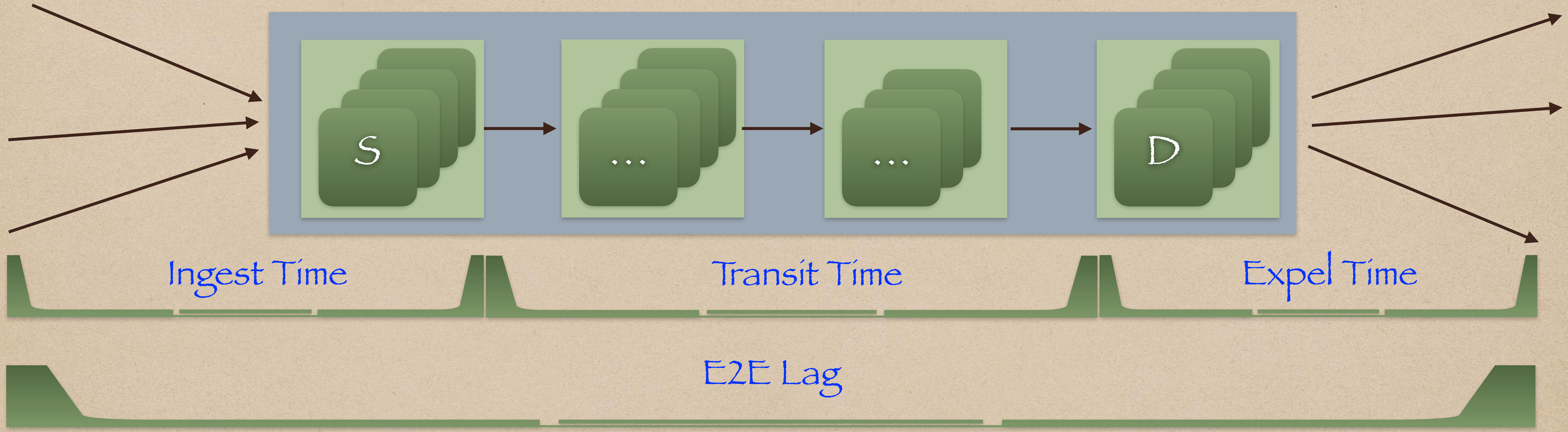
- This time includes overhead of reliably sending messages to Kafka

Performance



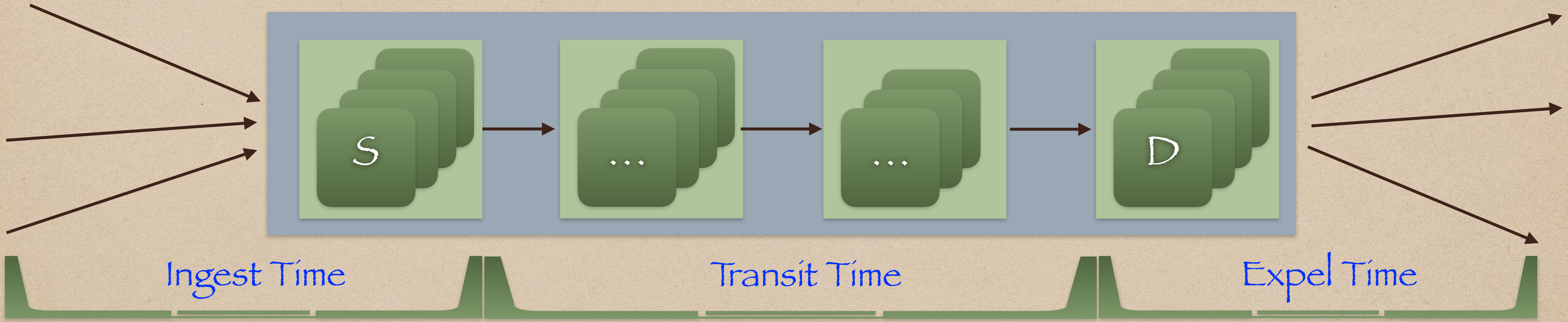
Expel Time : Time to process and egest a message at D.

Performance



E2E Lag : Total time messages spend in the system from message ingest to expel!

Performance



Transit Time : Rest of the time spent in the data pipe (i.e. internal nodes)

Performance Penalties

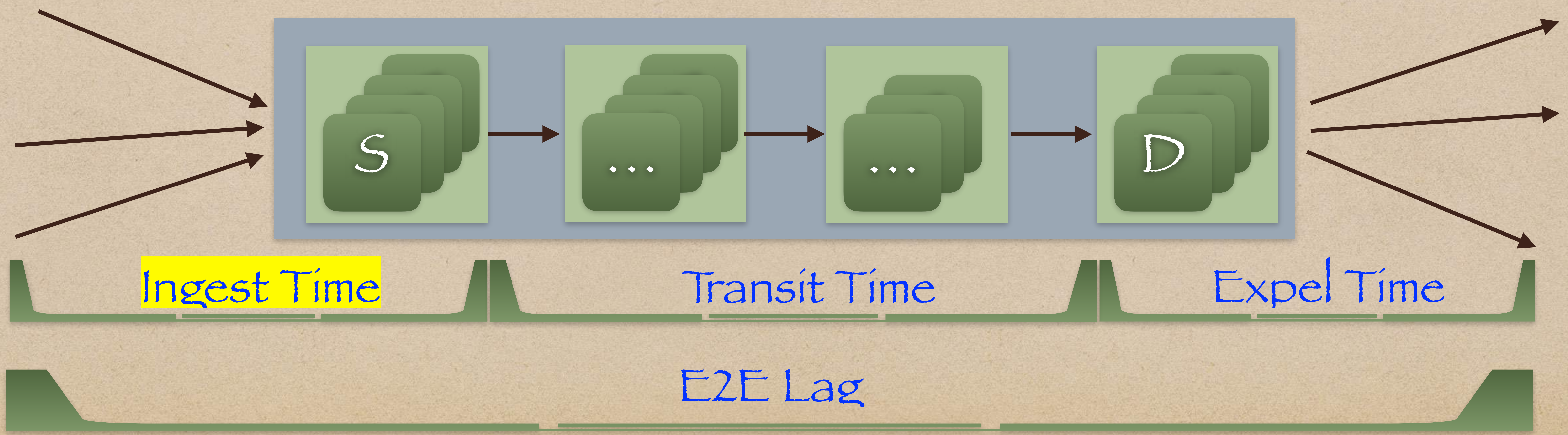
(Trading of Latency for Reliability)

Performance : Penalties

- In order to have stream reliability, we must sacrifice latency!
- How can we handle our performance penalties?

