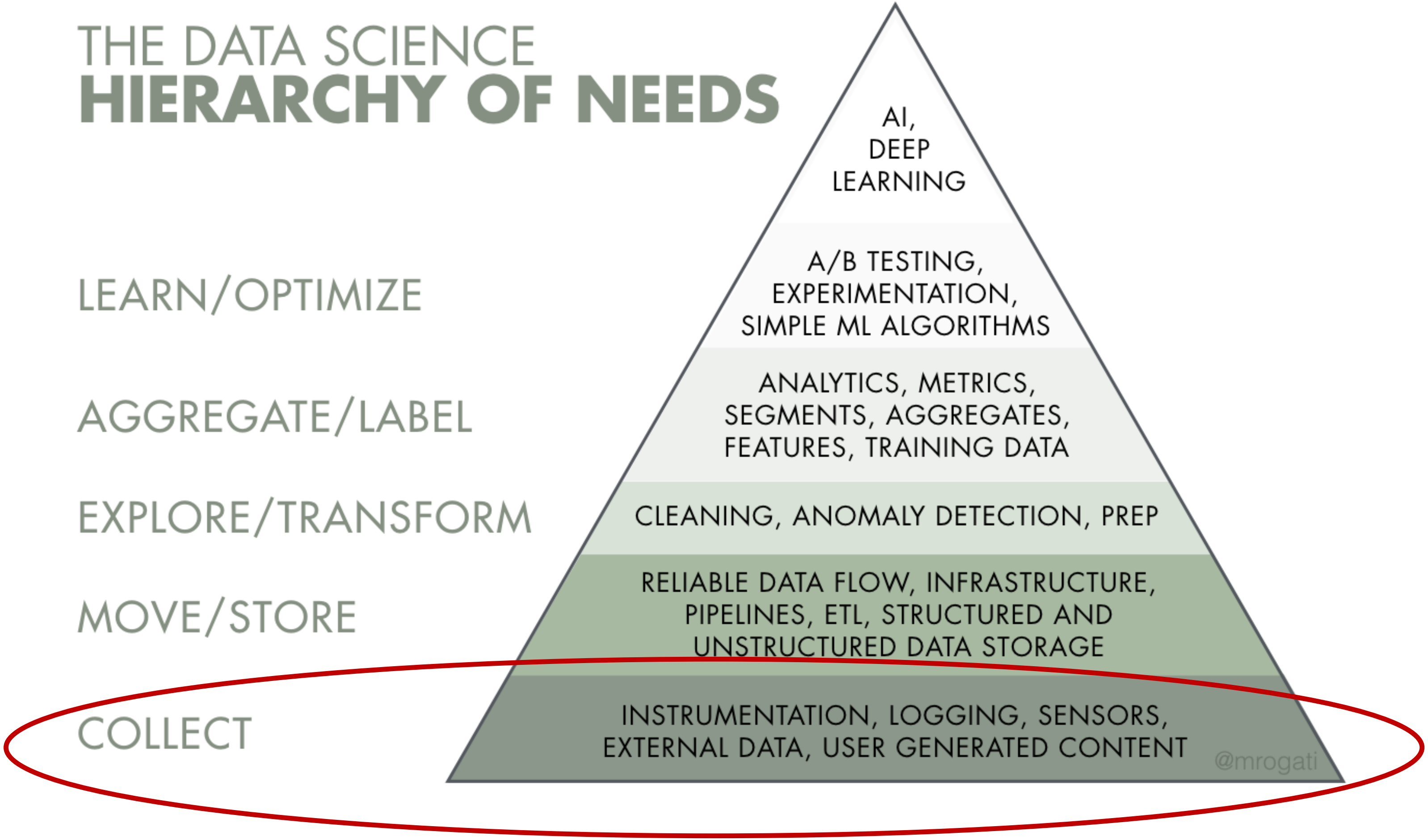
