Aalto University
School of Science

# Big Data Ingestion

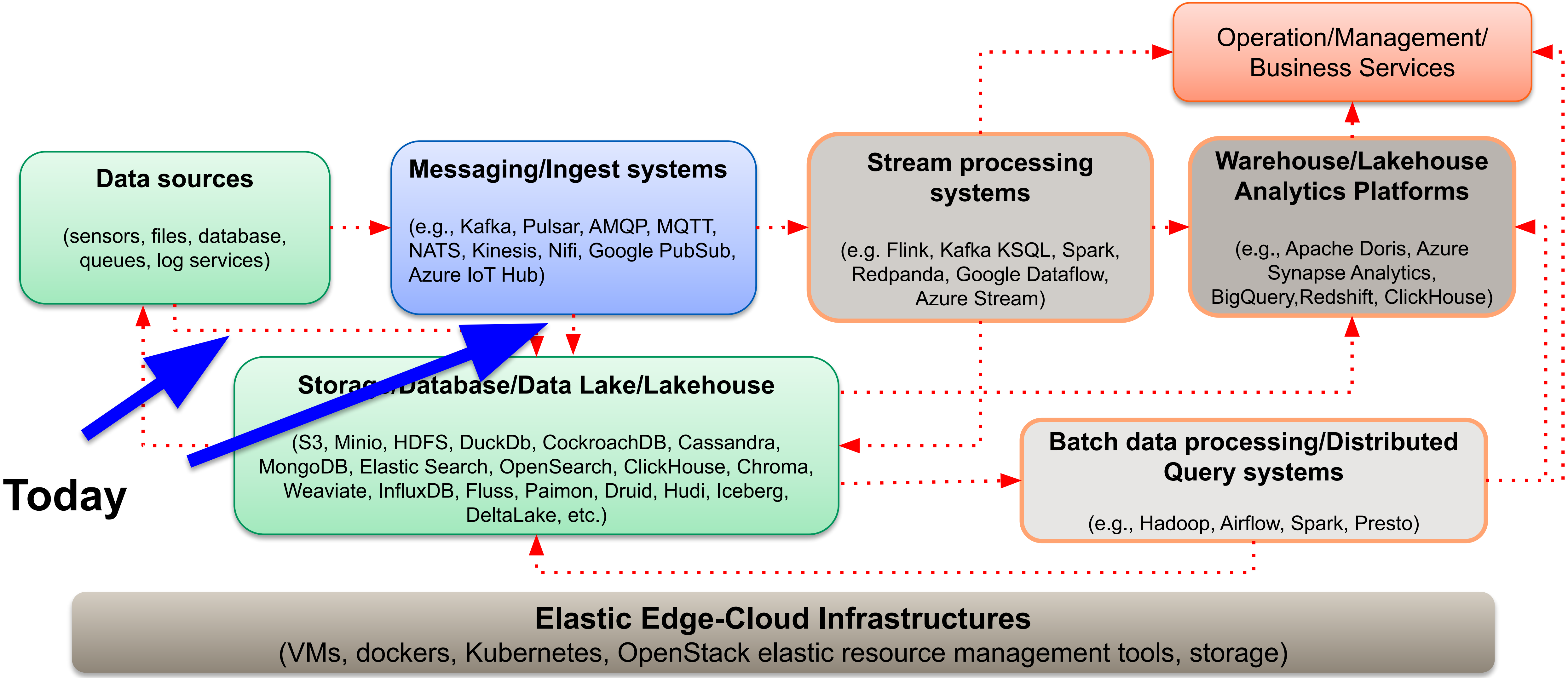
Hong-Linh Truong
Department of Computer Science
linh.truong@aalto.fi

28.1.2026

# Learning objectives

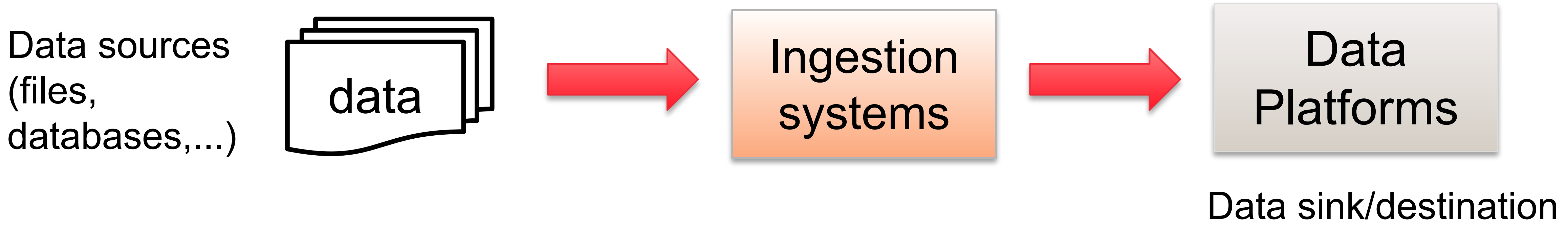
- Understand the overall design of data ingestion
- Study common tasks in data ingestion
- Understand and design efficient, robust data ingestion pipelines/processes
- Learn existing technologies/frameworks for your own design

# Our big data at large-scale: the big picture in this course



# Ingestion systems

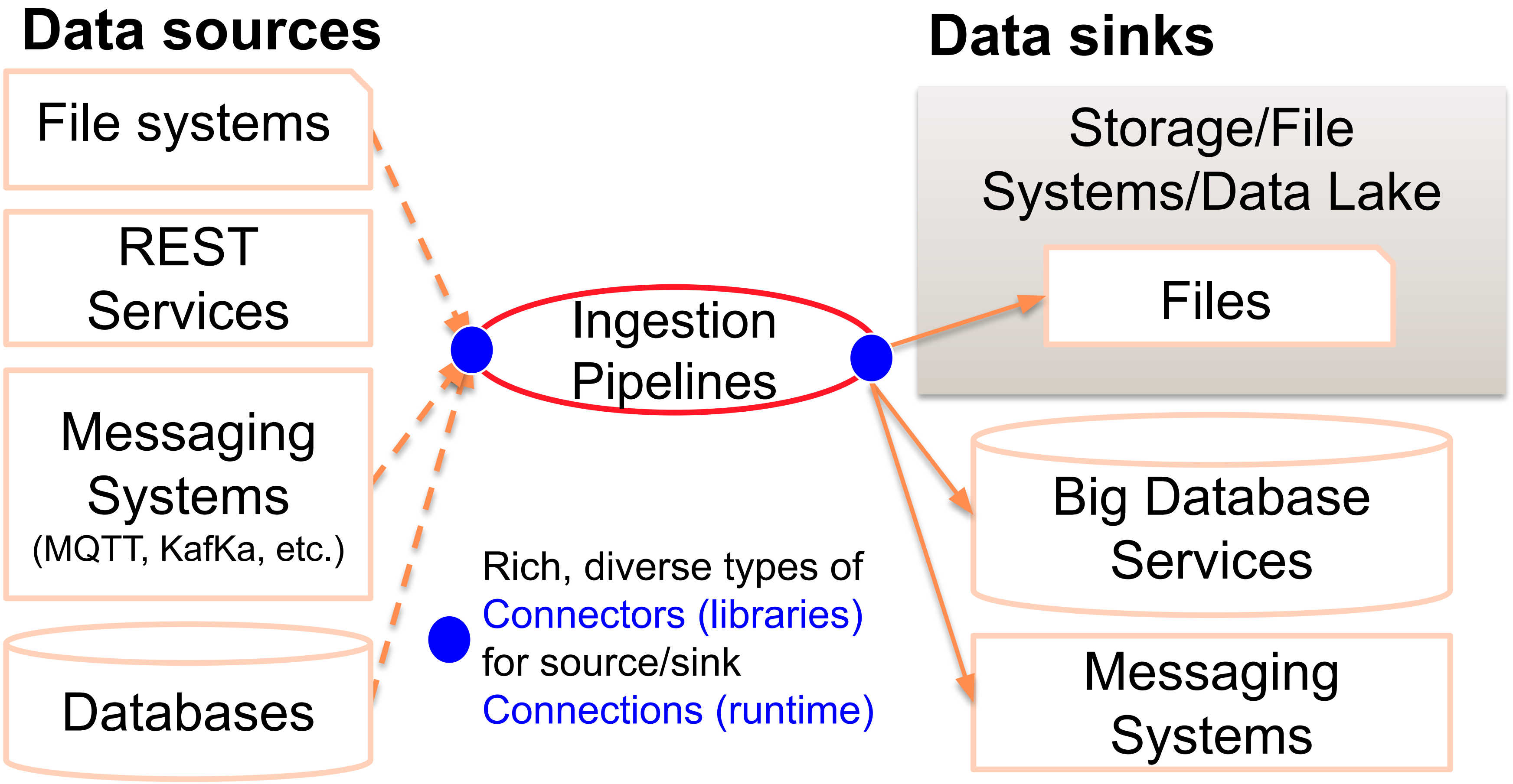
Data ingestion: move data from <u>different sources</u> into data sinks/destinations within data platforms




**The basic goals in terms of data operations:**

- insert new data and upsert (insert and update) data into sinks
  - insert can be "append-only" and big data files can be immutable
- at very large scale!

# Data sources and sinks




## Examples of data sinks

Hadoop File systems
Google Storage
Amazon Storage

Druid
Poison
Google BigQuery
Hive
MongoDB
ElasticSearch
Cassandra
InfluxDB
Hudi
Kafka, Pulsar

# Design and engineering aspects

- Tasks, pipelines and service models
- Non-functional requirements and service level agreement (SLA) between sources and platforms
- Deployment architectures

**Performance and consistency tradeoffs**

**Reusability and extensibility**

# Diverse requirements from data sources

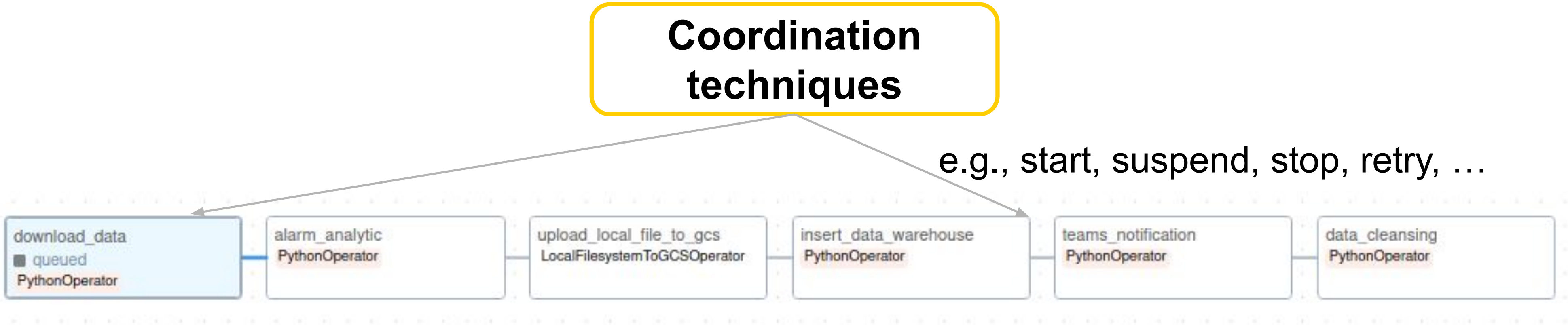
- Requirements based on data characteristics
  - multimodal data with structured, unstructured and semi-structured
  - speed, volume, accuracy, confidentiality, data regulation
- Interact with data sources:
  - Access APIs and protocols
    - REST API, ODBC, SFTP, specific client libs
    - MQTT, AMQP, CoAP, NATS, Kafka,…
  - Connection management:
    - performance, reliability and security
- How deep can a platform support complex requirements?
  - e.g., able to go into inside of data elements (understanding the syntax and semantics of data)?

