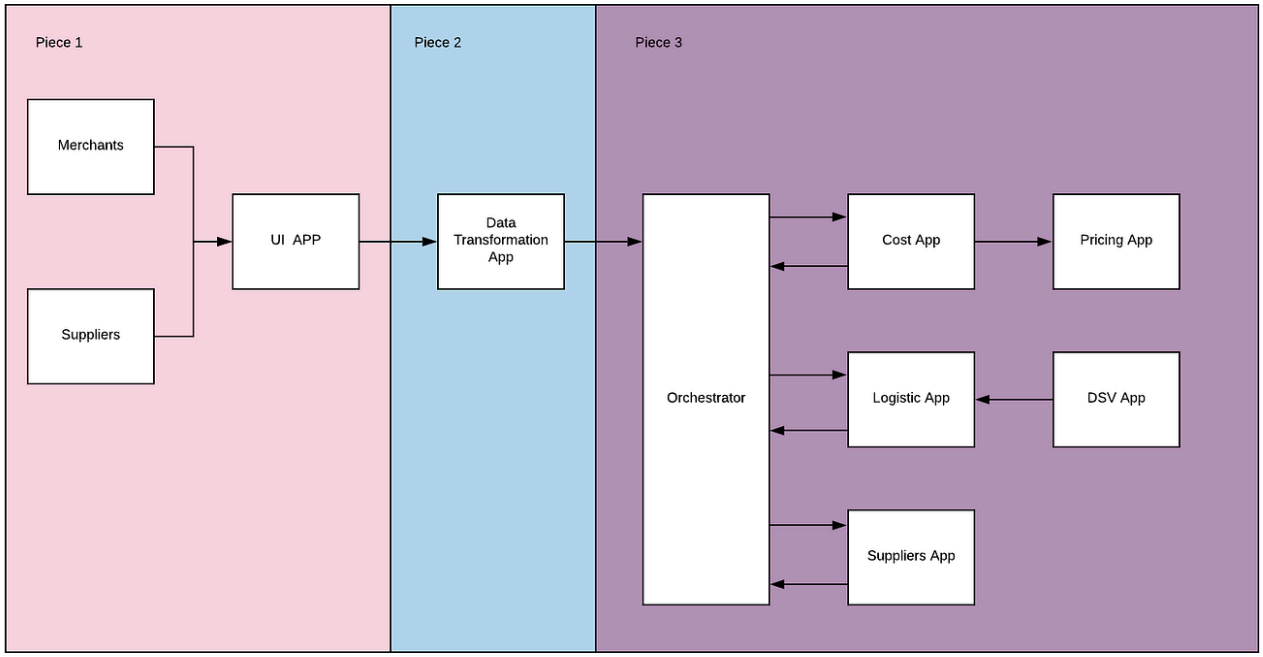
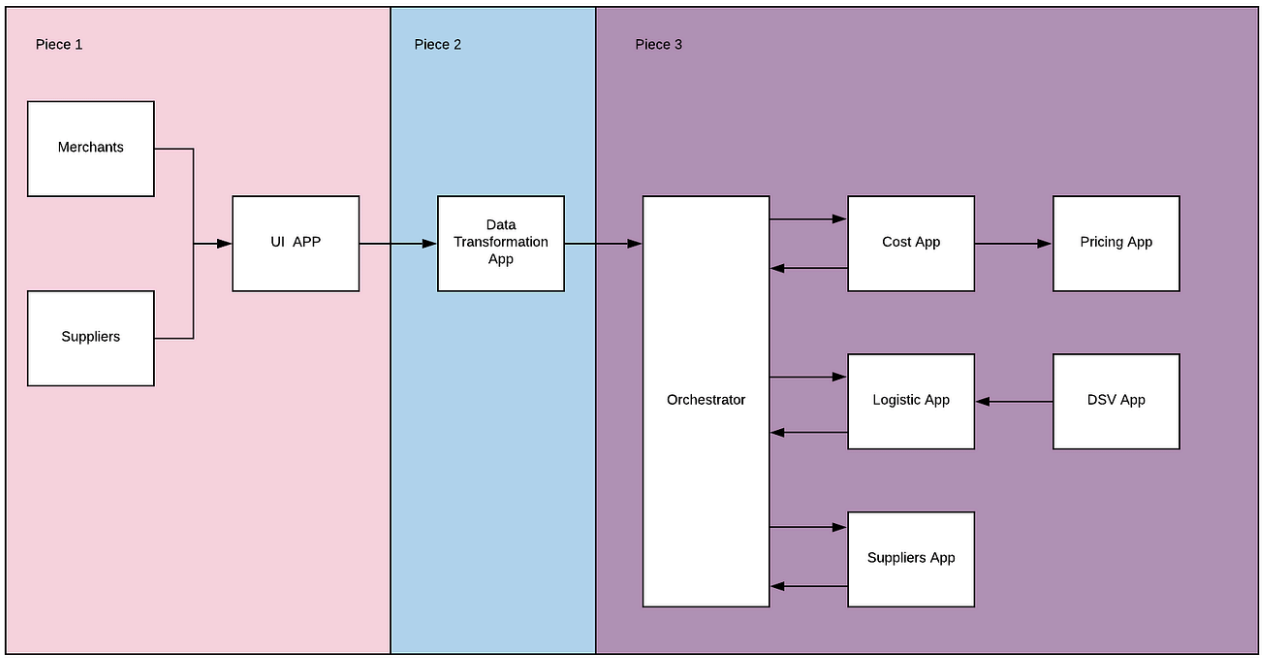
Kafka at Walmart