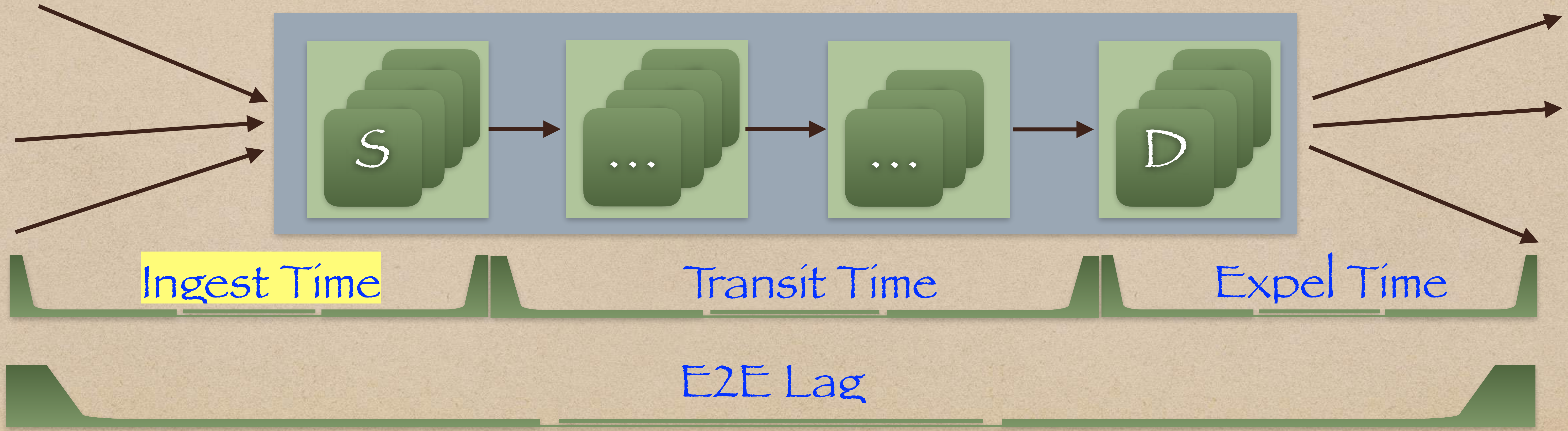
Performance

- 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



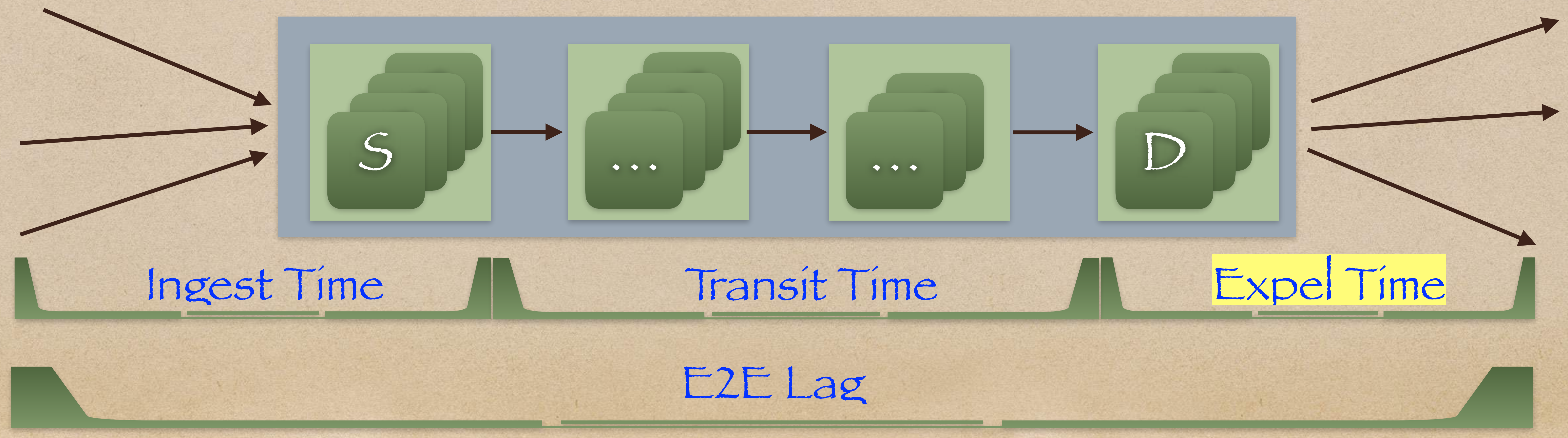
Performance

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



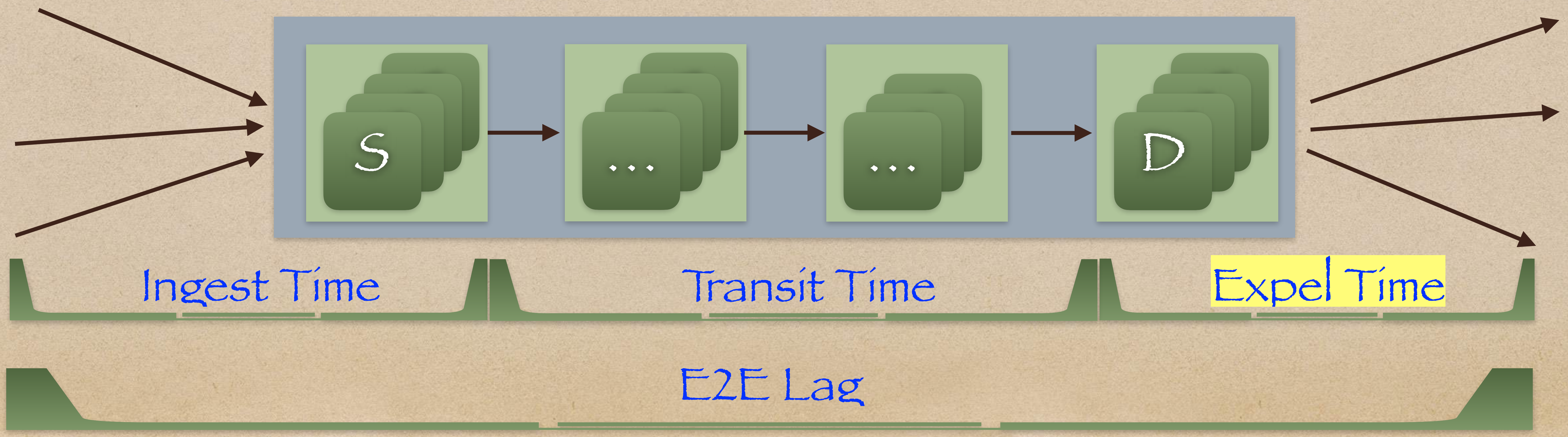
Performance

- 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

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

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

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

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

Global 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

Performance - Tiered Retries

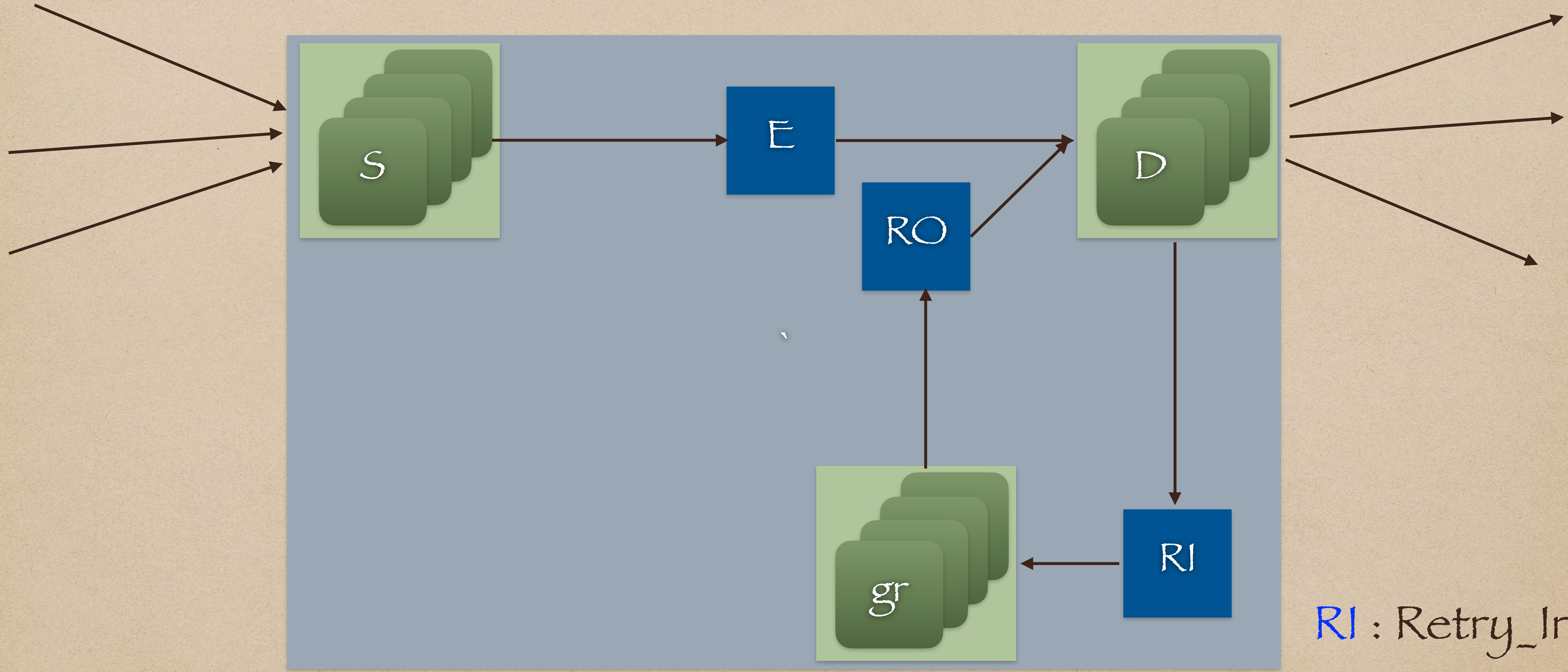
Local Retries

- 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

Scalability

- First, Let's dispel a myth!

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

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

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

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

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

The most important part of autoscaling is picking the right metric to trigger autoscaling actions

Scalability - Autoscaling EC2

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

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

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

What Next?

We now have a system with the Non-functional Requirements (NFRs) that we desire!

What Next?

What if we want to handle

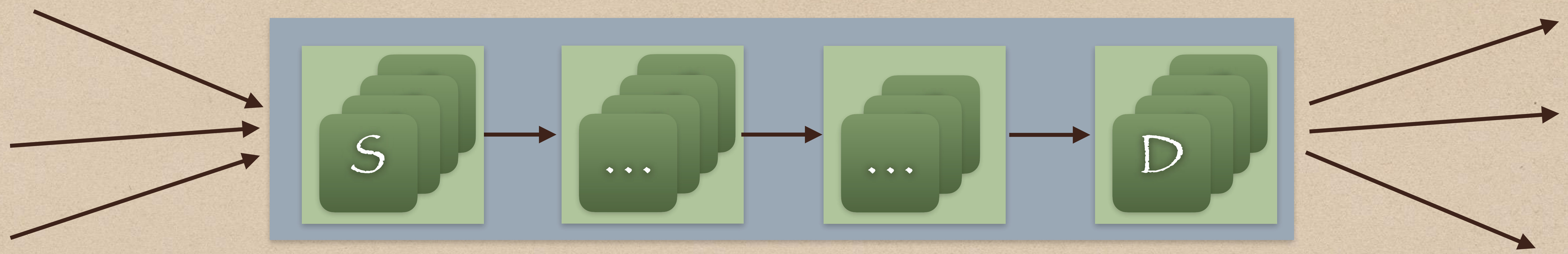
- Different types of messages
- More complex processing (i.e. more processing stages)
- More complex stream topologies (e.g. 1-1, 1-many, many-many)

What Next?

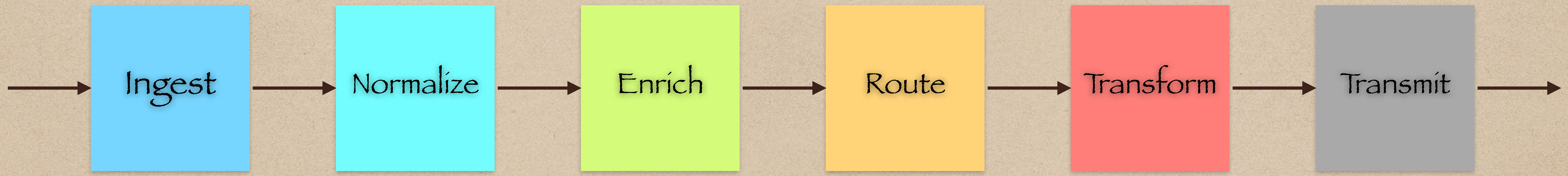
It will take a lot of work to rebuild our data pipe for each variation of customers' needs!

What we need to do is build a more generic Streams-as-a-Service (STaaS) platform!