# Data transformation

- Data transformation:
  - convert data from an existing form to another form for an (analytic) purpose
- Extract, Transform, Load (ETL)
  - Extract data from a source, Transform data and Load (save/store) data into a sink: Extract → Transform → Load
  - ETL has many operations to deal with the semantics/syntax of data and the business of data
- ELT: Extract → Load → Transform
  - Data transformation done after, within the (target) platform
- **Modern ingestion**
  - data transformation together ingestion tasks within a complex pipeline
  - both Transform → Load and Load → Transform designs

**Performance, correctness and quality assurance**

# Coordination of tasks in ingestion pipelines




- Big data ingestion involves
  - many tasks
  - multiple tenants/users
  - ad-hoc, on-demand vs scheduled task pipelines
  - data movement in a single vs cross data centers
- Complex coordination techniques
  - tasks are not in the same machine (executors)
    - → data exchange among tasks (data dependencies)
  - different ways to implement the coordination

When should the ingestion be executed? based on what?

# Modern ingestion pipelines: common tasks and requirements

# Tasks

- Common tasks
  - data extraction
  - change data capture
  - data wrangling/transformation
  - data storing
  - lineage and observability for quality assurance/governance (quality check)
  - backfilling
- Consumer/user defined tasks vs platform tasks
- Other supports within tasks
  - compression, encryption, end-to-end security

**Differences: batch vs near real time (streaming) ingestions**

# Extensible, composable tasks as plugins

- Basic tasks for big data ingestion can be (re)used in different cases
- Support end-user goals
  - enables the user to do many tasks through configurations and extensions
- Enable pluggable approaches is important

input data plugin/component → filter/extract/transform plugin/component → output data plugin/component

# Data access and extraction

- Data Access
  - obtaining/copy data from sources
    - including change data capture (CDC)
  - often built based on common protocols and APIs
    - connector library: strongly related to data storage/database/datalake sink/source)
    - runtime connection management: maintain list of connections created from connectors
  - reusability is important!
- Encryption, masking/anonymization
  - might need to be done when accessing and extracting data
  - also during transfers of data
  - data security requirements, personally identifiable information

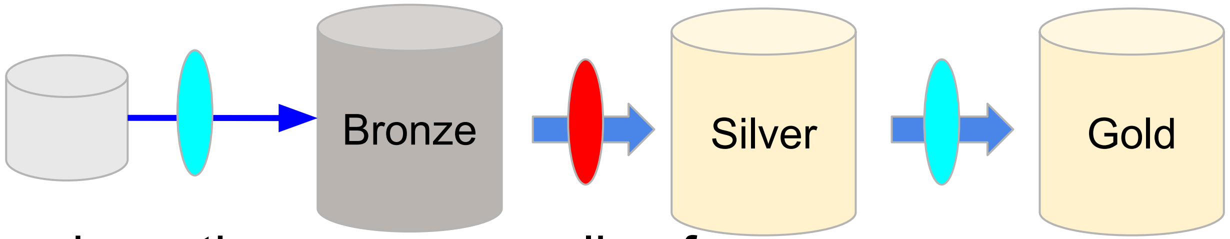
# Change data capture (CDC)

It is important to capture new data, changes at immediately (as soon as possible) for continuous analysis to support decision making

- The principles:
  - capture and ingest only new data by listening data changes
    - "new": application-specific, e.g., based on time, value, and version.
  - leverage many features of databases (update, query, insert operations), data stream offsets and status notification (e.g., the availability of new files)
  - customize detection mechanism
- Implementation in different tools
  - e.g., Redhat Debezium, Hudi DeltaStreamer, Kafka Connect

# Data wrangling

- Convert/transform data from one form to another
  - cleansing, filtering, merging, enriching, inferring, and reshaping data
- Require access to the data content!
- Key design choices
  - when and where: during the ingestion vs after the ingestion
  - by whom: which features will be provided by a platform provider?




ingestion of raw data

wrangling for clean, well structured data

# Examples

**Write your own code with Pandas/Dask and Dataframe?**

**Automatically generate code for wrangling, e.g. using GenAI?**

```
Alarms={}
with open(sys.argv[1], 'rb') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        try:
            #print row['Started']
            alarm_time = datetime.strptime(row['Started'], '%d.%m.%Y %H:%M:%S')
            #diff =start_time - alarm_time
            #print "different time is ",diff
            if alarm_time  >=start_time:
                #print(row['RNW Object Name'], row['Severity'])
                typeOfAlarm = 0
                cleanSeverity = re.sub('\W+','',row['Severity'])
                if (cleanSeverity in mobifone.AlarmSeverity.keys()):
                    typeOfAlarm = mobifone.AlarmSeverity[cleanSeverity]
                #print ("Type of Alarm: ",typeOfAlarm)

                if row['RNW Object Name'] in Alarms:
                    #print "Again"
                    severies =Alarms[row['RNW Object Name']];
                    serveries[typeOfAlarm]=serveries[typeOfAlarm]+1
                else:
                    serveries =[row['RNW Object Name'],0,0,0,0,0,0]
                    serveries[typeOfAlarm]=serveries[typeOfAlarm]+1
                    Alarms[row['RNW Object Name']]=serveries;
        except:
            print "Entry has some problem"
            print row
        #timestamp =long(row['TIME'])
        #times.append(datetime.datetime.fromtimestamp(timestamp/1000))
        #times.append(long(row['TIME']))
        #signals.append(float(row['GSM_SIGNAL_STRENGTH']))
dataframe =pd.DataFrame(Alarms,index=mobifone.AlarmSeverityIndex).transpose()
alarmdata =dataframe.as_matrix();
#TODO print Alarms to fine
#only for debugging
print dataframe
dataframe.to_csv(outputFile, index=False)
```

# Complex data transformation and processing

- More complex data processing
  - extract only important data
    - feature engineering
  - enrich data on the fly with external sources

## Example: extract vectors from images

```python
class TowheeExtractor(BaseExtractor):
    def __init__(self):
        self.towhee_feature_extractor = (
            pipe.input('file_path')
                .map('file_path', 'img', ops.image_decode.cv2_rgb())
                .map('img', 'embedding', ops.image_embedding.timm(model_name='resnet50'))
                .map('embedding', 'embedding', ops.towhee.np_normalize())
                .output('embedding')
            )

    def get_model_name(self):
        return "resnet50"

    def feature_extractor(self, image_file):
        ##currently only a single figure so we have to get the first element
        embedding=self.towhee_feature_extractor(image_file).get()[0]
        return embedding
```

# Complex code & libraries hide low level distributed/parallel tasks

- Complex distributed and parallel tasks for ingestion
- Complex coordination
- Underlying, internal task models:
  - MapReduce model
  - embarrassingly parallel model
  - full directed acyclic graph (DAG) task model

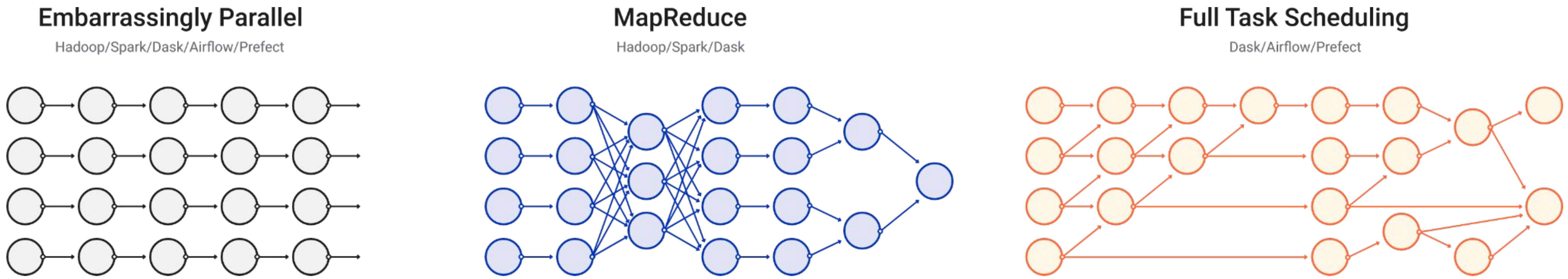
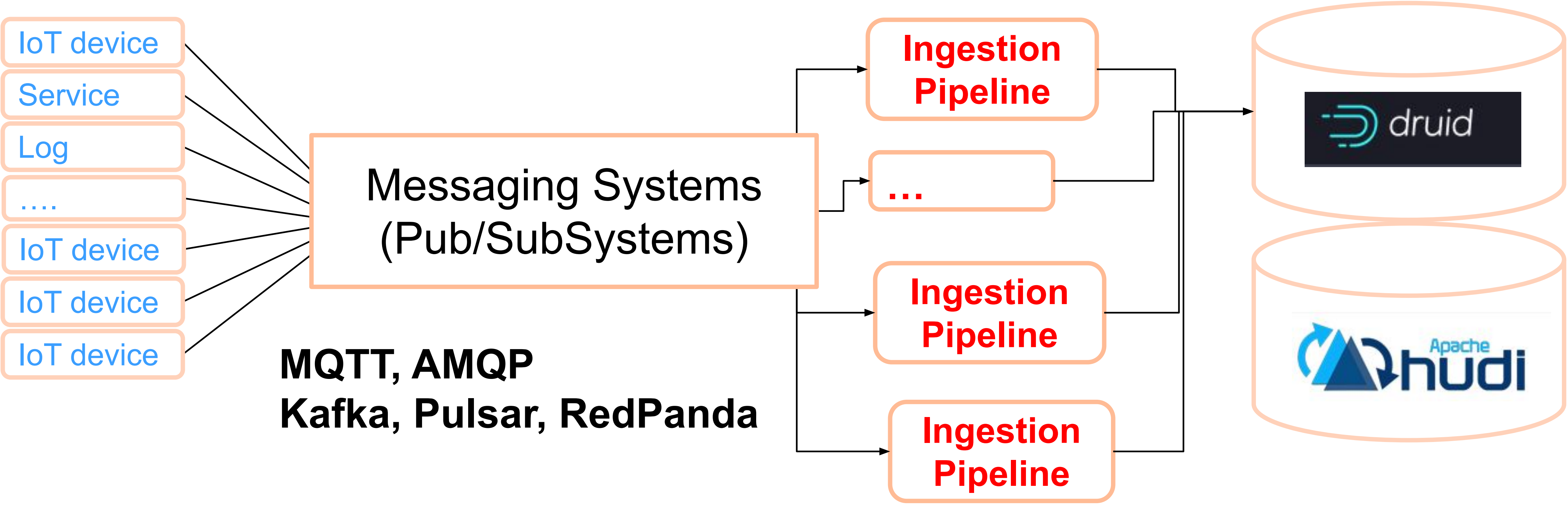