- C+L+S+p=P
- C: cost that walmart has to pay to buy each item from distributors
- L: logistic cost
- S: shipping cost
- p: walmart's profit per item
- P: price walmart represents to customers to purchase each item online
- As per equation, there are 5 micro-systems that use cost data: pricing, merchants, shippers, logistic and cost.
- Pricing uses cost data to update the price of an item.
- suppliers use cost to better represent supply chain dynamics.
- Logistic uses cost for logistical transport info
- merchants are originator of cost itself as they sell items on walmart.
- there are 3 pieces of software keeping data in sync:
- one retrives data
- one organizes data
- one distributes data
- retrieving consists of providing a user interface where people / programs can write the data to walmart.
- organizing data consists of transforming user input into data that is recognizable to walmart internal system.
- distributing : delivering each piece of data to each microsystem that needs it.





- orchestration piece happens through kafka/ HTTP call.
- Kafka is a msging service where one team writes a msg, and at a later time, another team responds indicating that they received msg.
- each time someone responds it is correlated with a particular key.
- and as each msg is distinct set of info, responder doesn't have to be present to hear the response.

Piece 1:

- Merchants write most of their information in excel documents in a grid like the following.

Item Gtin	Shipnode ID	Unit Cost	Allow other merchants to change cost	Cost Start Date	Cost End Date
062145556123	1234589	$39.88	Yes	4-11-19	4-21-19
012378912355	4522312	$21.24	No	4-08-19	4-11-19

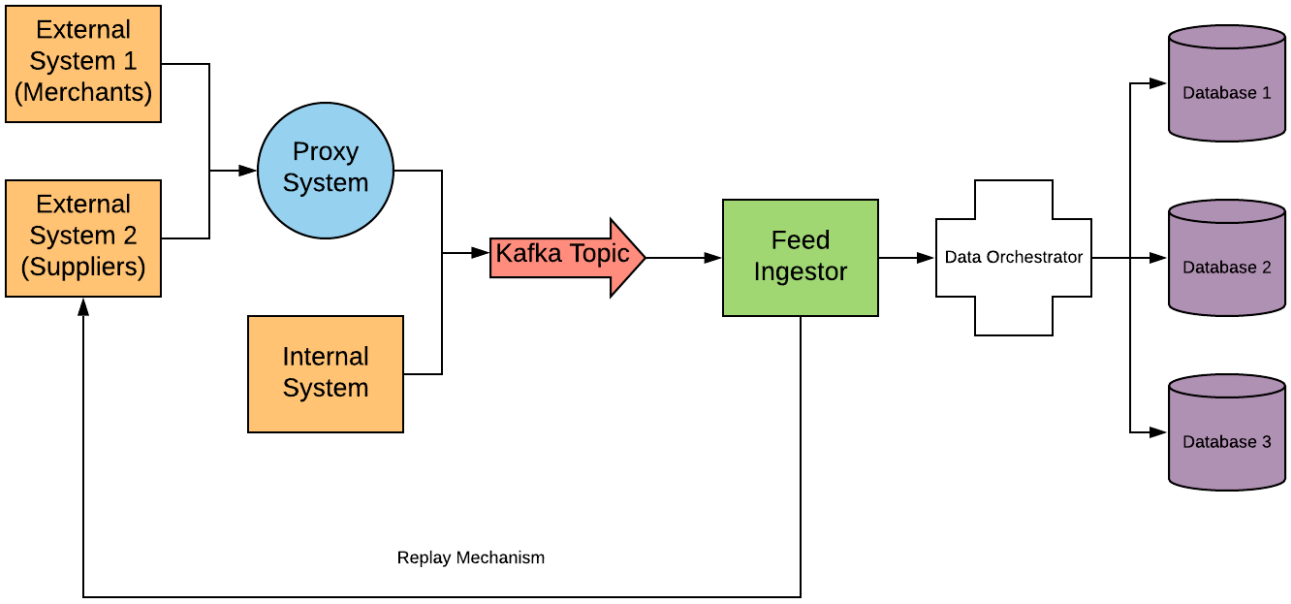
Piece 2:

- the data is broken into snippets for each team that needs it.
- eg. logistic team gets following snippet of data, derived from the grid above

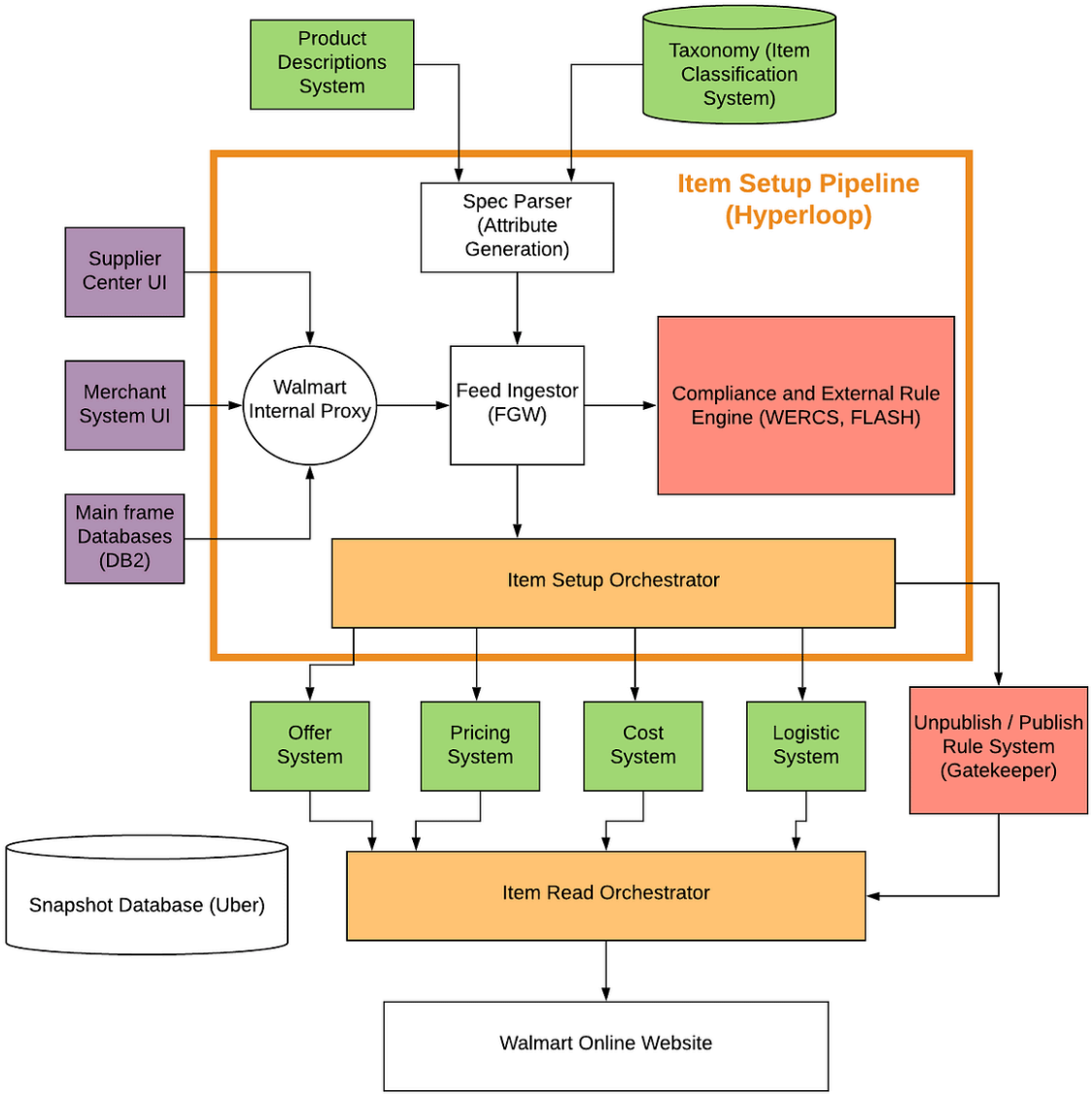
- walmart receives ~40 million cost updates per day and recalculates the prices of 80 million items per day.

- In total, data transformation system and orchestrator handle the delivery of 300 million attribute updates per day.