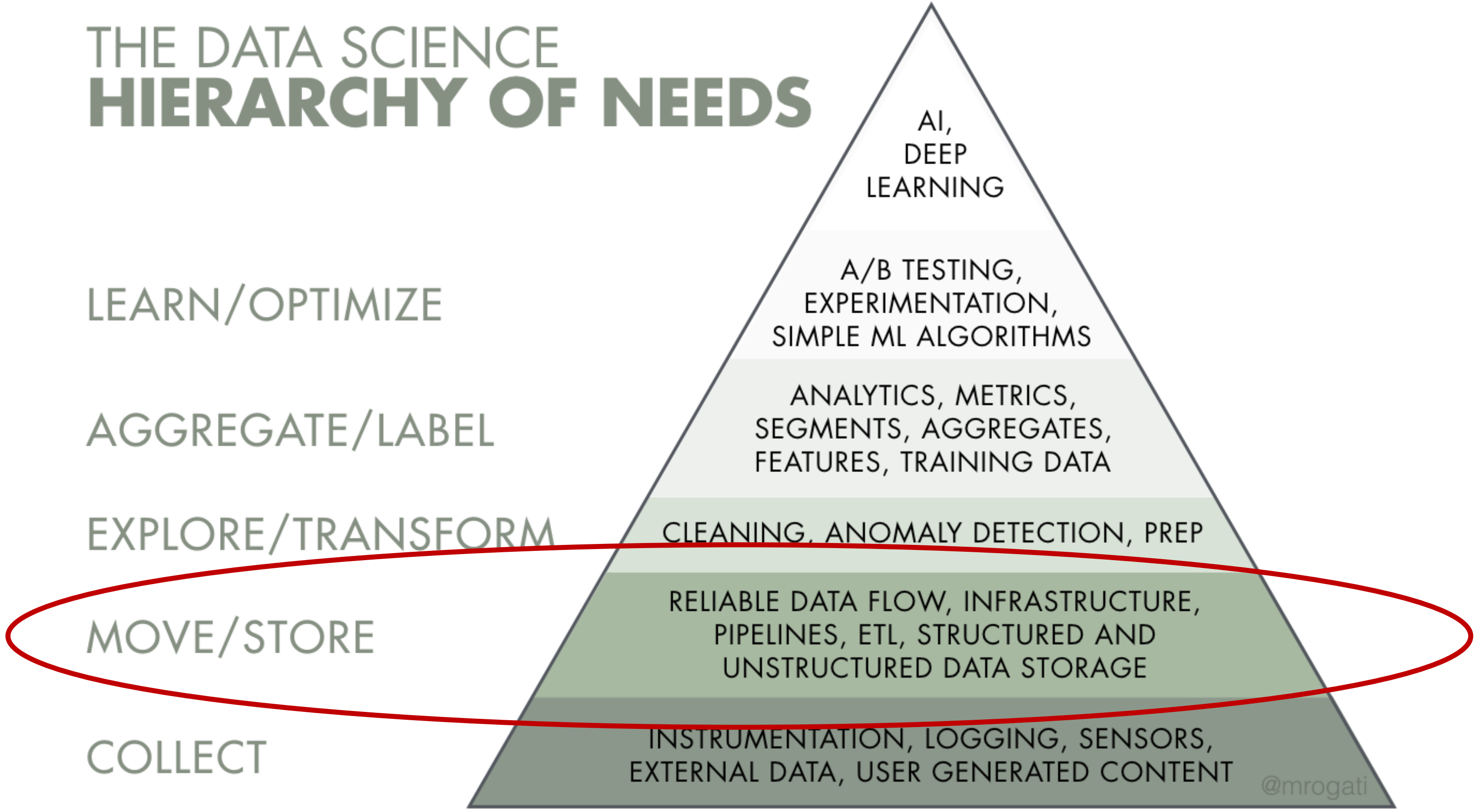
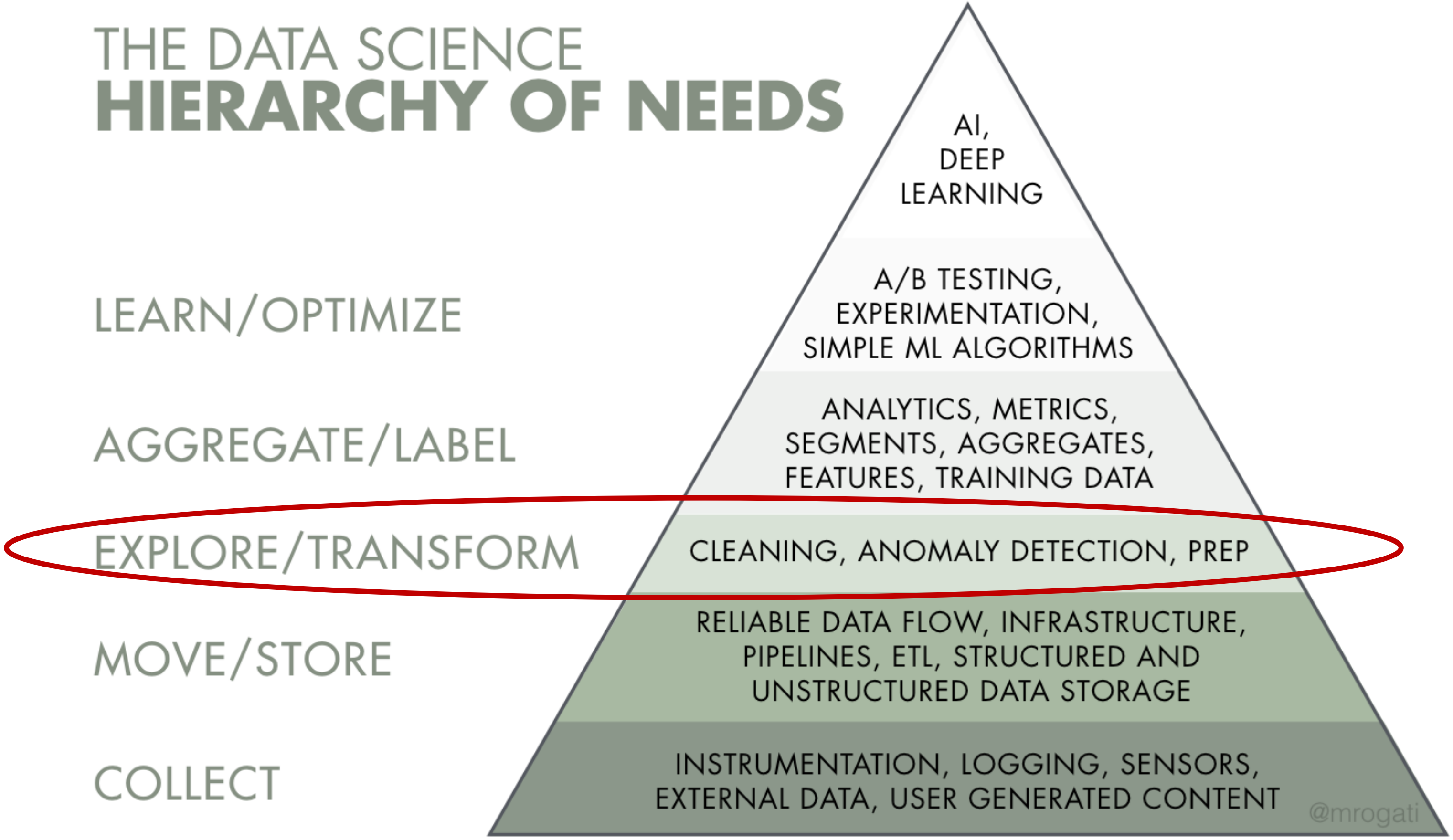