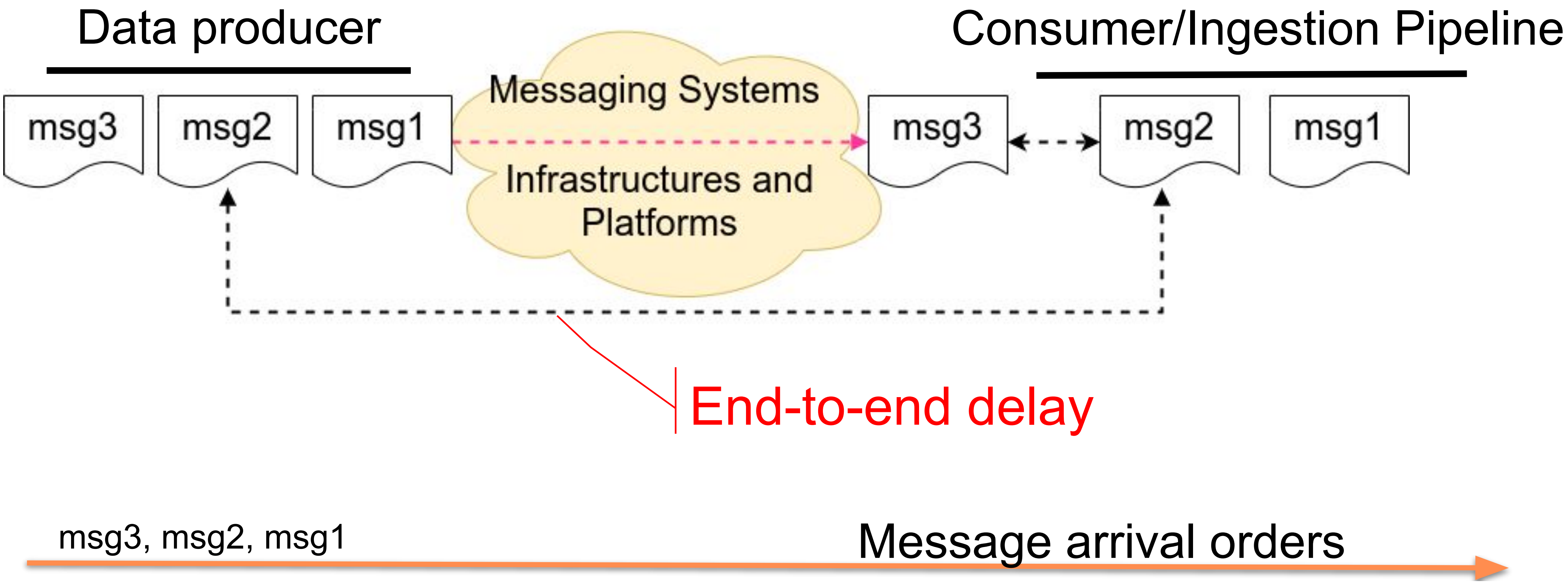

**Figure source:** *https://docs.dask.org/en/stable/graphs.html*

# Near real time streaming ingestion

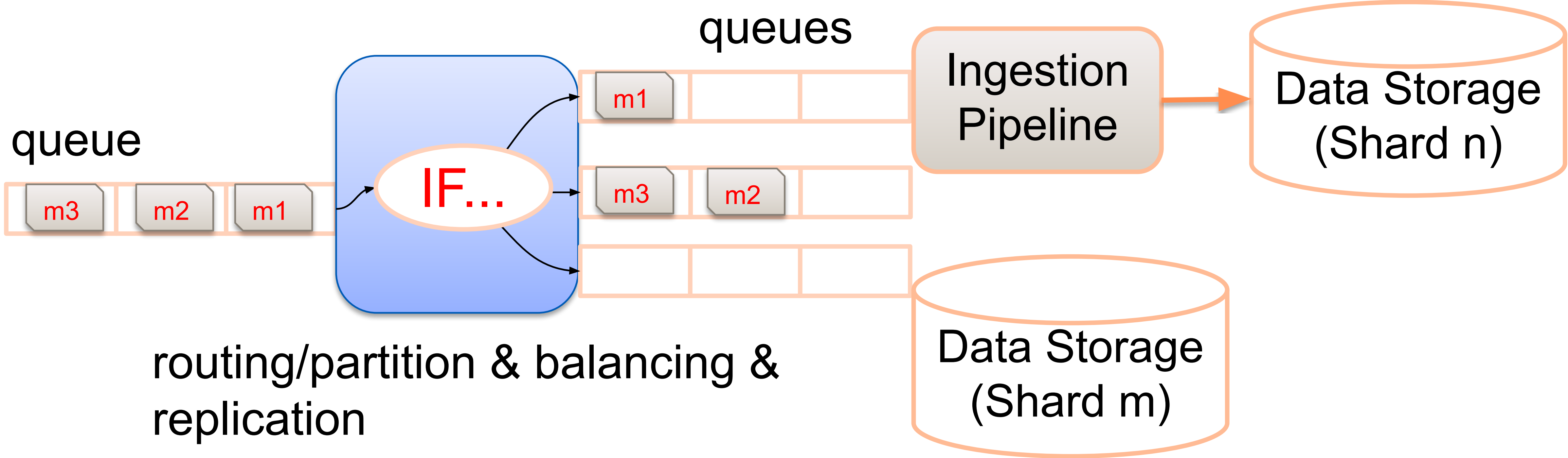
**Run as a service or arbitrary and can be complex**



# Key issues in streaming data ingestion



# Split (pub/sub) and partition with ingestion



# Some key issues for ingestion of streaming data

- Late data, data out of order
- Data loss or data duplication
  - e.g, at most once, at least once or exactly once
- Back pressure and data retention
  - for individual components or the whole pipelines
- Scalability and elasticity
  - changes in data streams can be unpredictable
- Data quality
  - how to do it with fast processing and minimum overhead

# Dealing with diverse data structures

- The data sender/producer and the receiver/consumer are diverse
  - implemented with different languages and software technologies
    - need to guarantee the message syntax and semantics
  - performance overhead due to data format conversion
- Solutions: don’t assume! agreed in advance
  - agreed in advance ⇒ within the implementation or with a standard
  - know and use tools to deal with syntax differences
- Understanding the syntax allows some automatic transformations/quality checks
- But semantics are domain/application-specific

# Dealing with diverse data structures: example of interoperability in data transfer: Arvo

```
{
        "namespace": "bdp.courses.aalto.fi",
        "type": "record",
        "name": "event",
        "fields": [
            {"name":"station_id", "type":"string"},
            {"name":"datapoint_id", "type":"int"},
            {"name":"alarm_id", "type":"int"},
            {"name":"event_time", "type":"int"},
            {"name":"value", "type":"float"},
            {"name":"valueThreshold", "type":"float"},
            {"name":"isActive", "type":"boolean"}

        ]
}
```

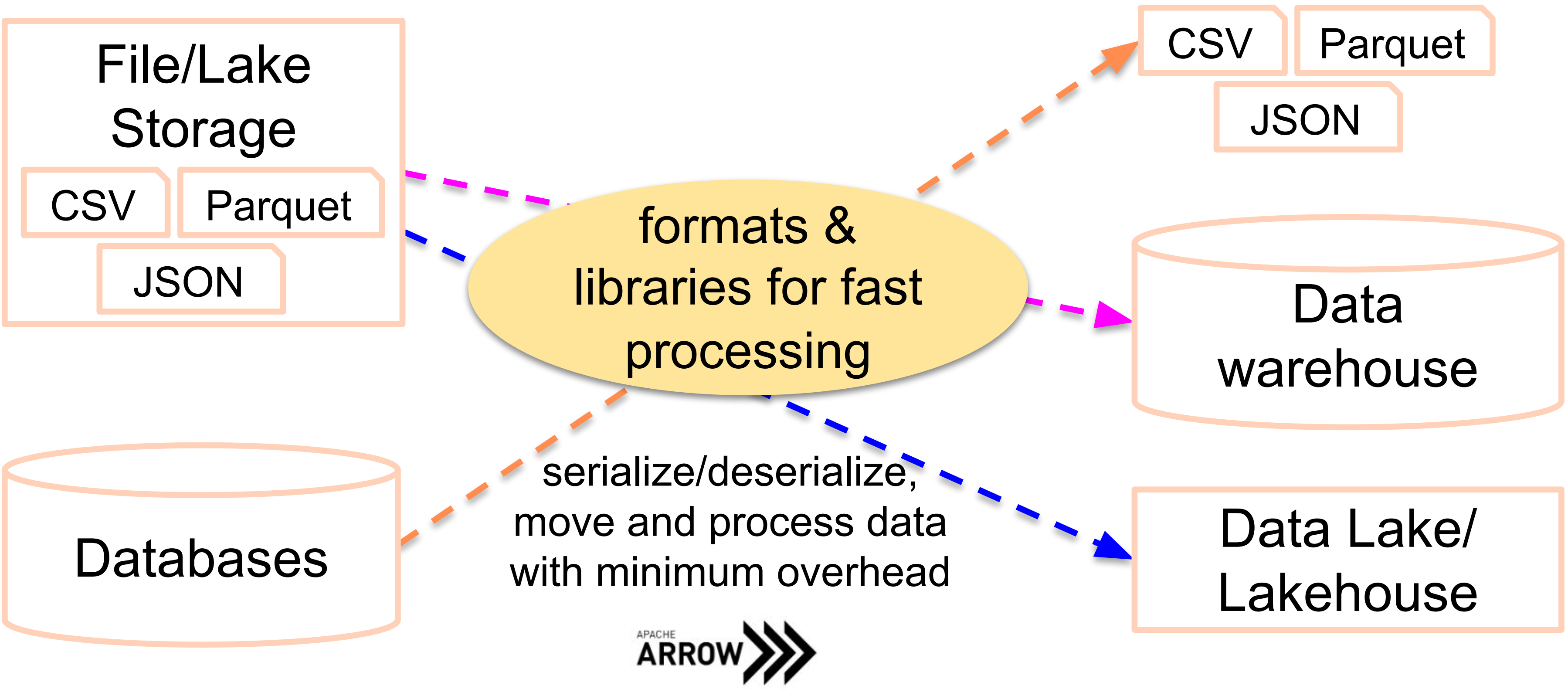
Syntax specification
https://avro.apache.org/