Performance Enhancements for the Item Setup Orchestrator

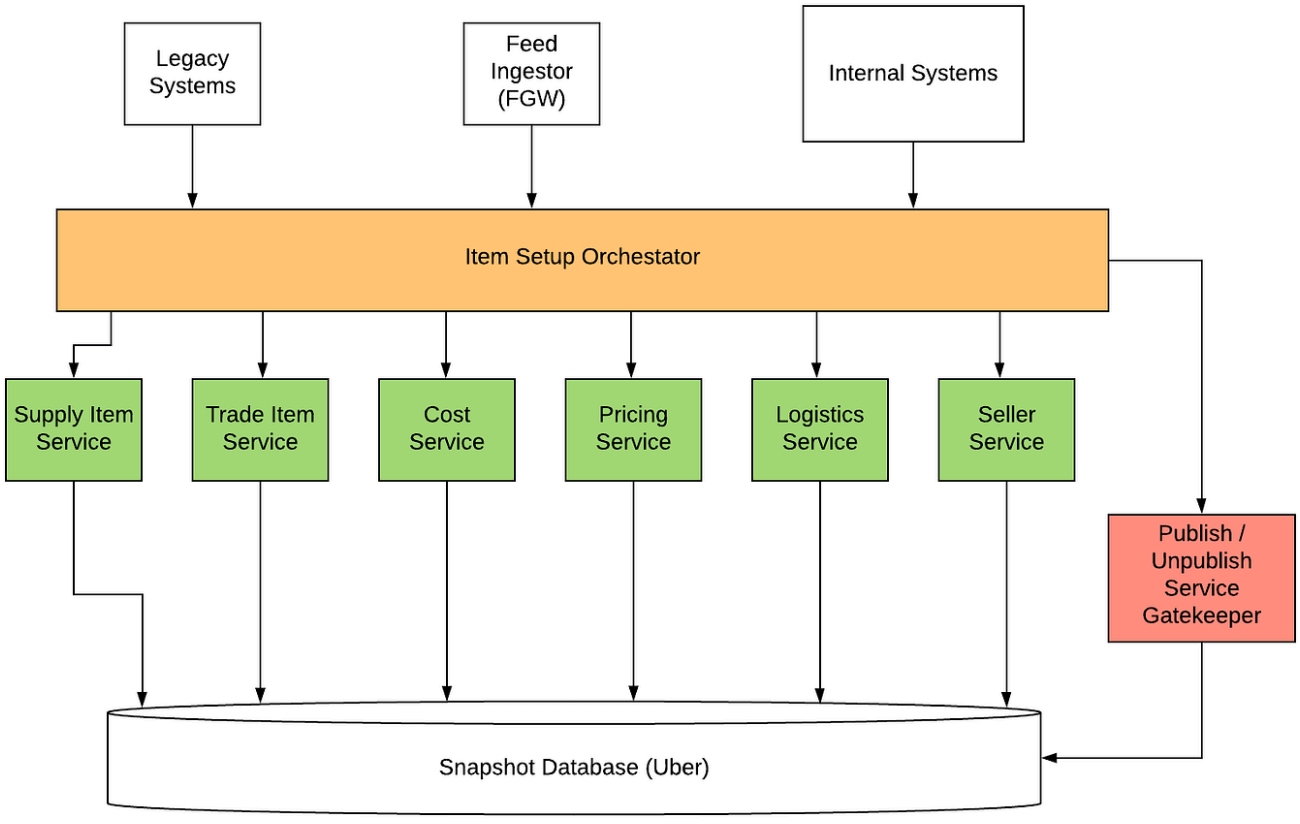


- Walmart's backend is where the data that builds the frontend is created.
- eg., Walmart has a DB that contains the images you see online.
- When you click the page on Walmart's website to buy Colgate toothpaste, the Item Read Orchestrator pulls images from Walmart's Product DB and pricing from Walmart's Pricing DB.



- Hyperloop consists of 3 systems:
- Spec parser
- Feed Ingestor
- Item setup orchestrator
- 1st step of selling an item online is storing all an item's data.
- eg. an item is made up of seller info, offer info, product description, pricing, logistic, shipping info.
- seller info. is made up of a series of attributes: name, seller description, seller id, item specific info. like its identifier etc.
- sometimes, storing data is more complicated bcz certain pieces of data depende on each other.
- eg. you can't change the seller name without changing seller id.
- next system is the data ingestor.
- The data ingestor is a state and transformation manager.