Building StaaS

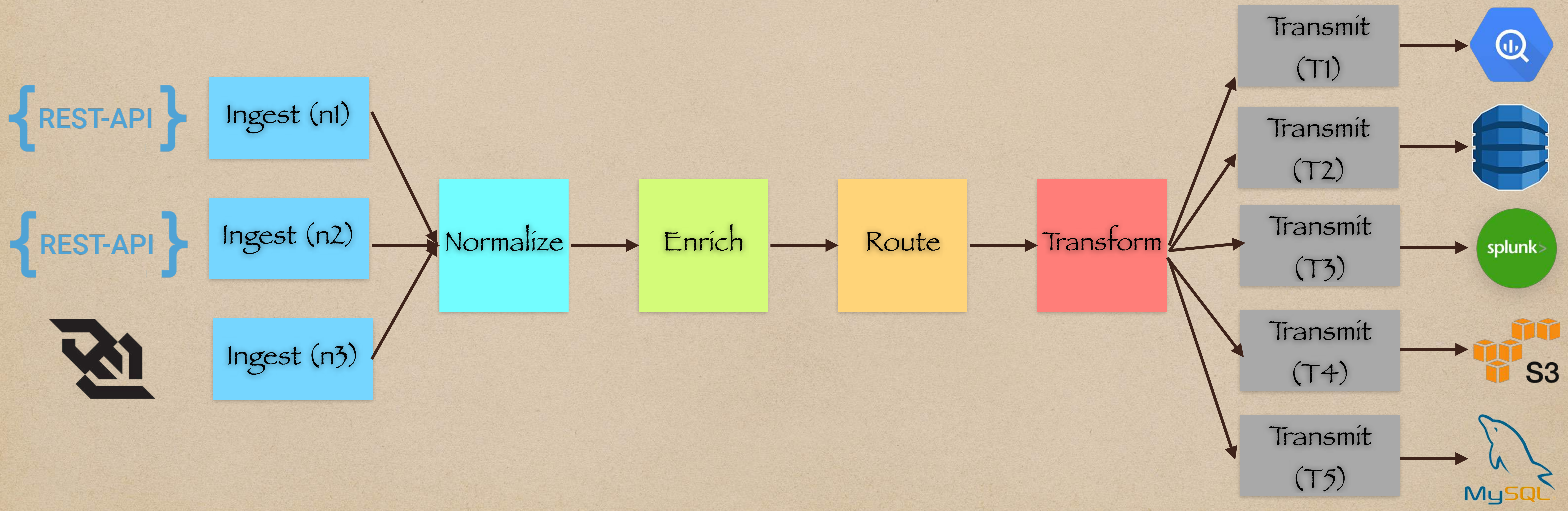


- Firstly, let's make our pipeline a bit more realistic by adding more processing stages



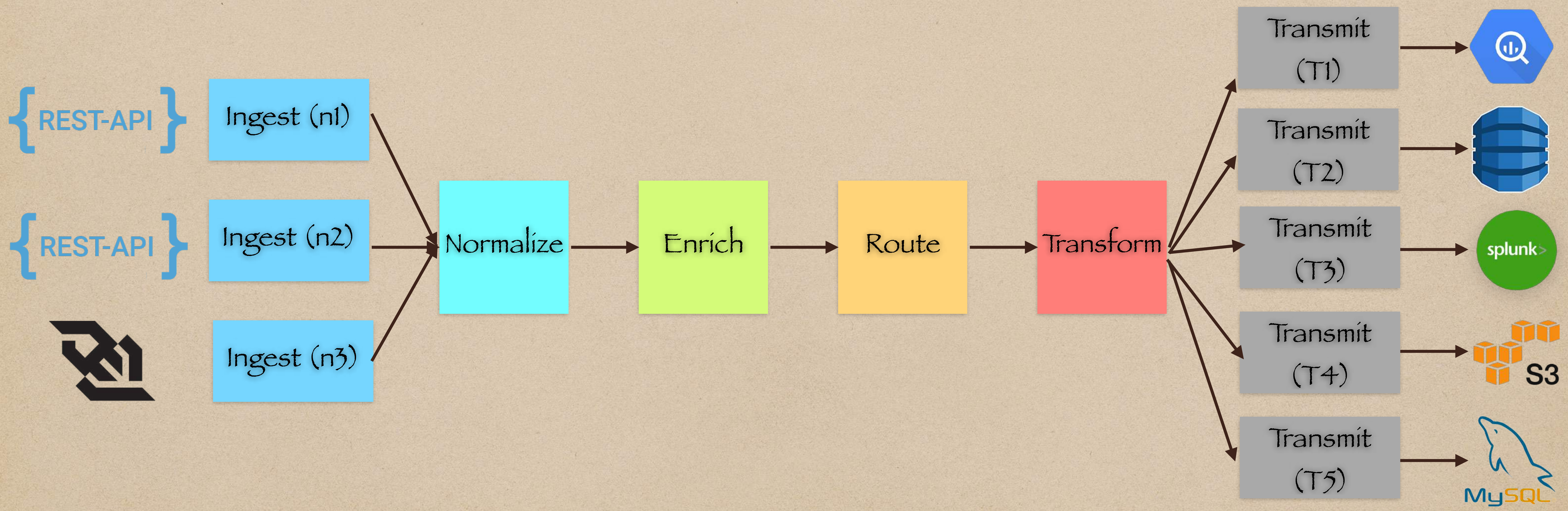
Building StaaS

- And by handling more complex topologies (e.g. many-to-many)



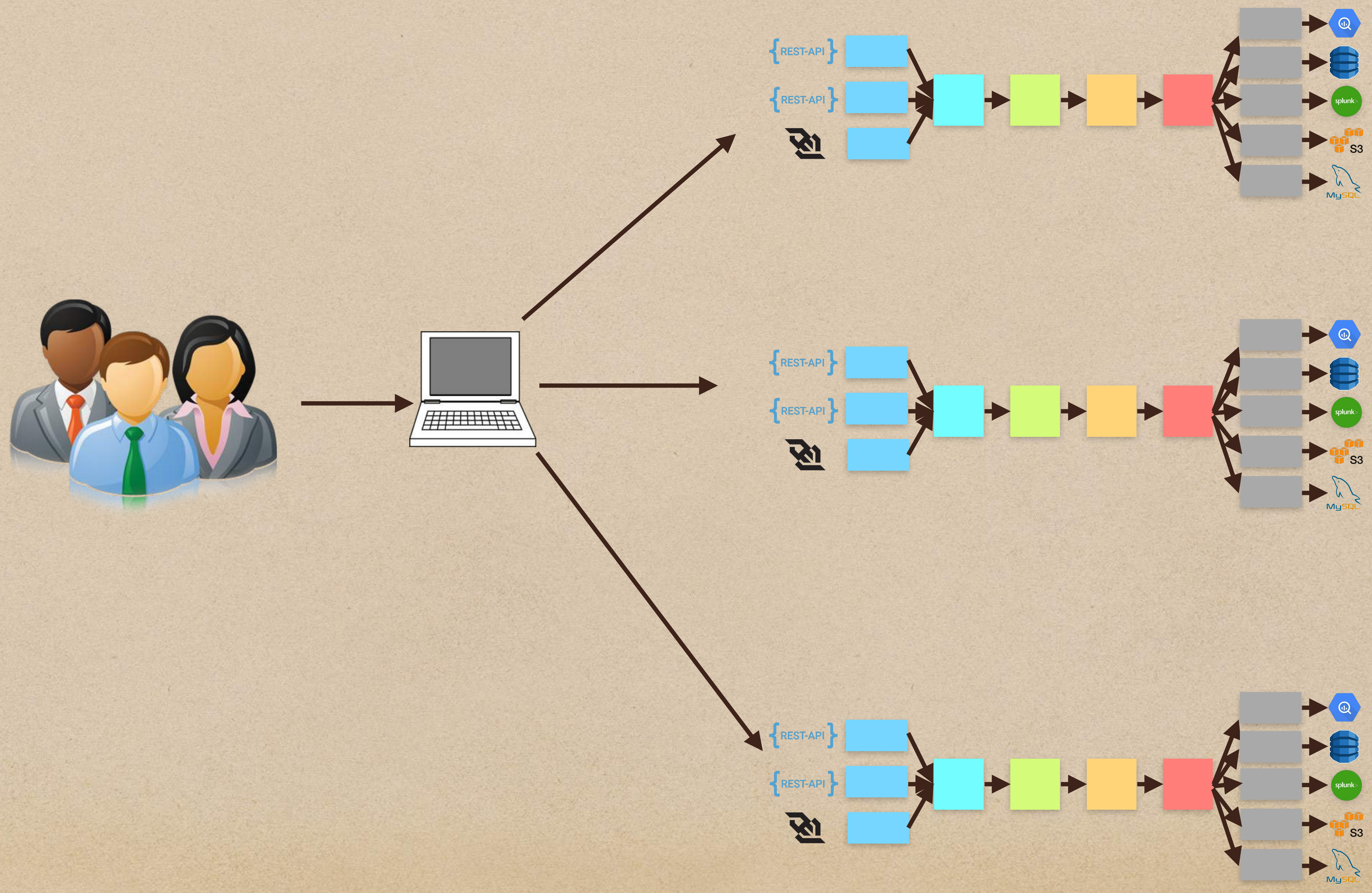
Building StaaS

This our data plane — it send messages from multiple sources to multiple destinations