Relevant issues: data compression and security; schema validation and evolution

# Dealing with diverse data structures: interoperability in data processing in ETL

**Data sources and formats**

**Data sinks and formats**




**Example: Apache Arrow**

in-memory columnar format

rich ecosystems

https://arrow.apache.org/

# Ingestion pipelines/processes: composition, deployment, orchestration, and quality assurance

# Complex deployment and composition models (1)

Understanding strong dependencies between protocols/APIs, security, performance, connection management, and service-level agreement (SLA)

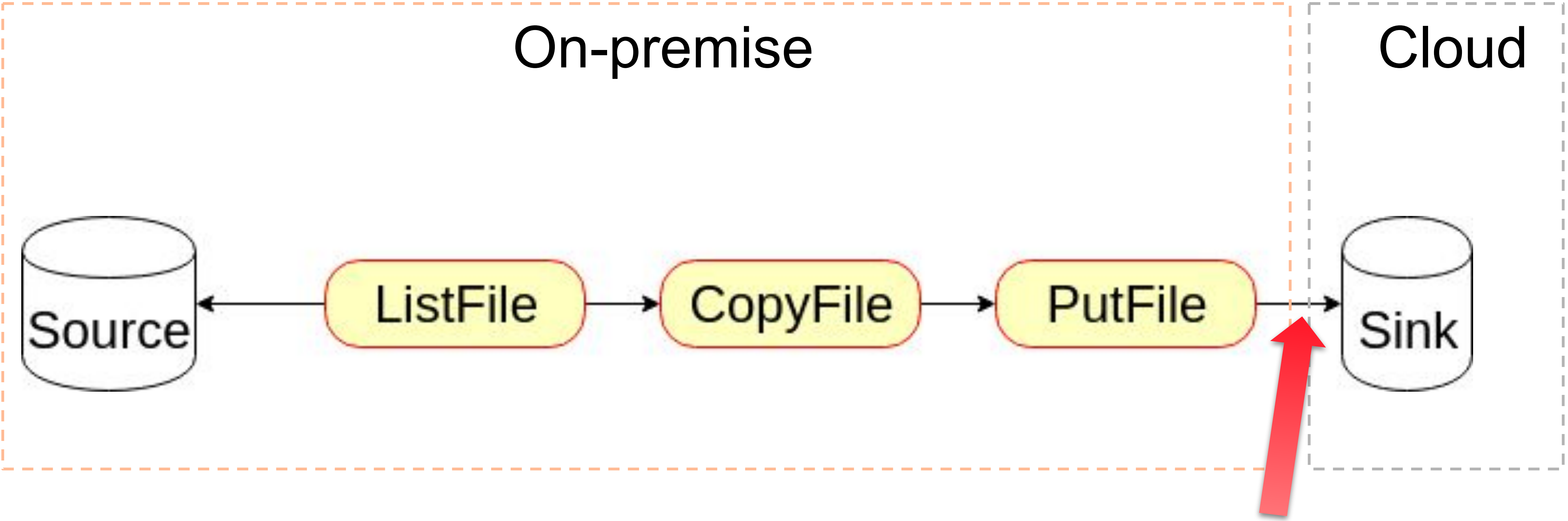
**An exemplified ingestion pipeline**


**Third-parties/ Tenant**

- Ownership
- Development and deployment
  - models: workflows, container, serverless
  - where
- Scheduling strategies
- Failure handling

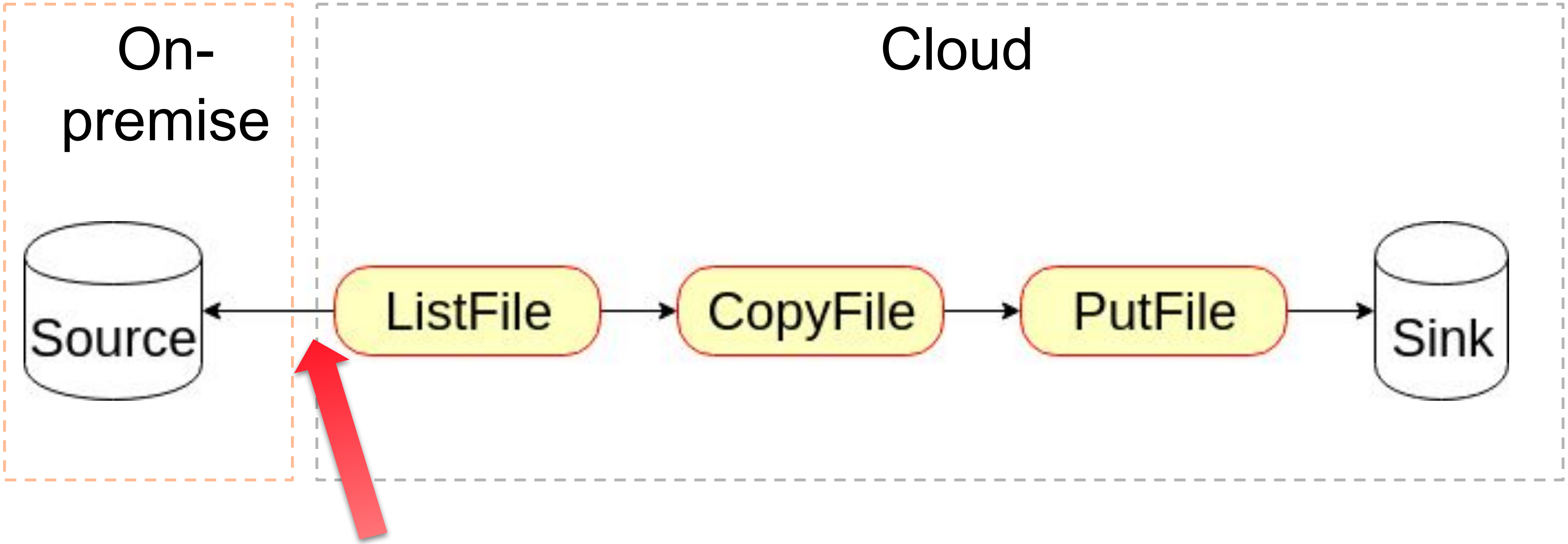
**Tenant/ Platform provider**

# Complex deployment and composition models (2)



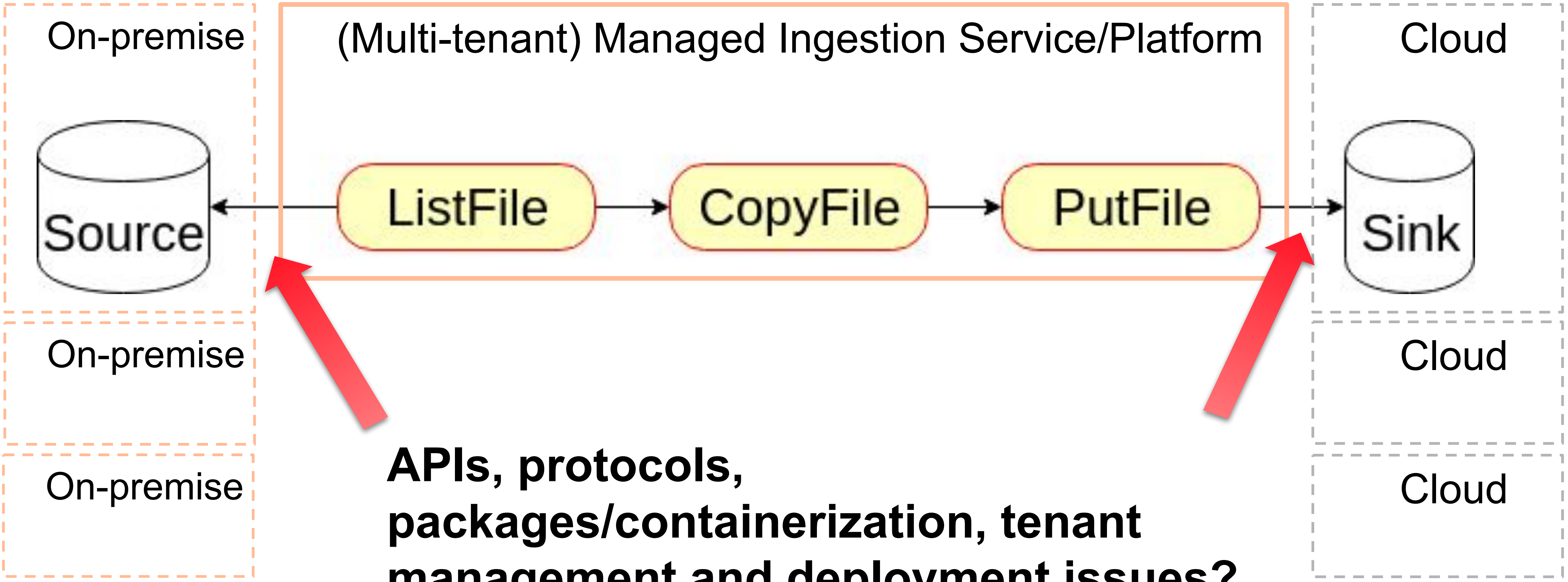

**APIs, protocols and deployment issues?**

# Complex deployment and composition models (3)




**APIs, protocols and deployment issues?**

# Complex deployment and composition models (4)




**APIs, protocols, packages/containerization, tenant management and deployment issues?**

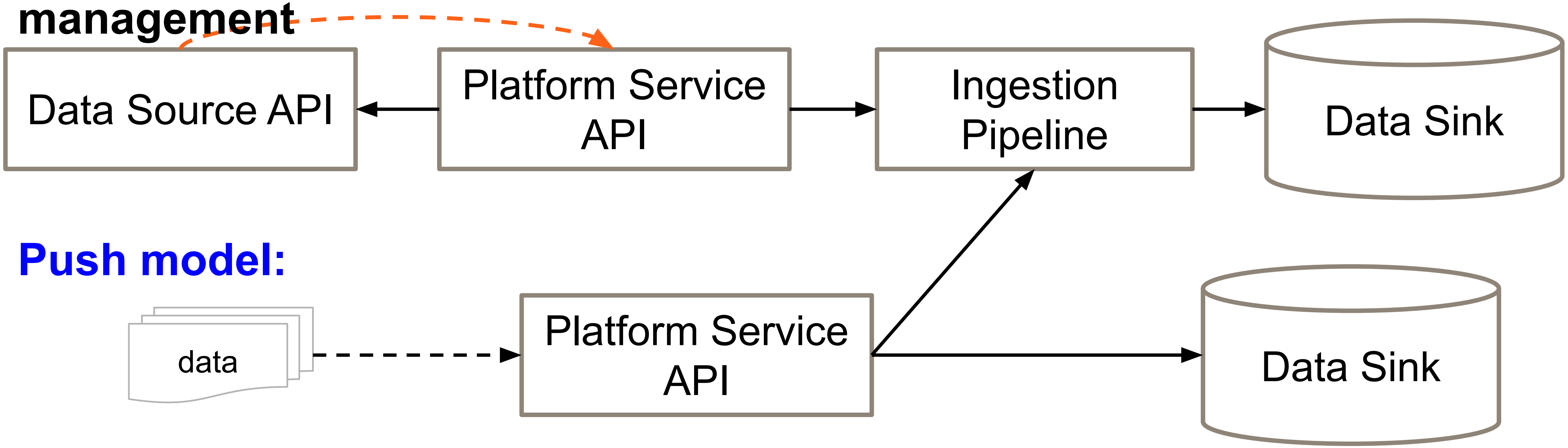
# Orchestrating batch ingestion pipelines

