Event-Driven Architecture

...

Event-Driven Architecture

- An event-driven architecture uses **events** to trigger and communicate between decoupled services
- Common in modern applications built with microservices.
- An event is a change in state, or an update, like an item being placed in a shopping cart on an e-commerce website.
- Events can either carry the state (the item purchased, its price, and a delivery address) or events can be identifiers (a notification that an order was shipped).

Event-Driven Architecture

- Event-driven architectures have three key components:
  - event producers: publish event to router
  - event routers: filter and push to consumers
  - event consumers: receive event and take actions accordingly
- Producer services and consumer services are decoupled, which allows them to be scaled, updated, and deployed independently.

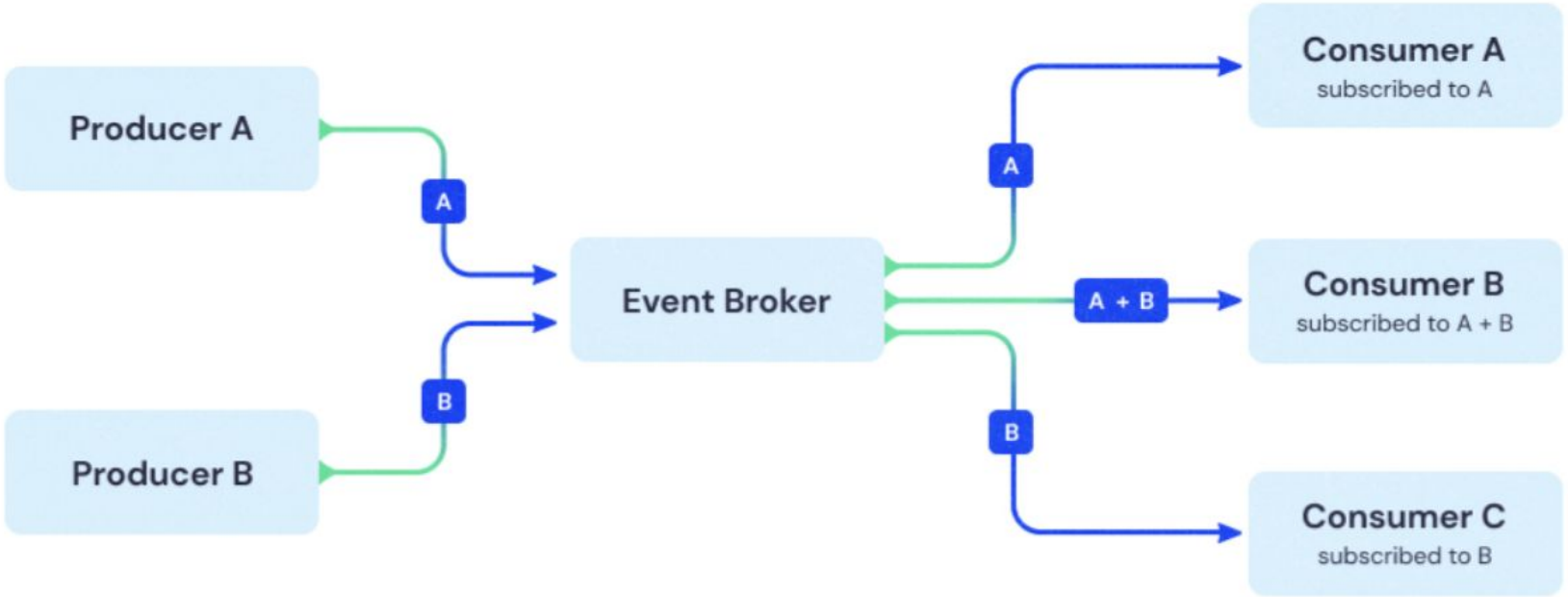
What is an event?

- An event is defined as a change in the state of the system.
- An event is anything that generates a message by being created, published, detected, or consumed.
    - The event is distinct from the message because
        - the event is happening
        - the message is the notice that conveys that the event took place.
- In an event-driven architecture, an event will almost always trigger one or more actions or processes in reaction to its occurrence.

What is an event?

- An example of an event could be requesting a password reset.
    - One possible reaction is to simply log the incident for future reference.
    - Another reaction could be to send an email confirmation to the user requesting to reset their password.