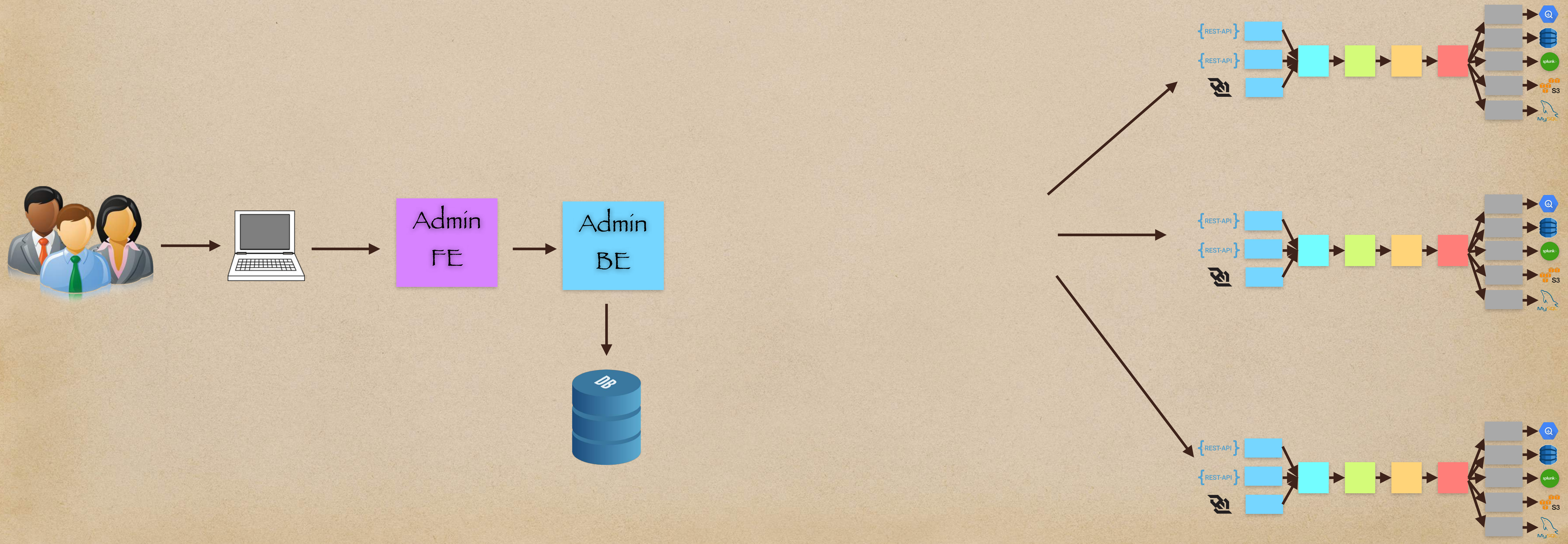
Building StaaS

But, we also want to allow users the ability to define their own data pipes in this data plane