- Data to be ingested is bounded
  - files or messages are finite
- Ingestion architectural styles
  - (1) Direct APIs, (2) reactive pipelines, (3) tasks/workflows
- Incremental ingestion
  - dealing with the same data source but the data in the source has been changed over the time (related to change data capture)
- Parallel and distributed execution
  - use workflows and distributed processing engines

# Simple, direct APIs for ingestion

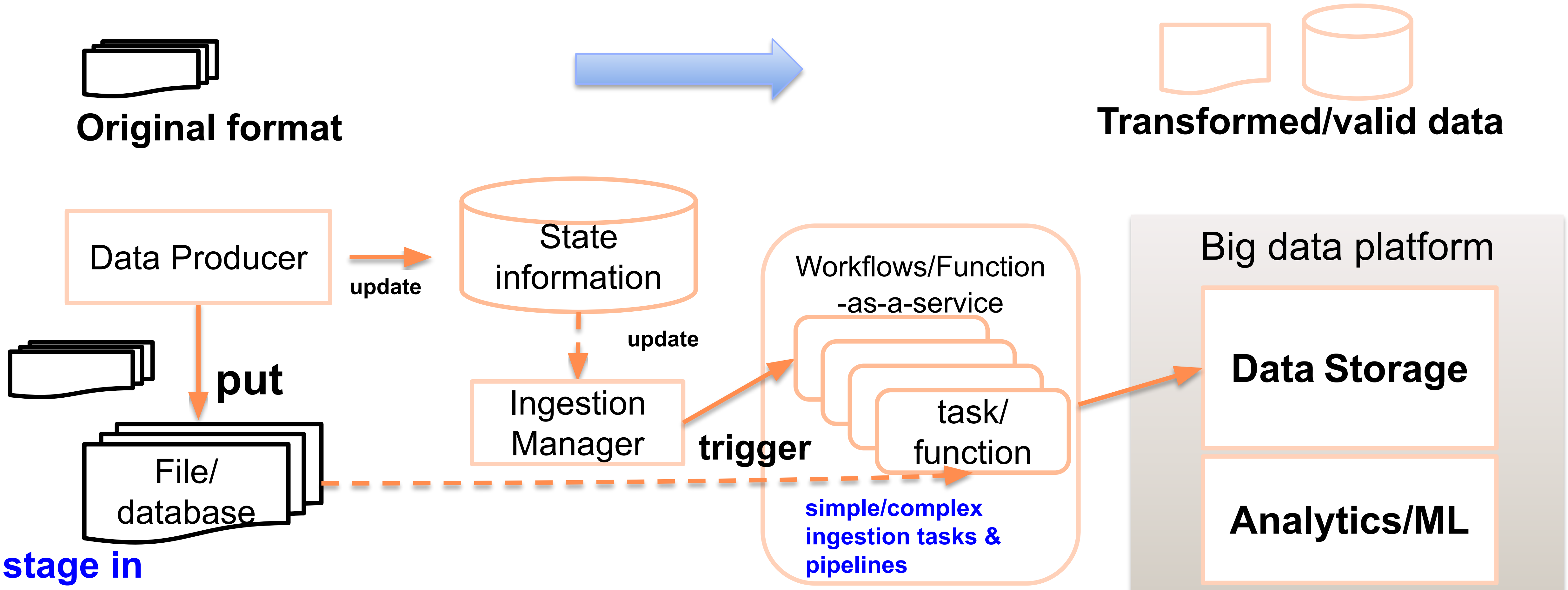
**Pull model:** register webhook/API ⇒ Connection management

**Push model:**




**Analyze pros and cons in the design**

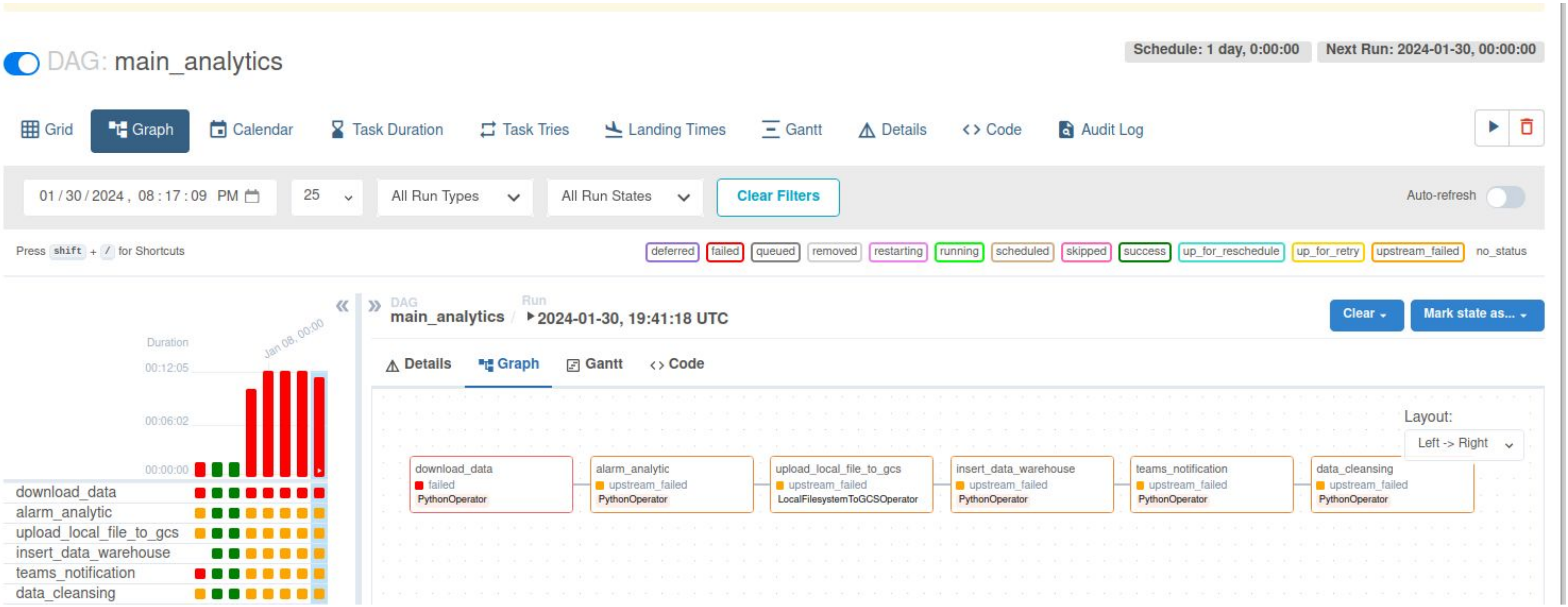
# Reactive pipelines with functions/workflows/containers



# Ingestion workflows orchestration

- Different tasks
  - access and copy, extract, covert, quality check, and write data
  - tasks are connected based on data or control flows
- Workflows
  - a set of connected tasks is executed by an engine
  - tasks can be scheduled and executed in different places
  - flexible designs
- Different tenants have different service level agreements
  - performance, reliability, and cost.

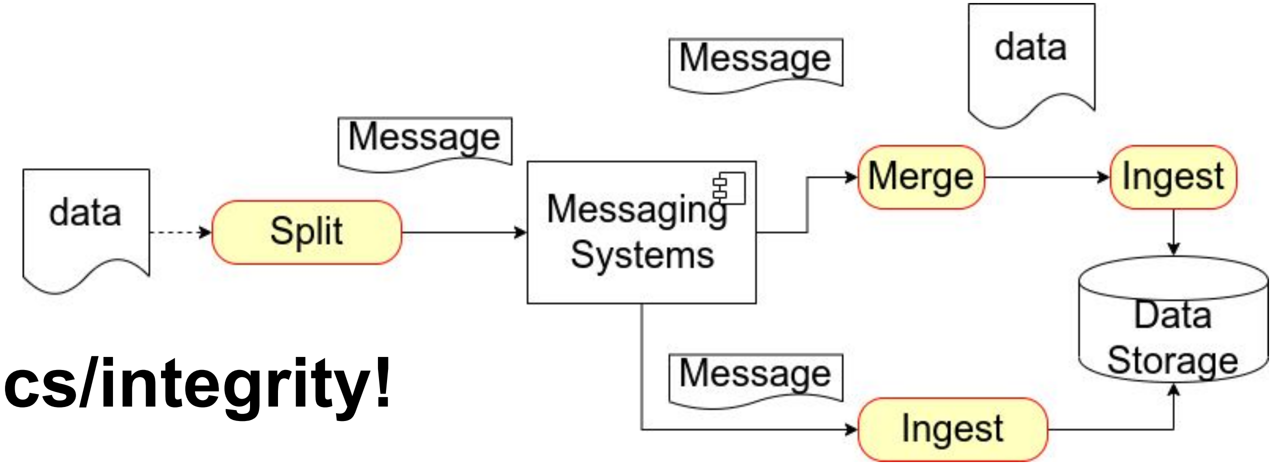
# E.g., workflow based on scheduled time, with Apache Airflow



# Microbatching in data ingestion

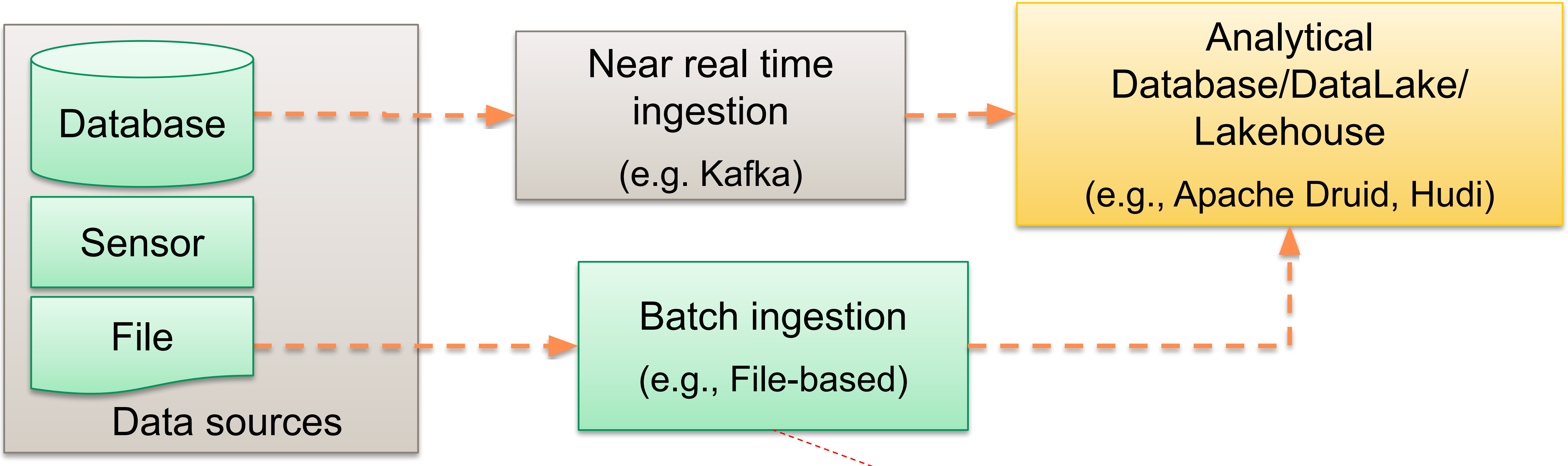
- Microbatching: we mean the strategy to deal with big dataset using batches of data (small chunks)
  - not necessary the same as using batch systems to transfer small data in near real time
- Data is split into different chunks for ingestion
  - using streaming or batch systems to transfer data chunks
  - chunks are ingested into the system, or merged and then ingested