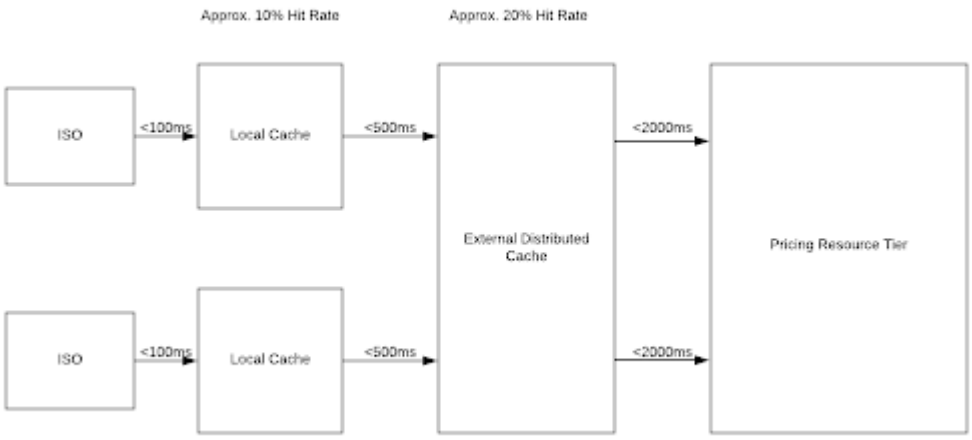
- First, it keeps track of the process of changing an item's attributes.
- This is especially important because the number of systems and grouping of information that exists across systems.
- eg. Logistics information is dependent on pricing and seller information. If seller information is changed, logistics and other downstream systems need to be looped through and updated as well. Additionally, if one system fails, the data ingestor needs to retry and reprocess the entire system.
- Another vital function the data ingestor performs is data transformations. Data from merchants is transformed from excel documents into Walmart data objects.
- final system is the item setup orchestrator.
- The item system orchestrator transports the data from the data ingestor to each of the systems responsible for the data.
- The item setup orchestrator takes price data from the data ingestor to the price system itself.
- In order to facilitate these final transactions, the item setup orchestrator is full of REST APIs and kafka tunnels.
- The orchestrator is especially useful when you want to chain messages and send information to multiple resource tiers at the same time.
- For example, cost information is sent to three systems the Supplier System, the Logistics System, and the Cost System.



Performance Improvements:

- made during the holidays.
- The item setup orchestrator was optimized because it takes the longest, touches the most systems, and has the most room to optimize.
- ISO was optimized in 3 ways.
- First, we split and dedicated machines for certain jobs.
- Certain jobs that ISO performs have higher priorities and shorter SLAs (service level agreements).
    - A service level agreement is the length of time a system has to finish processing a task.
- Thus using these different priorities, we can dedicate machines and resources.
- The second optimization was adding an internal and external distributed cache to minimize external reads.
- The third optimization was careful memory management using yourkit.
- ISO's most important functions are setting up items and setting up the prices for items.
- these jobs are extraordinarily critical and requires short SLAs.
- Prices should be reflected online as soon as merchants update them.
- likewise ISO has 2 hours to update every system with the merchant data it received.
- ISO dedicated 60 machines to process all the item updates it receives.
- Additionally, as soon as the resources are available to execute another item setup, the machine will start processing the next item setup.
- These item setup updates will be prioritized higher than the other jobs on those machines.
- ISO executes roughly 20 million item setups every day and has a peak volume of roughly 40 million item setups.
- Price updates were prioritized a little differently as they are updated directly through rest updates.
- Items setups occur throgh kafka topic.
- the consumers, ISO, picks the speed it takes items from the topic.
- A rest call is opposite in that rathen than ISO picking speed it progresses, caller of ISO determines speed for which item is updated.
- ISO handles rest calls in 2 ways using a load balancer.
- first, a load balancer distributes rest calls from users to 60 machines.
- load balancer sends msgs in a round robin style, which means each machine gets the same amount of traffic.
- after the msg gets past the load balancer, ISO rate limits and if traffic and speed of msgs are too fast for downstream services, ISO will throttle and return to rest call that service is unavailable.

- price ingester will retry msgs that the service is available.
- ISO receives 60 milliion price updates per day and has a peak volume of roughly 80 million msgs.
- second optimization, introducing local and external distributed cache.
- Caches are critical bcz not all operations in ISO are writes.
- significant amount of operations are reads.
- In read operation, rather than making a call to an external system, which takes time and may fail, we can read from a local/ external cache.
- A cache is memory optimization, where previous copies of data are stored.
    - eg, in Walmart, ISO looks up the price of a water bottle from the pricing resource tier and stores the price in the local cache. The next time ISO looks up the price of the same water bottle, ISO can retrieve the price from the local cache rather than having to look up the data from the pricing resource tier again.
- reading from the local cache is more than 20 times faster than reading from pricing resource tier.
- drawback from local cache is that it has limited capacity, and data must be looked up prior to search.
- local cache has the capacity to store roughly 20 minutes of data.
- The limited capacity is partially solved by the external distributed cache.
- When writing to the local cache, all the machines also asynchronously write a larger external database.
- The external database is shared by all the machines in the system.
- When looking for data in a cache, not only are you getting a single machine's history, you are getting all the machines' historical data.
- Between the two, we were able to reduce the number of read calls by 25%, speeding up entire item setup process by 5%.
- hit rate of all caches combined is close to 15%.