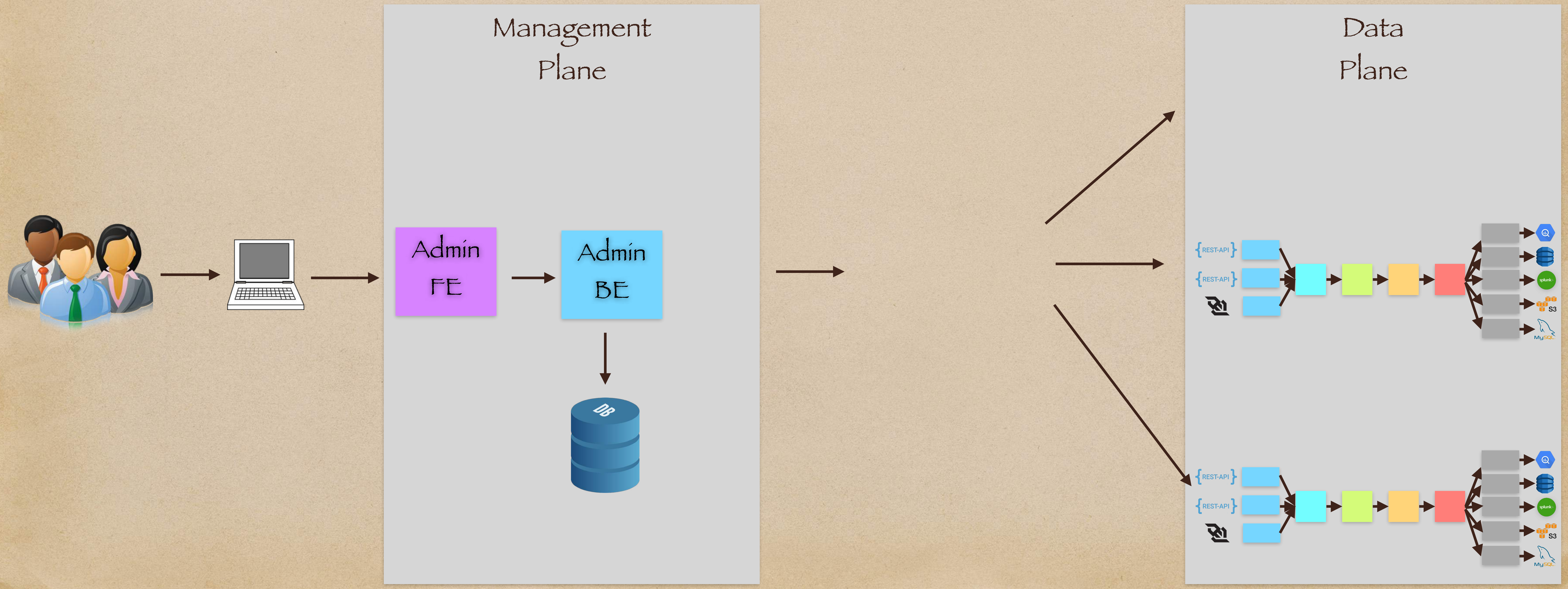
Building StaaS

Hence, we need a management plane to capture the intent of the users



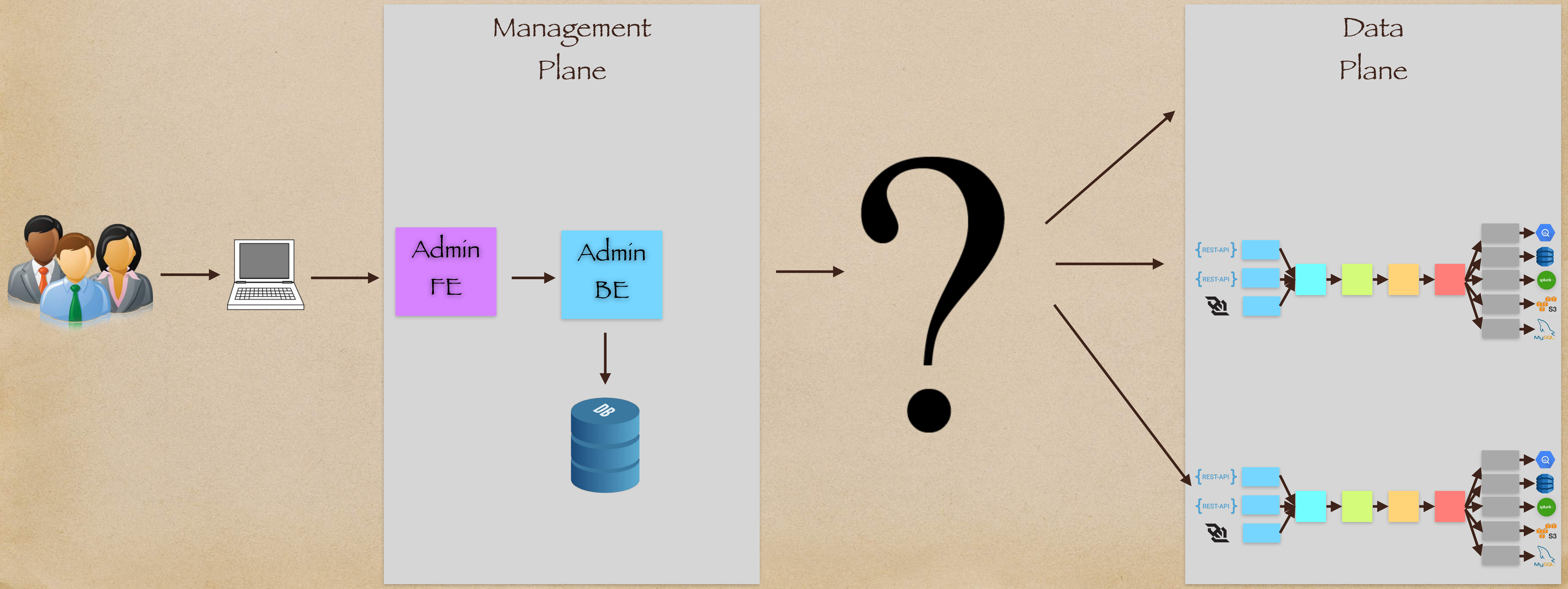
Building StaaS

We now have 2 planes: a Management Plane & a Data Plane



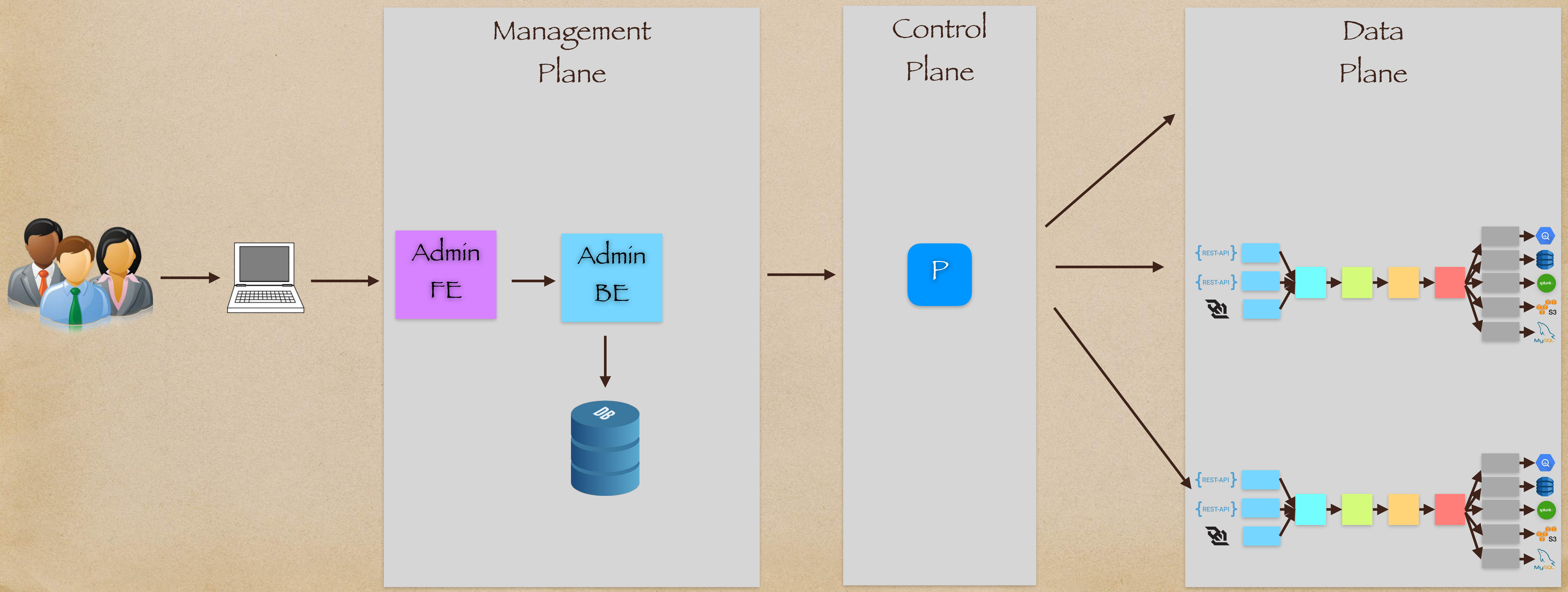
Building StaaS

Hence, we need at least 2 planes : Management & Data



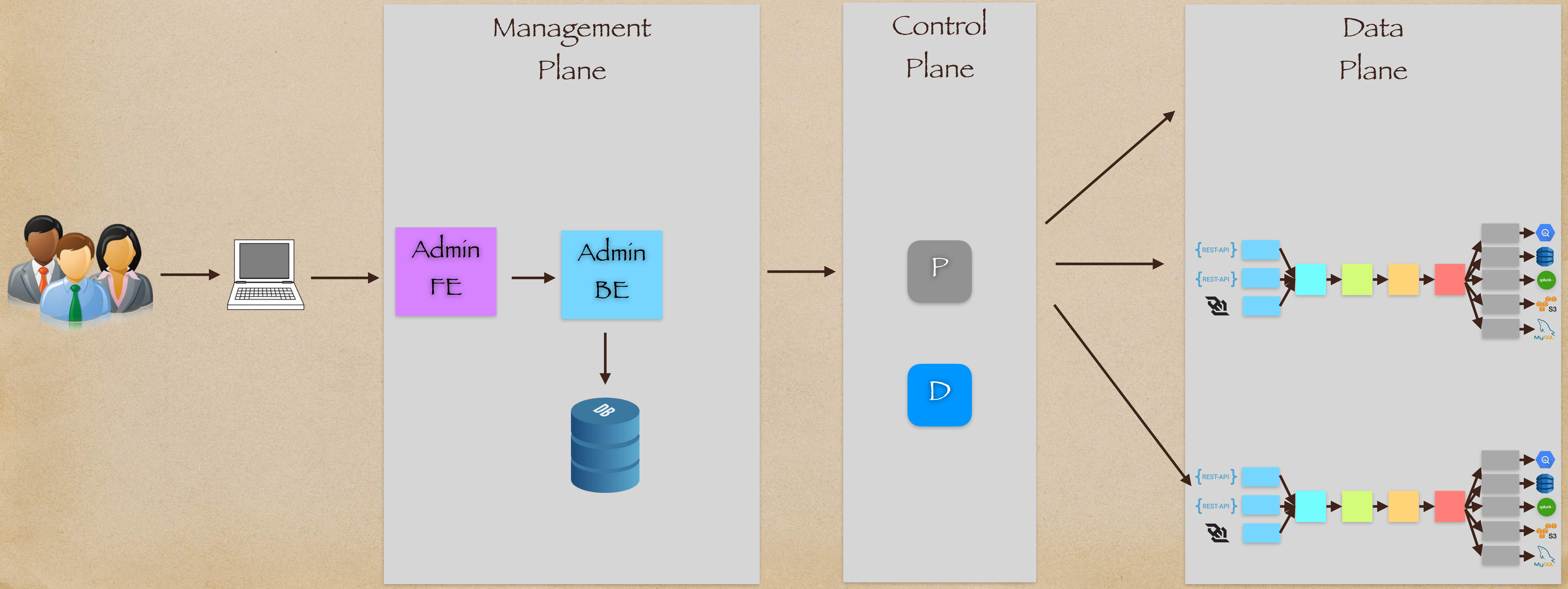
Building StaaS

We also need a Provisioner (P)



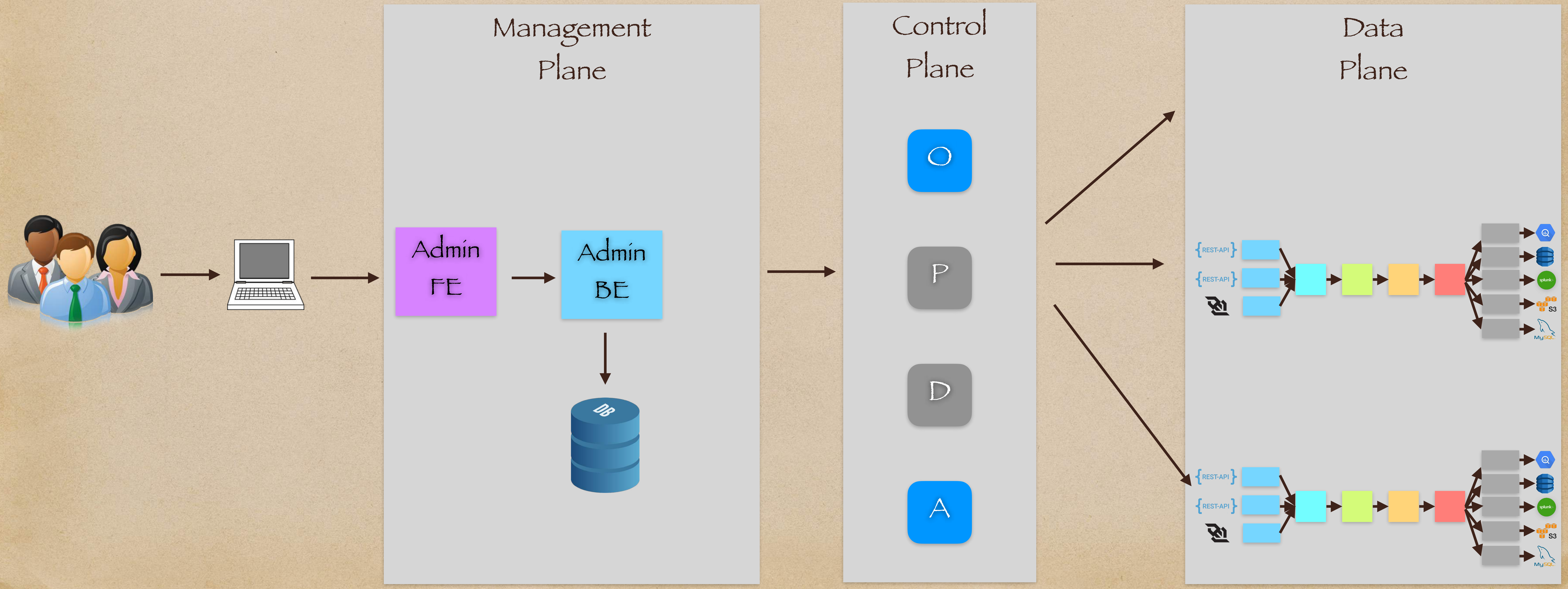
Building StaaS

We also need a Deployer(D)



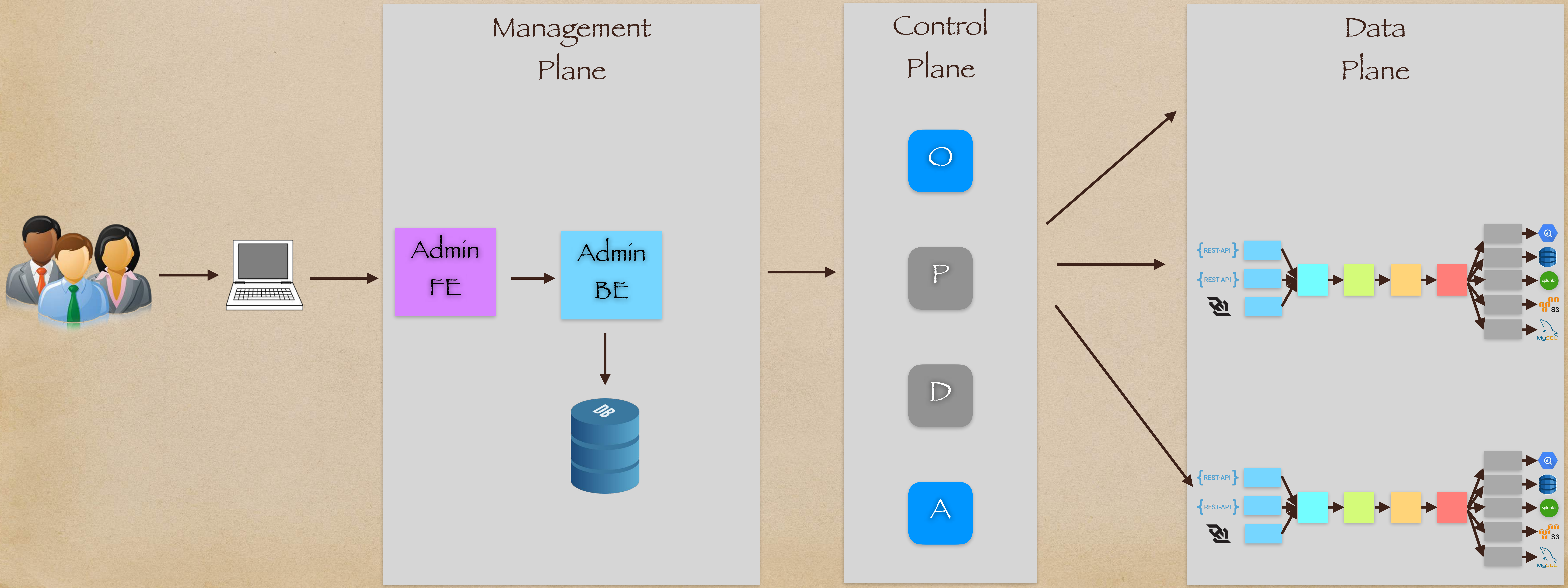
Building StaaS

Finally, we can add systems to promote health and stability: Observer(O) & Autoscaler (A)