Event-Driven Architecture: Pub/Sub



Publish/Subscribe

- Convey messages from an event sourcing system to decoupled target systems.
- Publish messages to a shared topic.
    - One or more consumers subscribe to that topic to listen for relevant events.
- The publisher does not need to know anything about the subscribers under the pub/sub paradigm.
    - Subscribers/Consumers keep track of which messages they have received and are in charge of self-managing load levels and scaling.

Something can be both a publisher and a subscriber!

- Example: News scraping pipeline

Event-Driven Architecture

- Coordination done by cloud provider
  - You do not need to write custom code to poll, filter, and route events
  - The event router will automatically filter and push events to consumers.
  - The router also removes the need for heavy coordination between producer and consumer services
- You just define the producer, consumer, and router
  - Event-driven architectures are push-based, so everything happens on-demand as the event presents itself in the router.

Event-Driven Architecture

- Scale and fail independently
  - Services are only aware of the event router, not each other.
  - This means that your services are interoperable, but if one service has a failure, the rest will keep running.
  - The event router acts as an elastic buffer that will accommodate surges in workloads.

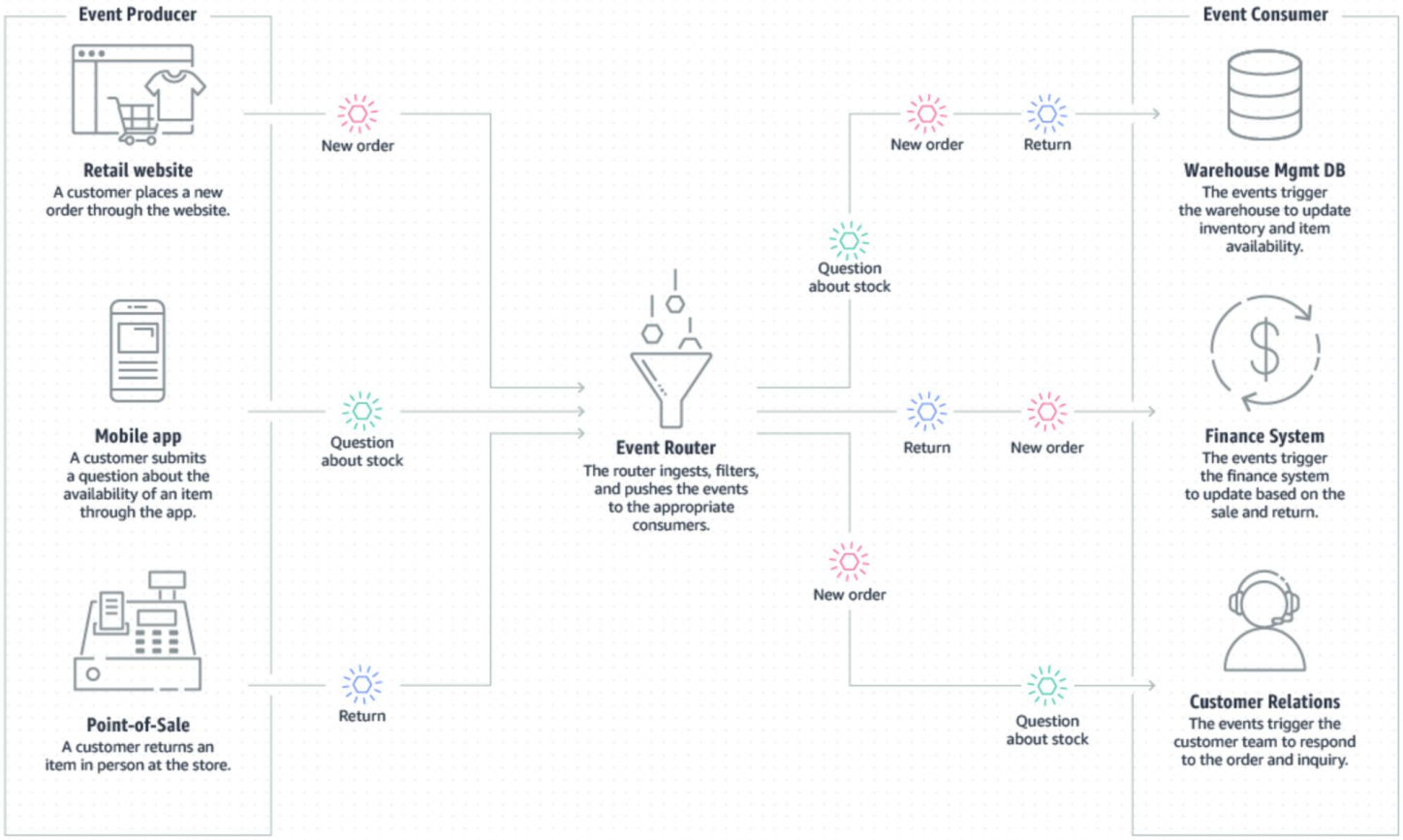
Deferred Execution

- If you're used to REST-based APIs, the concept of deferred execution is tricky.
- When you publish an event, **you don't wait for a response**.
- The event router "holds" (persists) the event until all interested consumers accept/receive it, which may be some time later.
- Acting on the original event may then cause other events to be emitted, which are similarly persisted.