**Example: with streaming system**



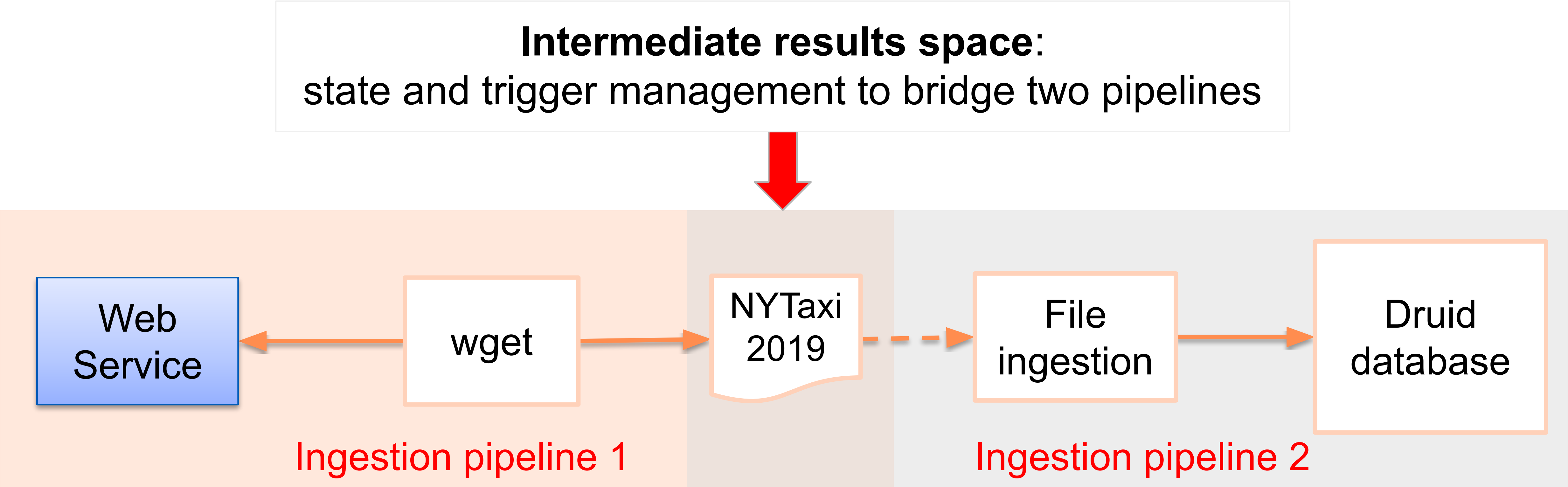

**Challenge: the data semantics/integrity!**

# Supporting multiple types of pipelines for the same data sink




Programming models, orchestration and scheduling are very diverse

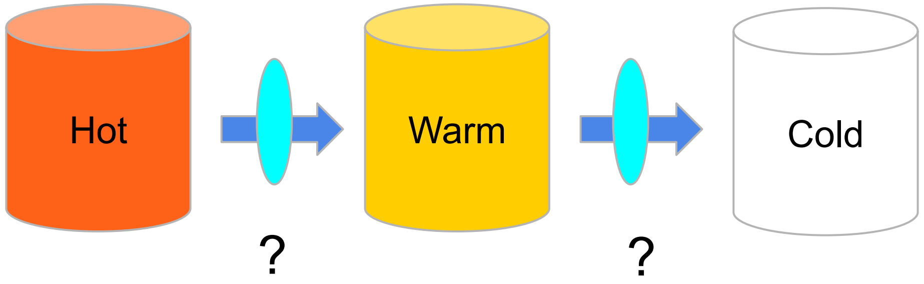
# Connecting different ingestion pipelines: bridge designs



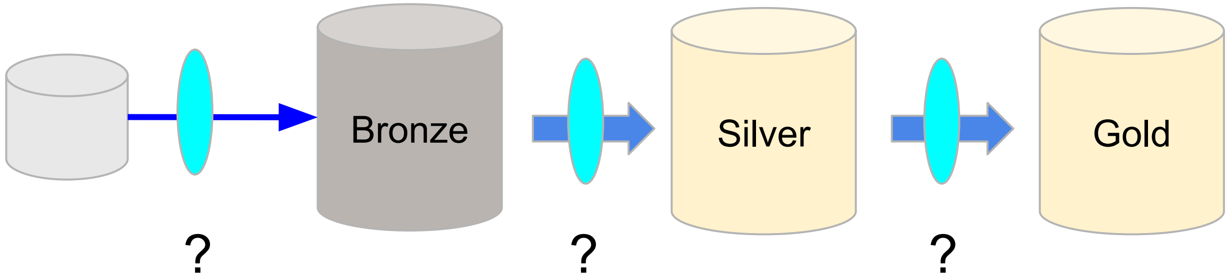

**Real-world: a single stack might not be enough, both pipelines and their connections are complex**

# Discussion: streaming or batch ingestion

## Access frequency: Hot → Warm → Cold

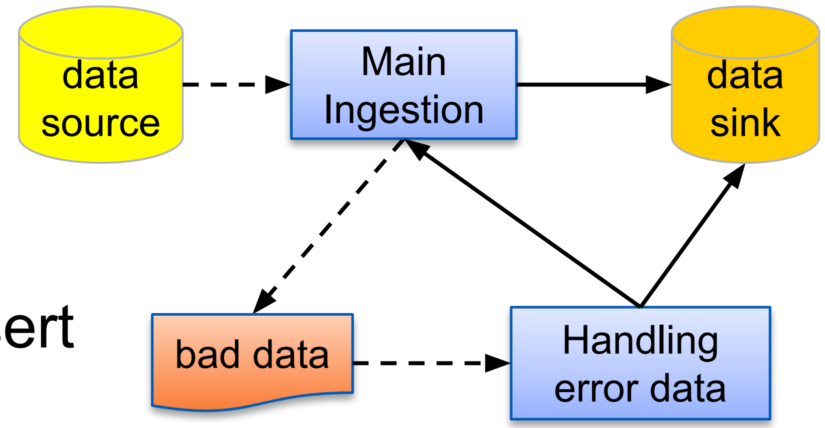



## Medallion architecture

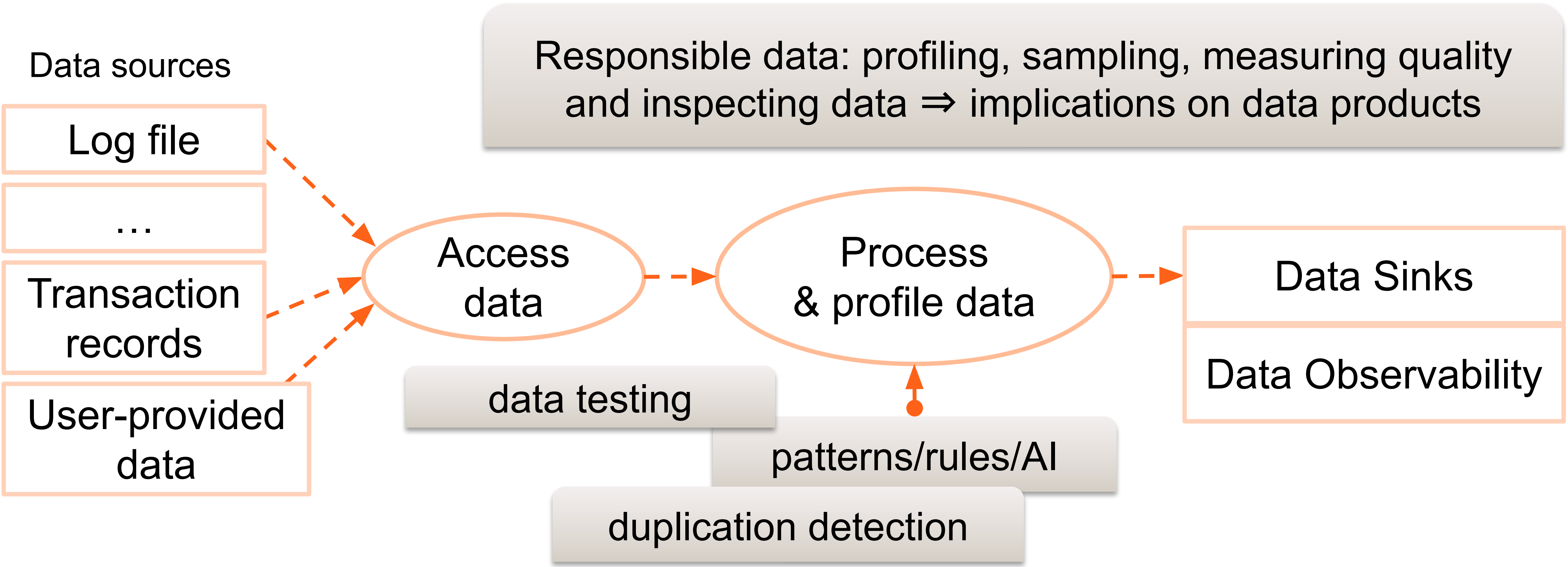


# Failure handling

- Data error records
  - abort completely vs ingest qualified records and return errors
  - strategies for duplicated data
    - discard, use last value, etc.
  - acceptance error threshold
- Idempotent design
  - not causing a problem/introducing new data, even if we repeat the same ingestion
- Differences for insert (including append-only) and upsert
- Task failure handling
  - Rollback and retry → costly
  - Avoid big data chunk
    - small data ingestion → performance overhead



# Quality control/data regulation assurance




**Challenging issues: misinformation, GDPR, data quality, inappropriate content**

# Data lineage and observability (1)

- FAIR principles (https://www.nature.com/articles/sdata201618)
  - findable, accessible, interoperable, and reusable
- Lineage/Provenance
  - capture relevant information for understanding how data has been moved, transferred, processed, etc.
  - metadata models: W3C Provenance Model, DataHub, etc.
- Key issues
  - which metadata must be captured?
  - based on existing tools or your own?
- Instrumentation/logging processes and automated data lineage → performance overhead!