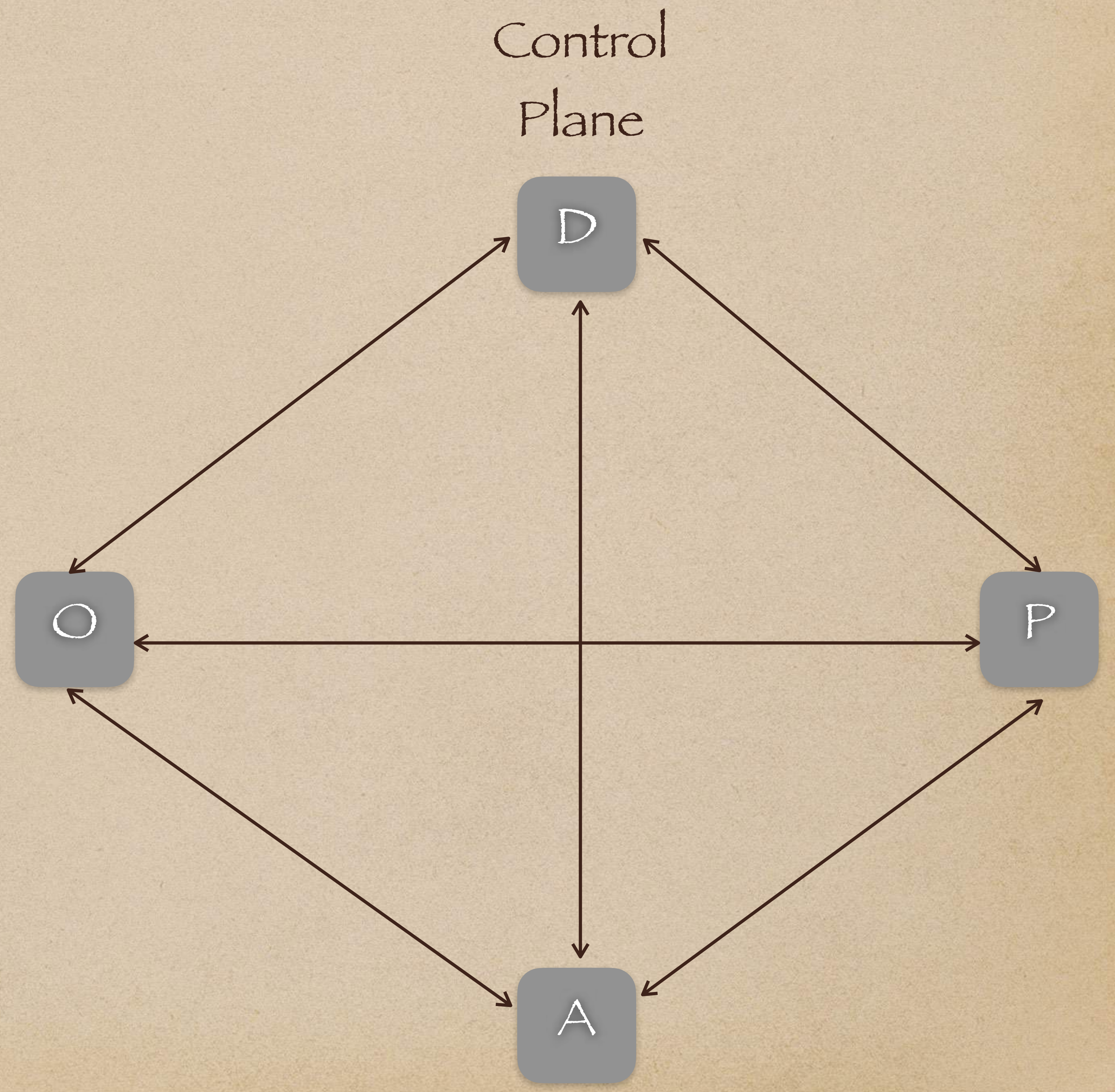
Building StaaS

Together these 4 services form the Control Plane



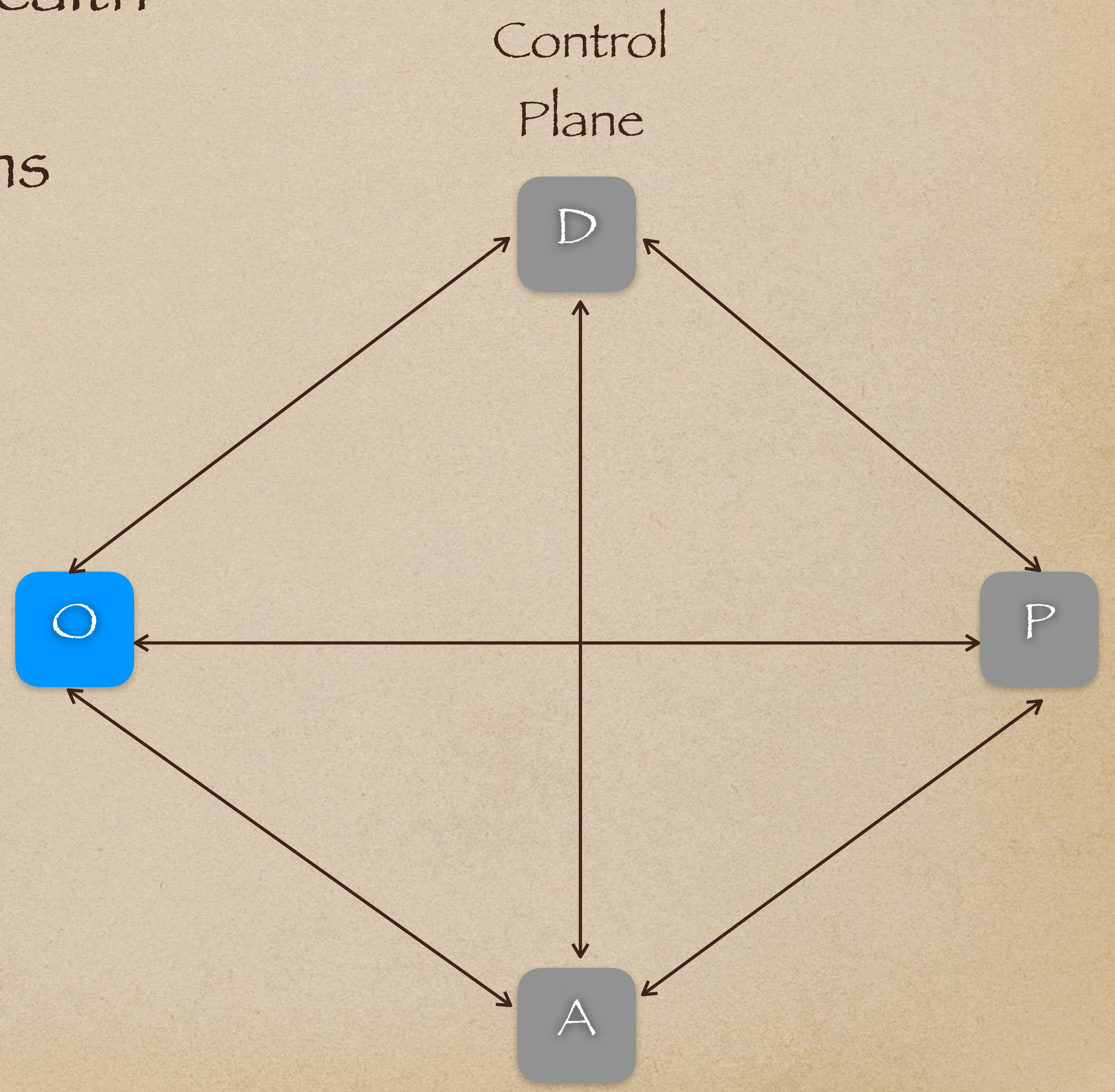
Building StaaS

The Control Plane Topology is a diamond-cross



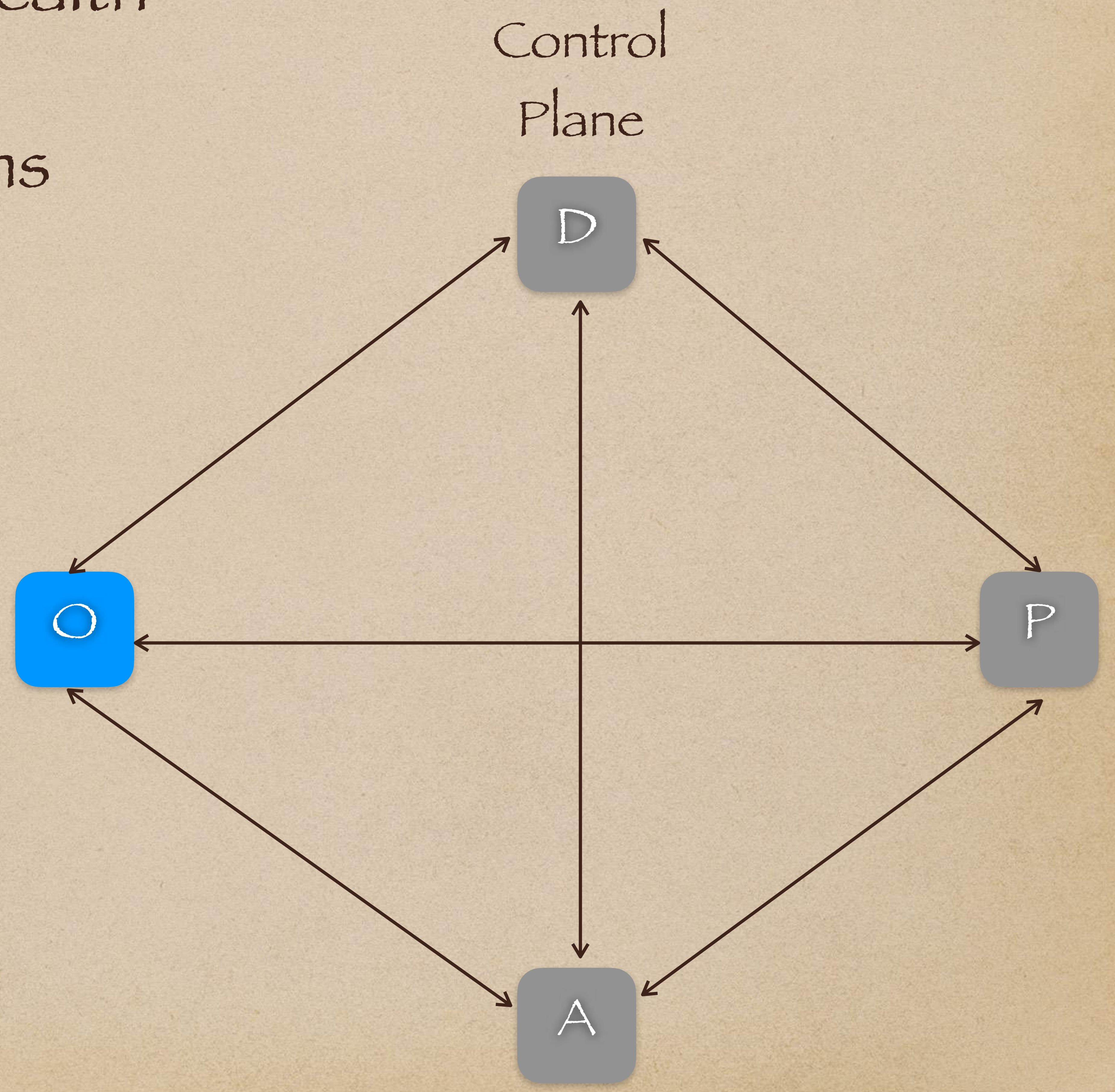
Building StaaS

- The observer(O) is the source of truth for system health
    - It is aware of D, P, and A activity & may quiet alarms during certain actions



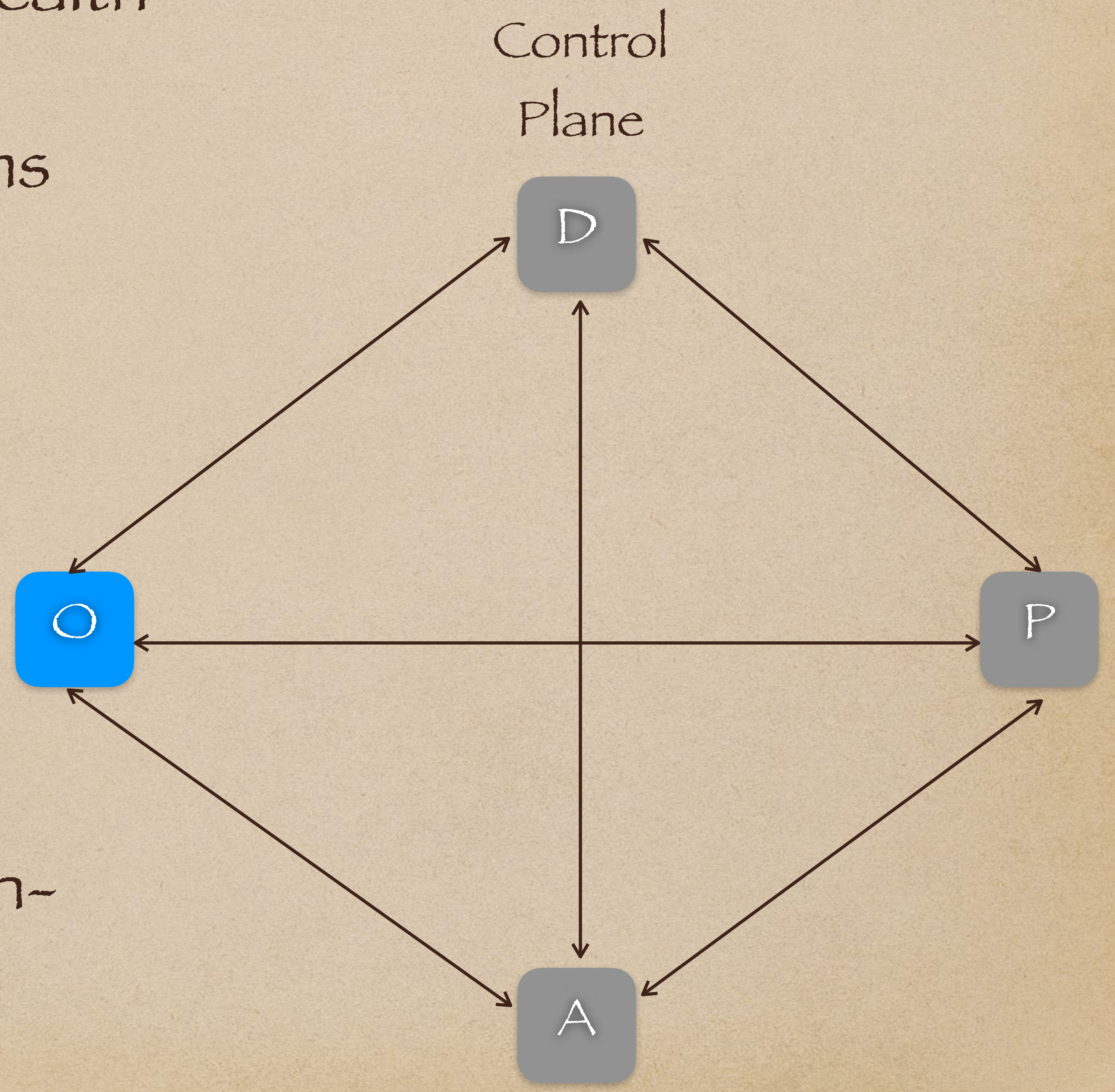
Building StaaS

- The observer(O) is the source of truth for system health
    - It is aware of D, P, and A activity & may quiet alarms during certain actions