- Third optimization profiling and looking for bottlenecks.
- Yourkit is a java profiler that looks for memory leaks, bottlenecks and other performance optimizations.
- It is useful bcz we can examine each thread in application, see what java classes are taking longest and using most memory.
- when profiling our application, we found 2 bottlenecks: logging and JSON serialization.
- bcz so many msgs stream through our system, we constantly converting msgs from string to java objects, and writing those msgs into a log to debug.
- The first version of log4j has many known deadlocks, and the next generation of log4j, log4j2 is 10 times faster than the old version.
- The second bottleneck was changing the json serializer. The item setup team chose the JSON serializer JsonIter. JsonIter is significantly faster than the previous JSON serializer and especially optimized to be multithreaded.

Reliably Processing Trillions of Kafka Messages Per Day

- Walmart deploys Kafka with 25k+ kafka consumers, across multiple clouds (public and privatre).
- use cases including: data movement, event driven microservices and streaming analytics.
- High consumer lag due to Kafka consumer rebalancing is the most common challenge in operationalizing Kafka consumers at scale.

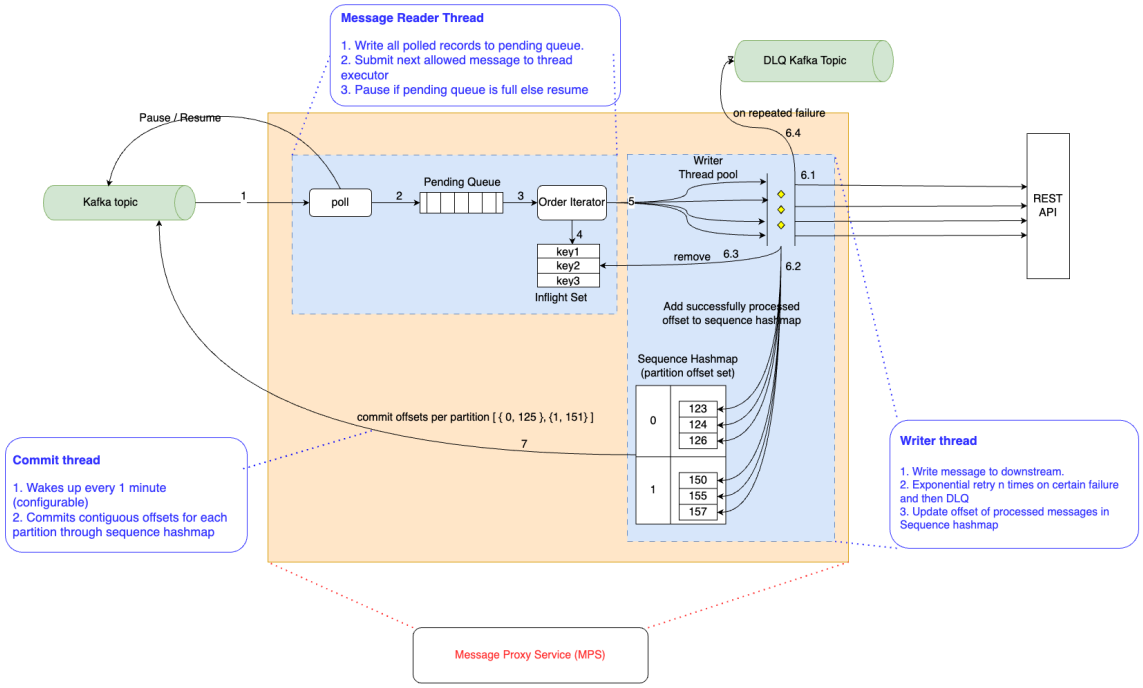
Challanges:

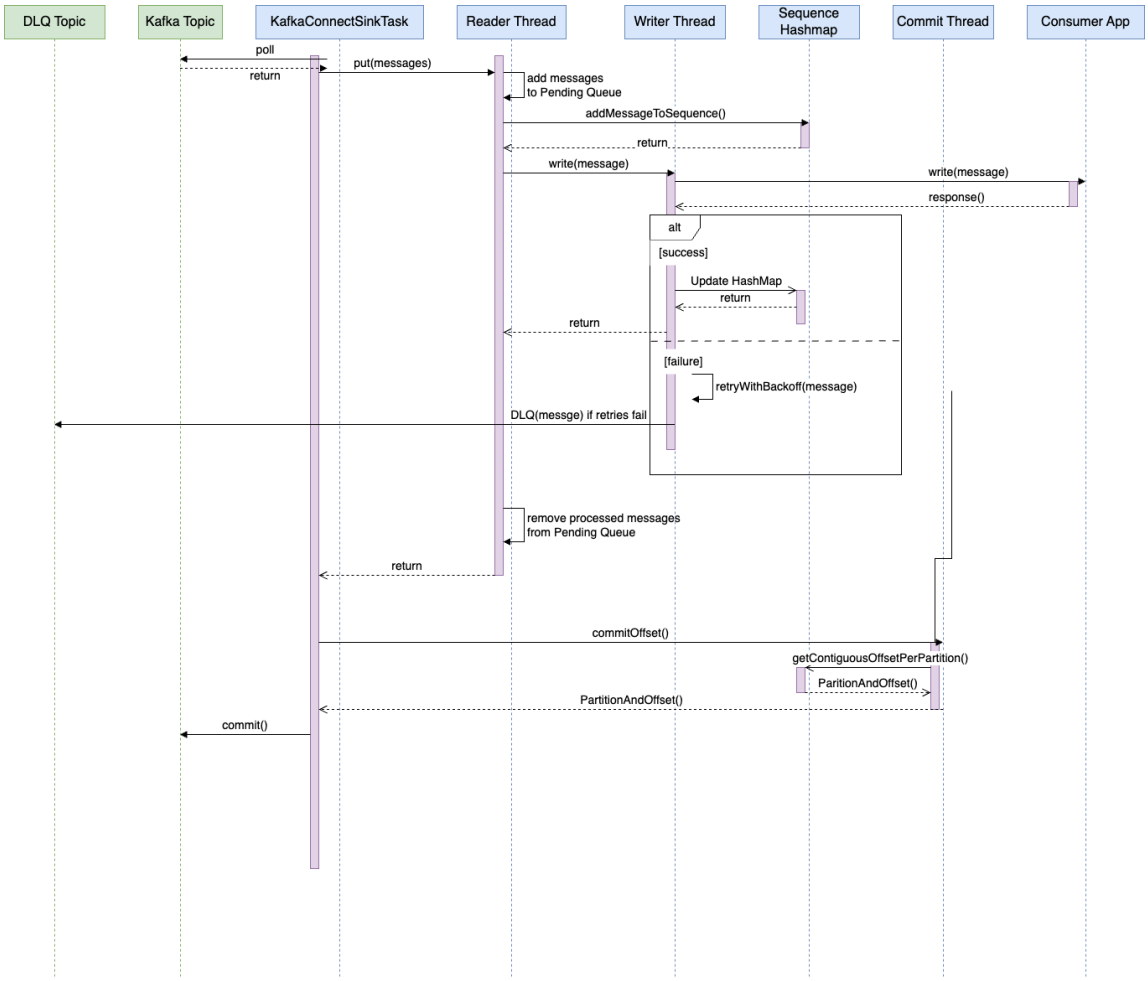
- ### Consumer rebalancing:
  - Kafka rebalancing is the process by which Kafka redistributes partitions across consumers to ensure that each consumer is processing a roughly equal number of partitions.
  - Consumer rebalancing can thus be triggered by multiple causes, including:
  - A consumer pod leaving a consumer group: This can be caused by K8s deployments or rolling-restarts or automatic/manual scale-ins.
  - A consumer pod entering a consumer group: This can be caused by K8s deployments or rolling-restarts or automatic/manual scale-outs.
  - Kafka broker believing that a consumer has failed (e.g., if the broker has not received a heartbeat from a consumer within [session.timeout.ms]: This will be triggered if the JVM exits or has a long stop-the-world garbage collection pause.

- Kafka broker believing a consumer is stuck (e.g., if the consumer takes longer than [max.poll.interval.ms] to poll for the next batch of records to consume): This will be triggered if processing of the previously polled records exceeds this interval.
- Given near-real time nature of commerece today, many kafka use-cases have tight delivery SLAs - these applications suffered from constant lag alarms due to frequent and unpredictable rebalances in production.

- **Poison pill**
  - Head-of-line (HOL) blocking is a performance-limiting phenomenon that can occur in networking and messaging system.
  - it happens if a kafka consumer encounters a msg that will never successfully be processed.
  - if msg processing results in an uncaught exception thrown to kafka consumer thread, consumer will re-consumer the same msg batch on next-poll of the broker - predictably, the same batch containing the same "poison pill" msg will result in same exception.
  - this loop will continue indefinitely until a code fix is deployed to kafka consumer application skipping problematic msg or correctly processing it or problematic msg is skipped by changing consumer offset.
  - this poison-pill problem is yet another problem associated with in-order processing of partitioned data streams.
  - kafka doesn't handle poison pill msgs automatically.
- **Cost:**
  - strong coupling between partitions in a topic and consumer threads that read from them.
  - maximum number of consumers of a topic can't exceed number of partitions in that topic.
  - If consumers are unable to keep up (lag) with topic flow, adding more consumer will require to increase the partitions also.
  - there is a general rule on the number of partitions you can add to a broker before needing to vertically scale the broker nodes up to next biggest size (4000 partitions/broker).
  - When there are thousands of pipelines, increasing the number of partitions becomes operationally onerous as it requires coordination among producers, consumers, and platform teams and imposes a small window of downtime.
- **Solution:**
  - Message Proxy Service (MPS) is one of the path that is available.
  - MPS decouples Kafka consumption from the constraint of partitions by proxying messages over HTTP to REST endpoints behind which consumers now wait.