Check: *https://datahubproject.io/docs/metadata-modeling/metadata-model*

# Data lineage and observability (2)

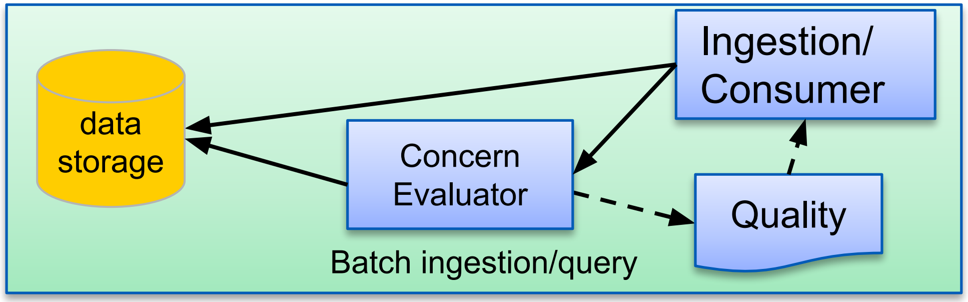
- Data observability: the health about data
  - near-real time metrics, offline checks and possible dashboards
  - similar to service observability, relying on traces, logs, metrics, etc.
- Focus on data
  - data metrics (volumes, data quality, schemas, lineage)
  - issues due to data problems
  - data ingestion processes/workflows
- Some solutions
  - validation of data against design schemas (e.g., Schema Registry in Kafka)
  - checks of realtime and offline data quality attributes → integrate with data ingestion processes or offline data profiling
  - integrated data quality tests in pipelines (e.g., data testing)

**Tools:** great expectations, OpenLineage, Microsoft Presidio, YData Profiling, aws deequ Public

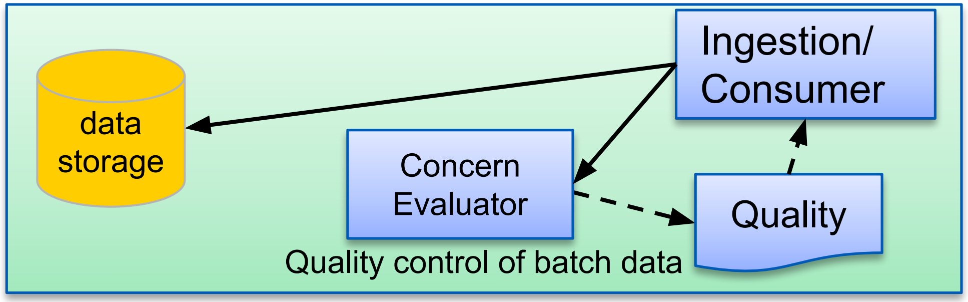
# Quality control and regulation assurance (1)

Design: different evaluation mechanisms

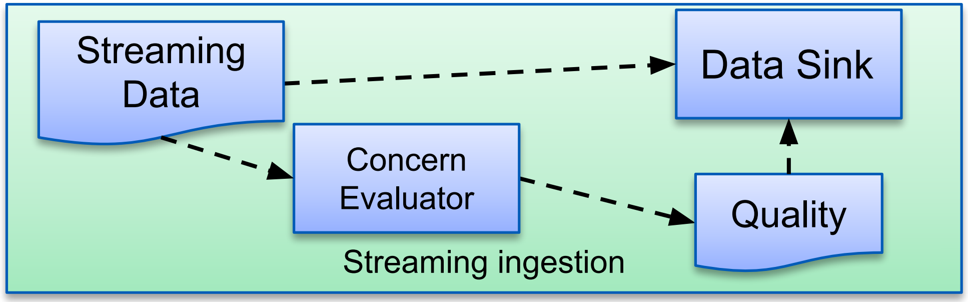
**Pull, pass-by-reference model for evaluating data concerns**



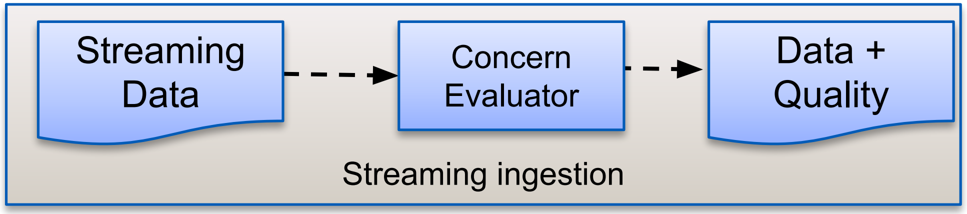

**Pull, pass-by-value model for evaluating data concerns**




**Push model for evaluating data concerns of active data sources:** sampling + delay




Potential performance bottleneck




**Reading Source:** H. -L. Truong and S. Dustdar, "On Evaluating and Publishing Data Concerns for Data as a Service," *2010 IEEE Asia-Pacific Services Computing Conference*, doi: 10.1109/APSCC.2010.54.

# Quality control and regulation assurance (2)

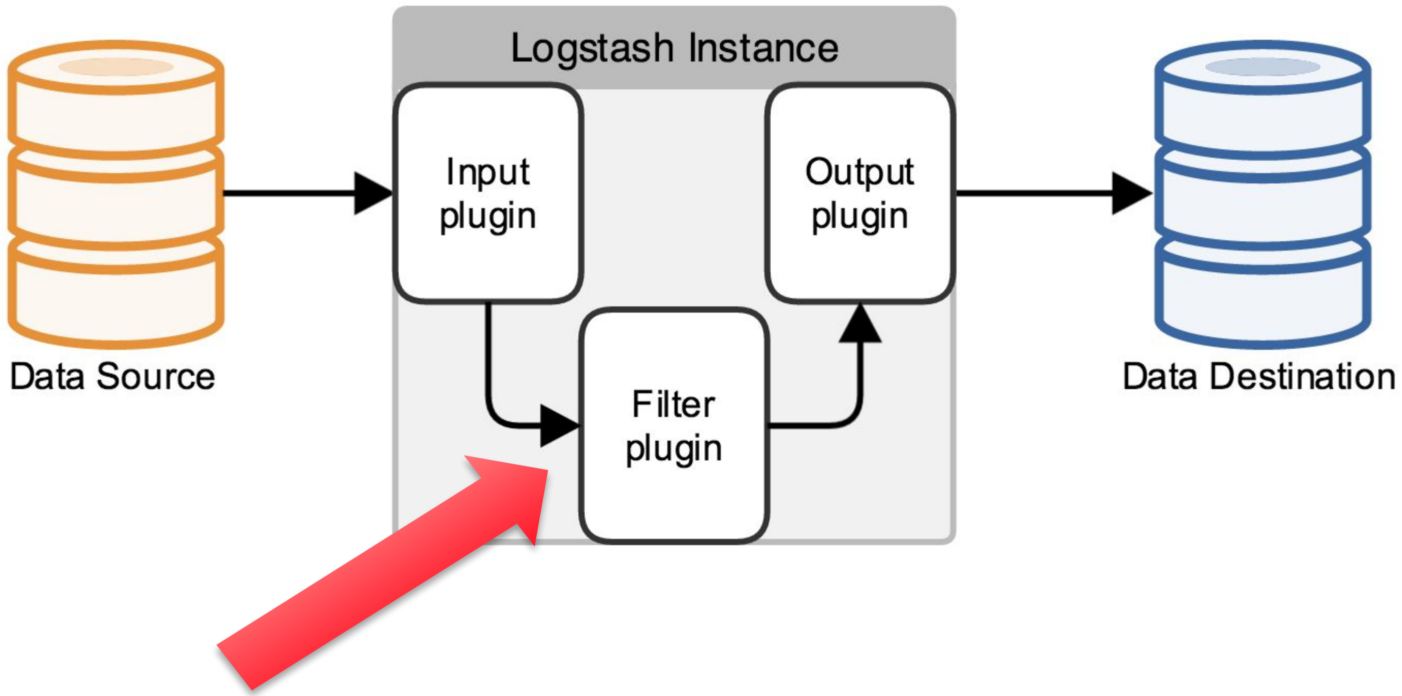
- Before, after or during the ingestion/transformation
- In-process vs out-process
  - in process: using libraries doing data quality → must be very fast
  - out-process: a separate task in the workflow or external programs/services
- Profiling, sampling, AI/ML techniques for data quality
- Examples:
  - Using a separate program like PyDeequ Spark to check quality
    - https://github.com/rdsea/bigdataplatforms/tree/master/tutorials/dataquality
  - Anonymizing data
    - https://microsoft.github.io/presidio/anonymizer/

# Example tooling for ingestion pipelines

# Study existing tools

- Different ways to deliver ingestion pipelines
- (Traditional) ways of REST API/specific client libraries
  - upload using put/get operations
- Workflows
  - self-developed workflows vs automatically generated workflows
- Pipelines are bundled into containers
  - self-developed vs generic pipelines based on user configurations

# Design tools for ingestion processes 1: Logstash




```
input {
  file {
    path => "${MY_INPUT_DIR}/bts-data-alarm-2017.csv"
    start_position => "beginning"
  }
}
filter {
  csv {
    separator => ","
    columns => ["station_id","datapoint_id","alarm_id",
  }

}

output {
    stdout {}
    #...
}
```