    - It can collect and monitor more complex health metrics than lag and loss. For example, in ML pipelines, it can track scoring skew



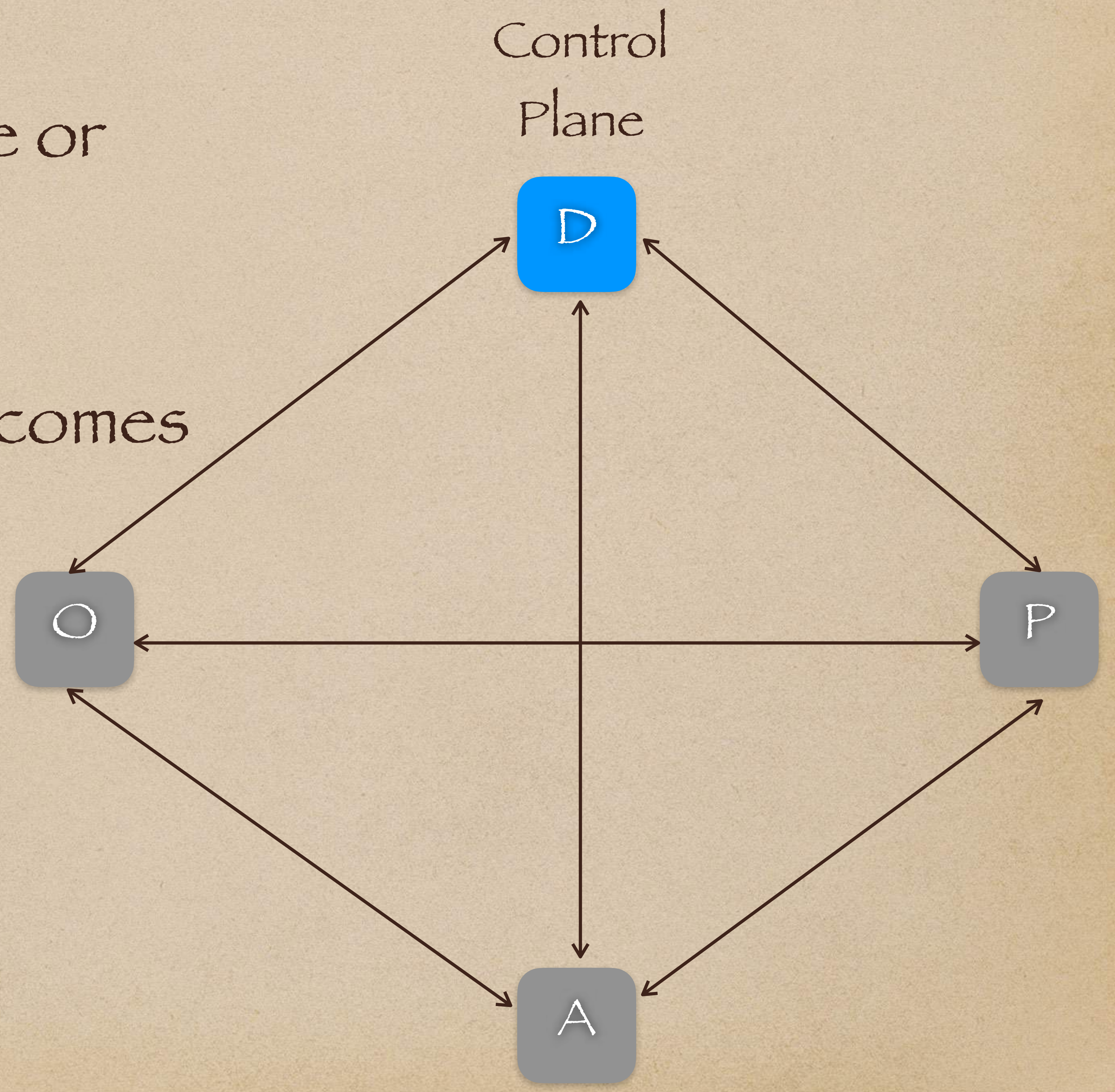
Building StaaS

- The observer(O) is the source of truth for system health
  - It is aware of D, P, and A activity & may quiet alarms during certain actions
  - It can collect and monitor more complex health metrics than lag and loss. For example, in ML pipelines, it can track scoring skew
  - The system can also detect common causes of non-recoverable failures & alert customers