- So event-driven architecture leads to cascades of events, **which are temporally and functionally independent of each other, but caused in a sequence.**
    - All we know is that event A will at some point cause something to happen.
    - The execution of the logic consuming event A isn't necessarily instant – its execution is deferred.

Publish/Subscribe

- In event-streaming systems, calls are usually always asynchronous; they deliver events without waiting for a response.
- Asynchronous eventing provides more scalability choices for both producers and consumers.
- However, if you want FIFO message-order guarantees, this asynchronous style can cause problems.

Event-Driven Architecture



Publish/Subscribe: Pros

- Adding or removing subscribers to a topic is a matter of configuration.
  - No complex programming is required.
  - Very scalable and flexible
- Pub-Sub activity is asynchronous
  - There is little risk of performance degradation due to a process getting caught in a long-running data exchange interaction.

Publish/Subscribe: Cons

- Testing can be a challenge.
  - Because interactions are asynchronous, testing is not a matter of making a request and then analyzing the result.
  - A message must be sent into the system, and then the test needs to observe the behavior of the process(s) under test to see when and how it handles the message.
  - And, if the process under test requires consuming many messages from a topic over time, the testing regimens can become more difficult to manage.
- An unexpected surge in message emission can cause bottlenecks in the network
  - Can have unexpected results for the consuming message broker.
- Requires a well-defined policy for message formatting and message exchange
  - Otherwise, message consumption can become mangled and error-prone.