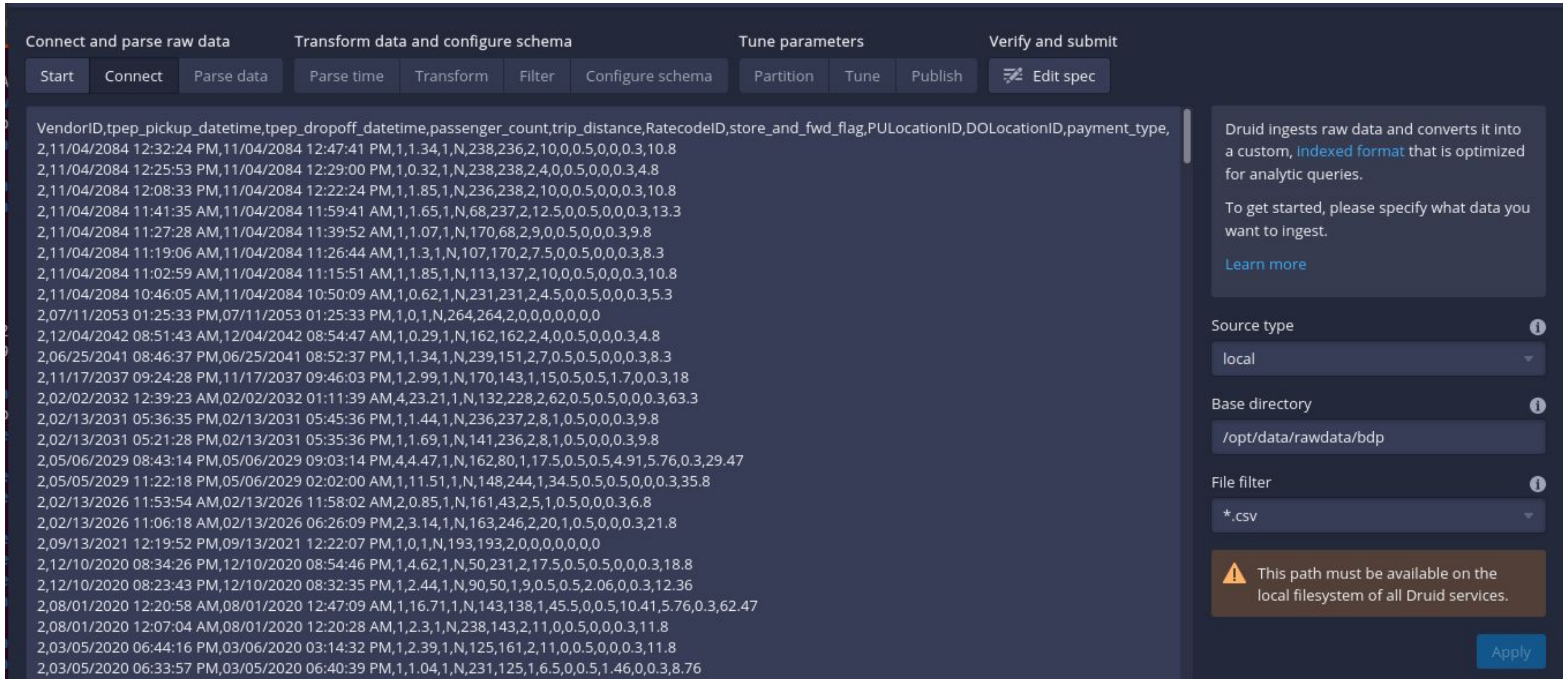
**Pipeline is defined in a configuration file**

**Pluggable approaches**

**Figure source:** *https://www.elastic.co/guide/en/logstash/current/getting-started-with-logstash.html* (from the previous version of Logstash)

# Design tools for ingestion processes 2: Apache Druid

Allow the user to build the plan: select tasks, define configuration, etc. and then generate ingestion pipelines



# Design tools for ingestion processes 3: Apache Nifi

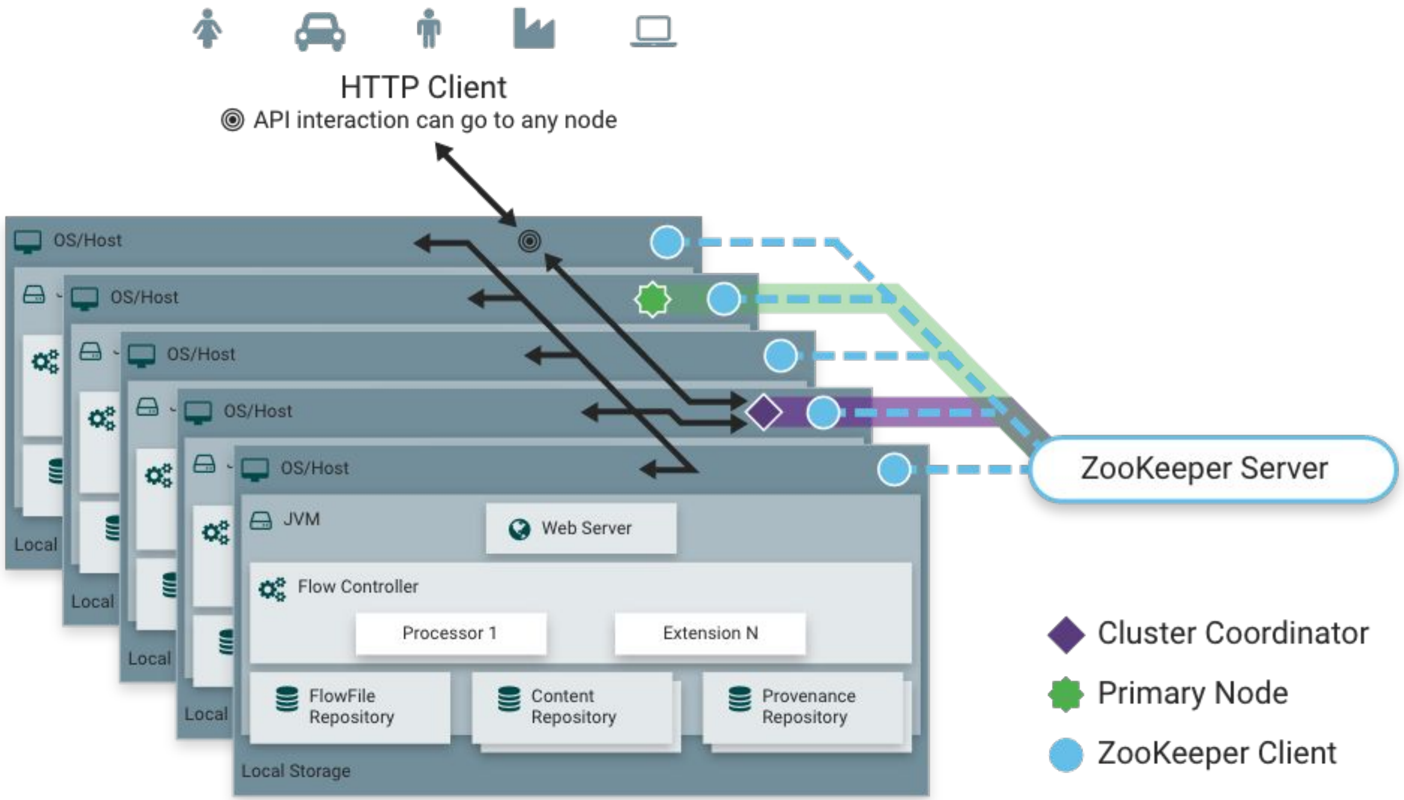


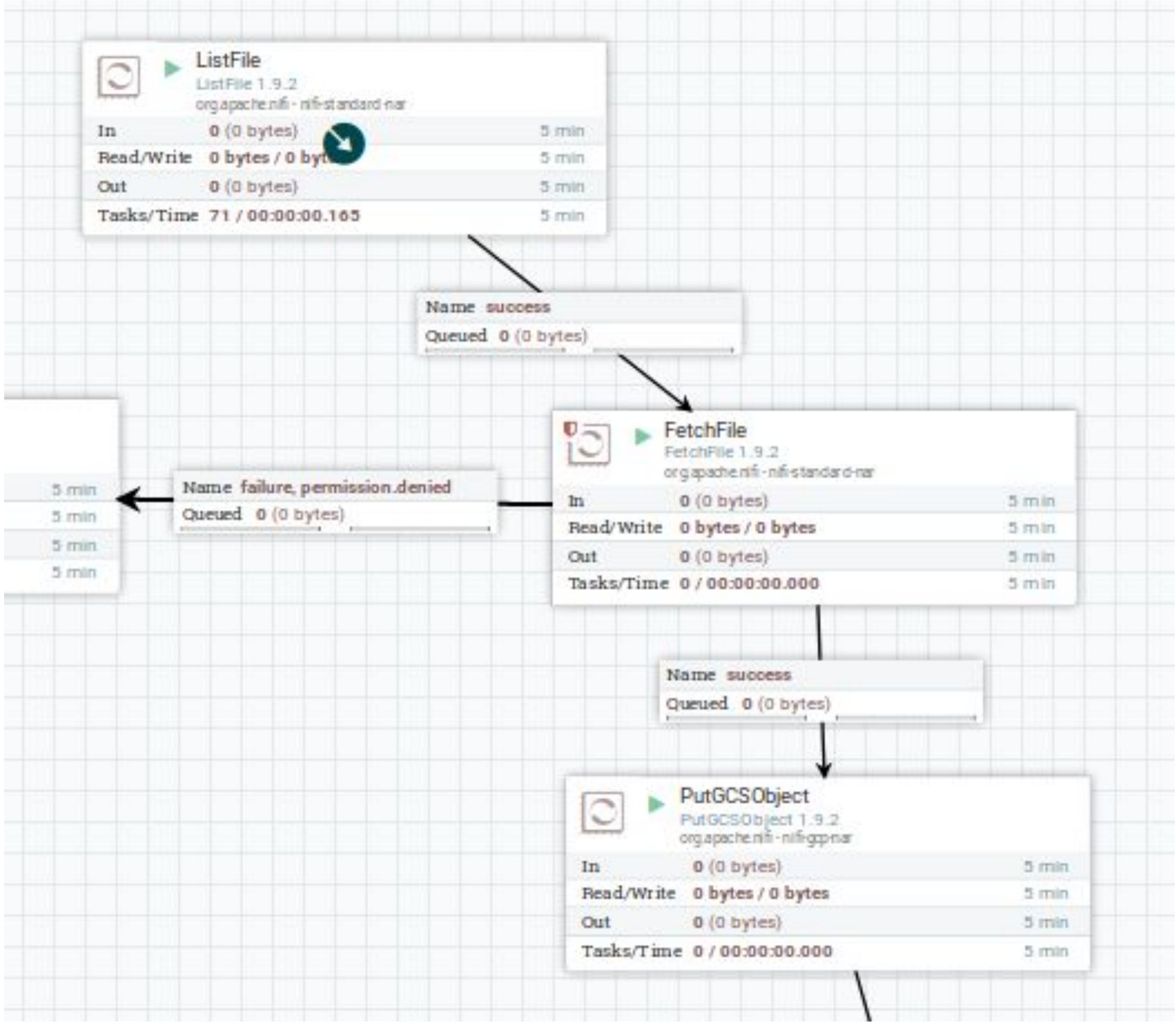

**Figure source:** *https://nifi.apache.org/nifi-docs/administration-guide.html#clustering*

# Design tools for ingestion processes: Apache Nifi - key concept

- Data is encapsulated into “FlowFile”
- Processor (Component) performs tasks
- Processor handle FlowFile and has different states
  - each state indicates the results of processing that can be used for establishing relationships to other components
- Processors are connected by Connection
- Connection can have many relationships based on states of upstream Processors

# Design tools for ingestion processes: Apache Nifi

See the tutorial: https://github.com/rdsea/bigdataplatforms/tree/master/tutorials/nifi



# Summary

- Different designs of data ingestion for batch and streaming
- Ingestion is a complex pipeline
  - many different sub tasks
  - complex requirements w.r.t performance, scale, failure handling
- Different tools/stacks/services available
  - share composable design principles, but different software models and deployments →explore them for your work
- Do real-world designs
  - hands-ons with widely used tools
  - complex designs but we do not need to “reinvent the wheel” → stay with core concepts and requirements to find the right tools!

Thanks!

Hong-Linh Truong
Department of Computer Science

rdsea.github.io

Kiitos
**aalto.fi**