- Via the MPS approach, Kafka consumption no longer suffers from rebalancing, while also allowing for greater throughput with a lower number of partitions.
- **Design:**
  - MPS Kafka consumer group consists of 2 independent thread groups: Kafka message_reader thread (a group of 1 thread) and message_processing_writer threads.
  - these groups are separated by a standard buffering pattern (pendingQueue).
  - reader threads writes to a bounded buffer (during the poll) and writer threads read from this buffer.
  - bounded buffer provides control on speed of reader and writer threads.
  - message_reader thread will pause consumer when pendingQueue reaches a max buffer size.
  - this separation of reader and writer threads makes reader thread incredibly lightweight and doesn't trigger a rebalance operation by exceeding max.poll.interval.ms.
  - Now, writer threads can take time needed to process messages.





- Reader Thread: make progress through inbound topics, applying back-pressure when pendingQueue is full.
- Order Iterator:
  - guarantees that keyed msgs are processed in order. It iterates through all msgs in pendingQueue and leaves msgs (temporarily skips) if there is already a msg with the same key in flight.
  - skipped msgs will be processed in subsequent poll calls once earlier msgs with same key are processed.
  - by ensuring that no more than 1 msg per key is in flight, MPS guarantees in-order delivery by key.
- Writer Thread:
  - part of a pool that provides greater throughput via parallelism.
  - it's job is to reliably write data to REST endpoints and DLQ'ing msgs if either retries are exhausted or non retry-able HTTP response codes are received.
- Dead Letter Queue:
  - DLQ topic can be created in every kafka cluster.
  - message_processing_writer thread intially retries msgs a fixed no. of times with exponential back off.

        - if this fails, msg is put in DLQ topic.
        - Application can handle these msgs later or discard them.
        - msgs can be placed in this queue when consumer service has a outage (timeouts) or if consumer service encounters a poinson pill (500 HTTP response)
    - Consumer service:
        - is stateless REST service for applications to process msgs.
        - this service contains business logic that was part of processing originally available in kafka consumer application.
        - with this new model, kafka consumption (MPS) can be separated from msg processing (Consumer service).
    - Kafka Offset Commit Thread:
        - kafka offset committing is implemented as a separate thread (offset_commit thread).
        - this thread wakes up at regular intervals (1 minute) and commits latest contiguous offsets which are successfully processed by writer threads.
        -

Type	Value	Description
Transport	HTTP/REST	
HTTP Accept	Accept=application/json Accept-Charset=ISO-8859-1	
HTTP Headers	Key, Topic, Offset, Partition, All Kafka headers	
Body	payload: {Kafka message value encoded in ISO-8859-1}	
API	POST/process	
Return codes	2XX	Success
	4XX and 5XX	Proxy keeps retrying these return codes.
	6XX	Retry n' times (configurable) and then DLQ.
	7XX	DLQ. This indicates message format is BAD or failed business validation stage in the application.
	Connect & Read Timeouts	Auto DLQ. We see poison pill messages continuously timeout. MPS logic, based on the rate and duration of timeouts per message detects it to be a poison pill and automatically DLQs (Dead Letter Queue) it and alerts the consumer application.

Conclusion:

- MPS has eliminated rebalances due to slowness in downstream systems as it guaratees reader thread will put all msgs in pendingQueue within mac.poll.interval.ms.
- the only rebalances we see are due to k8s POD restarts and exceedingly rare network slowness between kafka cluster and MPS.
- but with small consumer groups, the duration of these cycles is neglibile and don't exceed processing SLAs.
- MPS service and kafka cluster should be hosted in the same cloud/ region to reduce network related ussue between them.
- Cooperative handling of poison pills with applications detecting them and notifying MPS through the return codes 600 & 700 works as planned.

- the cost benifits of this solution are realized in 2 areas:
- stateless consumer services can scale quickly in k8s env, and don't have to be scaled upfront for the holidays or campaign events.
- secondly, kafka cluster sizes are no longer depedent on partition sizes, they can be scaled for throughput with about 5-10 MB per partition.
- sudden spikes in traffic don't need to scale kafka partitions anymore as stateless consumer services are easily auto scaled in k8s env to handle msg bursts.