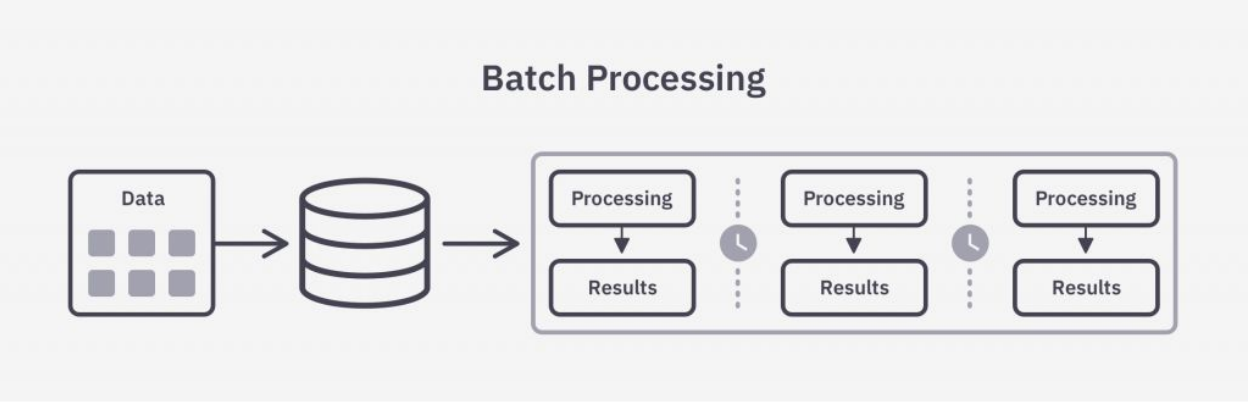
Pub/Sub: Dead Letter Queues

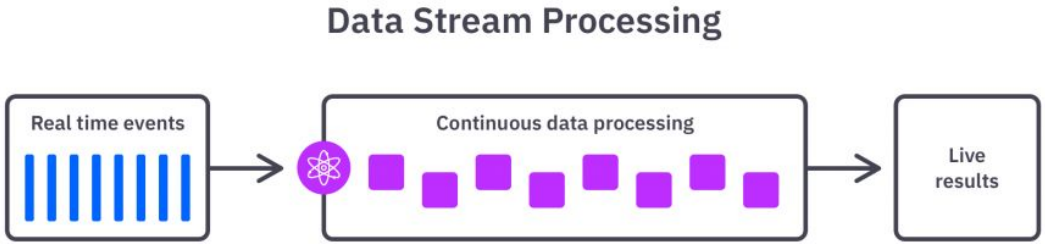
- A dead-letter queue lets you set aside and isolate messages that can't be processed correctly to determine why their processing didn't succeed.
  - Erroneous message content
    - Hardware, software, and network conditions might corrupt the sent data.
    - For example, hardware interference slightly changes some of the information during transmission.
    - The unexpected data corruption could cause the receiver to reject or ignore the message.
  - Changes in the receiver’s system
    - Receiving software has gone through changes that the sender is not aware of.
    - For example, you could attempt to update a customer's information by sending a message for CUST_ID_005. The receiver could fail to process the incoming message because it’s removed the customer from the system's database.

Pub/Sub Cloud

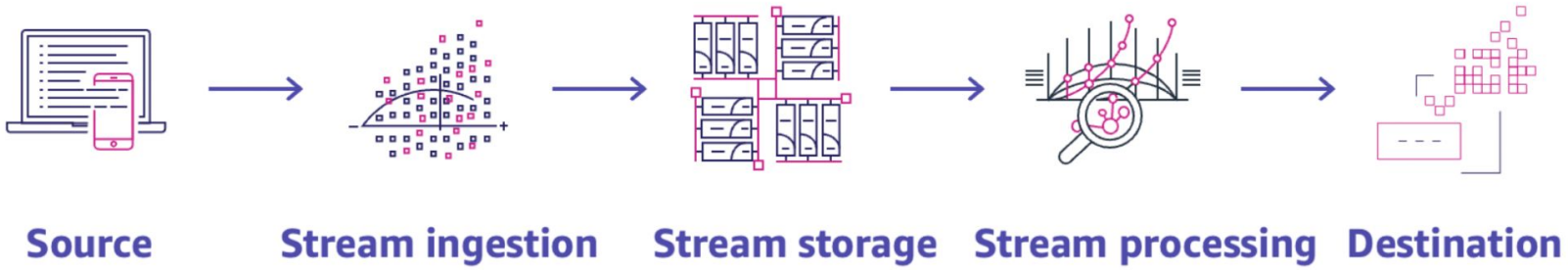
- Google
  - https://cloud.google.com/functions/docs/calling/pubsub
- AWS
  - https://aws.amazon.com/pub-sub-messaging/
- Azure
  - https://azure.microsoft.com/en-us/products/web-pubsub

Events: Batch vs Stream





Streaming Architecture



Streaming Architecture

- Streaming data refers to data that is continuously generated, usually in high volumes and at high velocity.
- A streaming data source would typically consist of continuous timestamped logs that record events as they happen
    - such as a user clicking on a link in a web page, or a sensor reporting the current temperature.
- Streaming architectures must account for the unique characteristics of data streams:
    - tend to generate massive amounts of data (terabytes to petabytes)
    - semi-structured and require significant pre-processing and ETL to become useful.

Streaming Architecture

- A streaming data architecture is a framework of software components built to ingest and process large volumes of streaming data from multiple sources.
- While traditional data solutions focused on writing and reading data in batches, a streaming data architecture:
  - consumes data immediately as it is generated
  - persists it to storage
  - may include various additional components per use case – such as tools for real-time processing, data manipulation, and analytics.

Streaming Architecture: Components

- Source: data sources:
  - Sensors
  - social media
  - IoT devices
  - log files generated by using your web and mobile applications
  - mobile devices
- Generates semi-structured and unstructured data as continuous streams at high velocity.