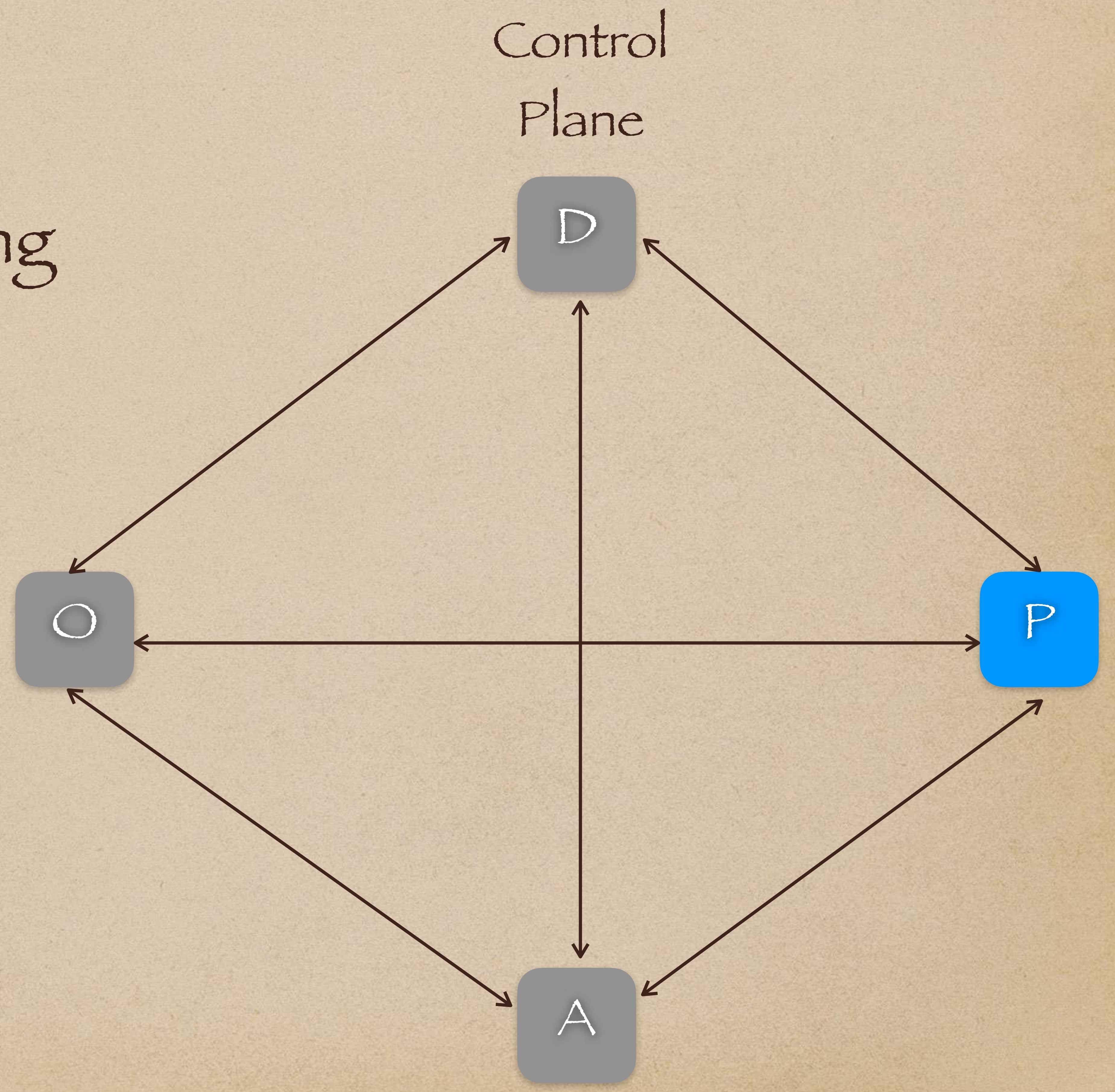
Building StaaS

- The deployer(D) deploys new code to the data plane
  - It can however not deploy if the system is unstable or autoscaling
  - It can also automatically roll back if the system becomes unstable due to deployment



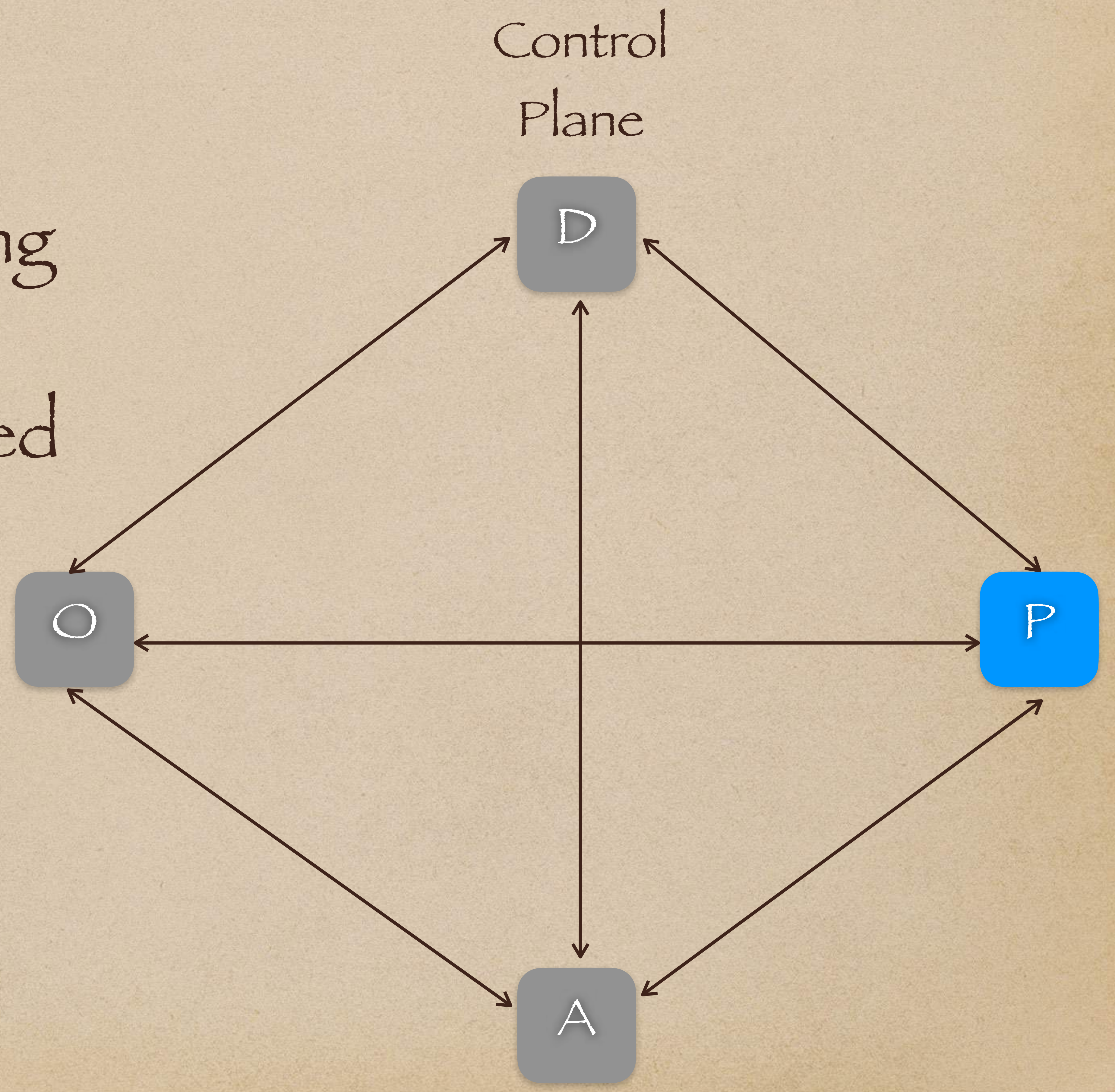
Building StaaS

- The provisioner(P) deploys customer data pipes to the system.
    - It can pause if the system is unstable or autoscaling



Building StaaS

- The provisioner(P) deploys customer data pipes to the system.
  - It can pause if the system is unstable or autoscaling
- The provisioner(P) can also control things like phased traffic ramp ups for new deployed pipelines




Conclusion

Conclusion

- We have built a Streams-as-a-Service system with many NFRs as first class citizens

- While we've covered many key elements, a few areas will be covered in future talks (e.g. Isolation, Containerization, Caching)

- Should you have questions, join me for Q&A and follow for more on (@r39132)

Thank You for your Time

And thanks to the many people who help build these systems with me..

- Vincent Chen
- Anisha Nainani
- Pramod Garre
- Harsh Bhimani
- Nirmalya Ghosh
- Yash Shah
- Aastha Sinha
- Esther Kent
- Dheeraj Rampali
- Deepak Chandramouli
- Prasanna Krishna
- Sandhu Santhakumar
- Maneesh CM & team at Active Lobby
- Shiju & the team at Xminds
- Bob Carlson
- Tony Gentille
- Josh Evans