Streaming Architecture: Components

- Stream storage
  - The stream storage layer is responsible for providing scalable and cost-effective components to store streaming data.
  - The streaming data can be stored in the order it was received for a set duration of time, and can be replayed indefinitely during that time.

Streaming Architecture: Components

- Stream ingestion
  - The stream ingestion layer is responsible for ingesting data into the stream storage layer.
  - It provides the ability to collect data from tens of thousands of data sources and ingest in near real-time.

Streaming Architecture: Components

- Stream processing
  - The stream processing layer is responsible for transforming data into a consumable state
    - Data validation
    - Data cleanup
    - Data normalization
    - Data transformation
    - Data enrichment.
  - The streaming records are read in the order they are produced, allowing for real-time analytics, building event driven applications, or streaming ETL.

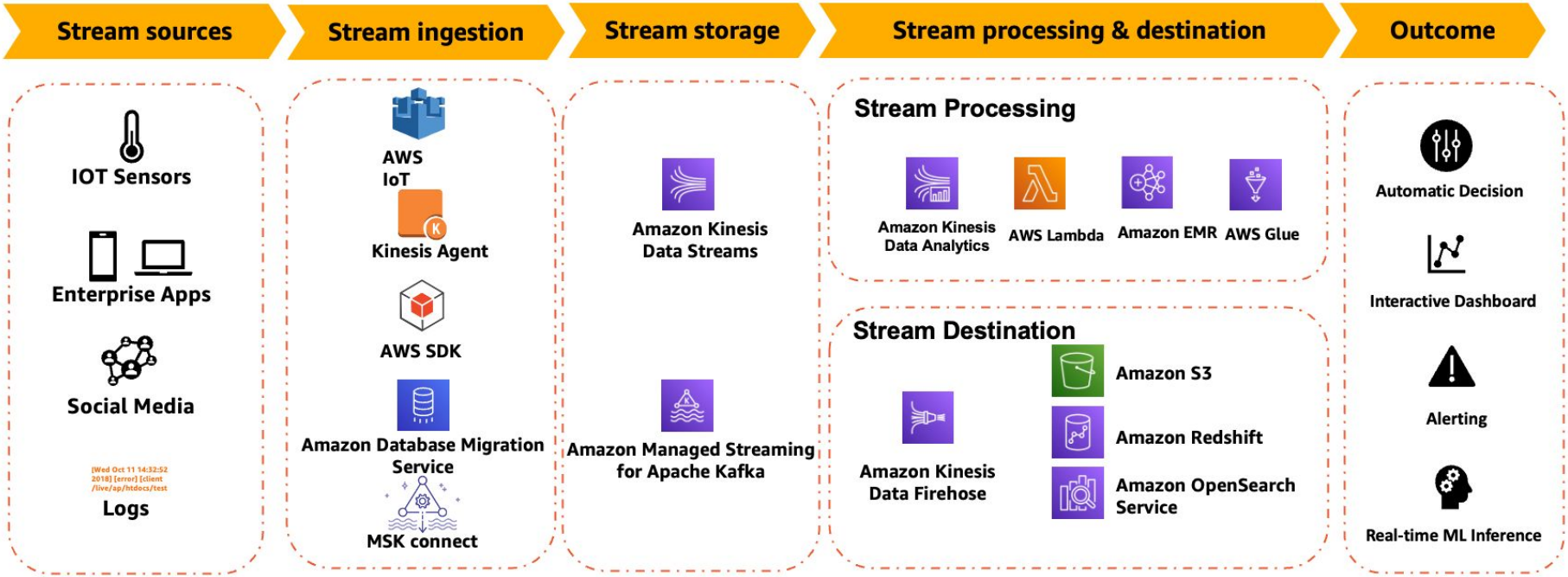
Streaming Architecture: Components

- Destination
  - The destination layer is like a purpose-built destination depending upon your use case.
    - Database
    - Data storage
    - Application logic or server

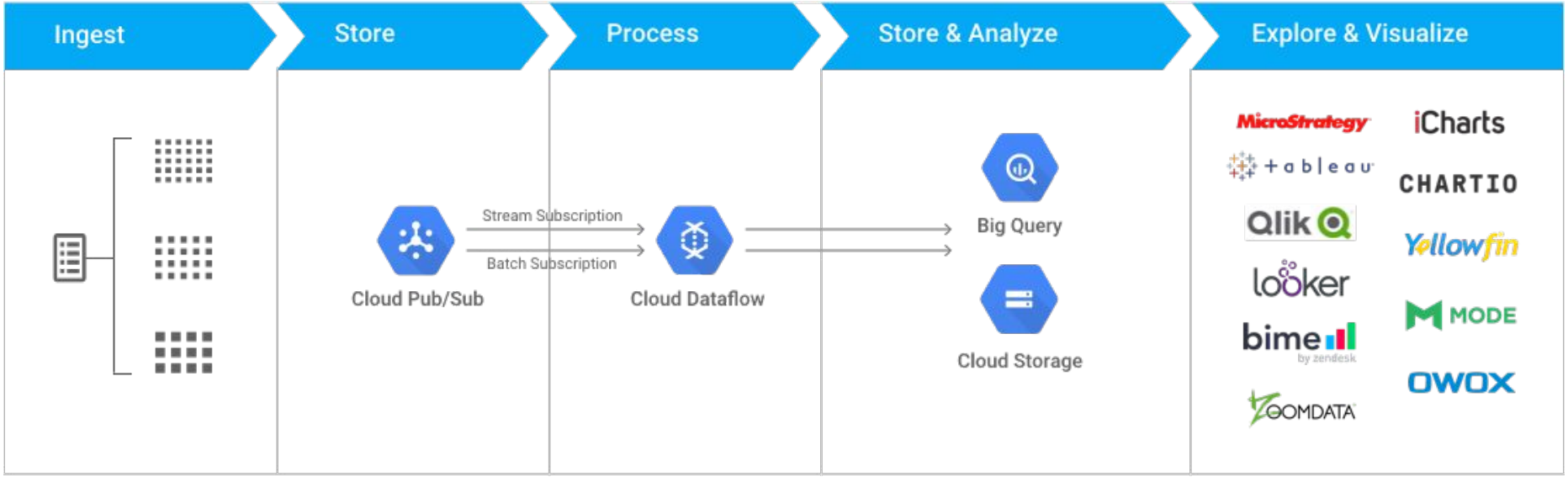
Pub/Sub vs Streaming

- Data structure:
  - Pub-sub: structured messages between publisher and subscriber
  - Streaming: data source semi-structured or unstructured, later, structured
- Message Storage
  - Pub-sub: Deleted after messages consumed
  - Streaming: Stored regardless of consumption
- New Subscribers
  - pub-sub : New subscribers cannot access old messages
  - Streaming: Newly added stream processors can access old data
- Message type:
  - Pub-sub: can be batched
  - Streaming: not batched

AWS streaming



GCP streaming



Django: MTV

- The Model-View-Template (MVT) is slightly different from MVC.
- The main difference between the two patterns is that Django itself takes care of the Controller part (Code that controls the interactions between the Model and View)
- In Django-land, a “view” is a Python callback function for a particular URL, because that callback function describes which data is presented.
- Furthermore, it’s sensible to separate content from presentation – which is where templates come in.
- In Django, a “view” describes which data is presented
- a view normally delegates to a template, which describes **how** the data is presented.
- Where does the “controller” fit in, then? In Django’s case, it’s probably the framework itself: the machinery that sends a request to the appropriate view, according to the Django URL configuration.

Some Notes on the Project

- Your homeworks are graded!
- DO NOT need the “customer” journey (you just put the questions in datastore yourself)
- DO NOT necessarily need a user login
    - although totally fine, helpful in some ways, and not too hard to implement
- DO need a **functional URL** where I can go play the quiz!
- Django uses a variant on MVC called MTV
    - It’s a bit different than what we learned in class!
    - Can use Flask if you want regular MVC; see links in project description
- Do need a DB of some sort to store whatever information you need for the leaderboard.
    - “read/write”
- MVC recommended, since you will at least need some “view”

In-Class

- Please finish up the actual design for your group and show it to me for review
  - This is for your benefit :)
- Once you have done that, start coding the python backend!
  - The classes that you have described in your diagram
    - Suggest using Github for version control