Data engineering - I



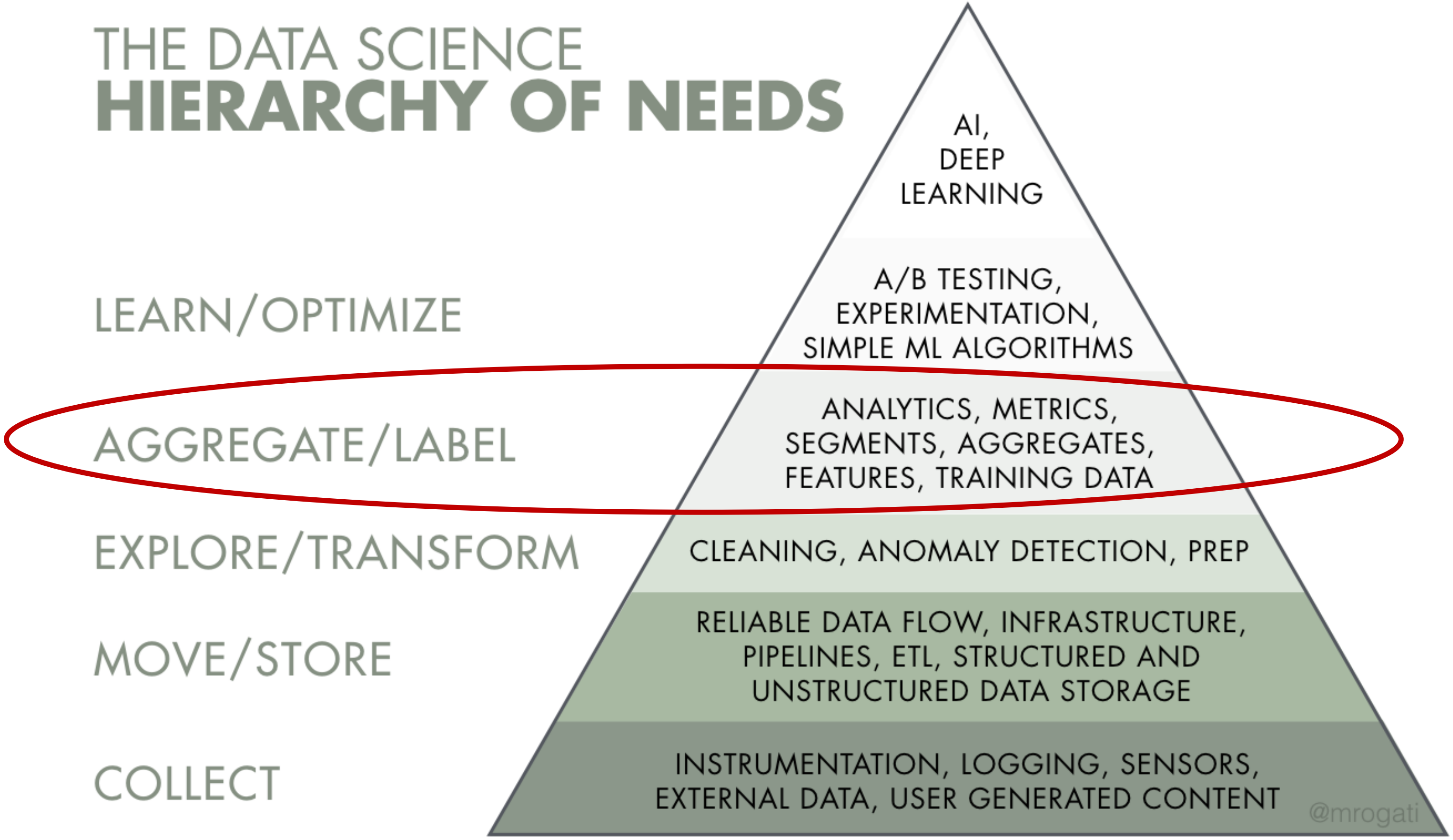
The AI Hierarchy of Needs (Monica Rogati, 2017)



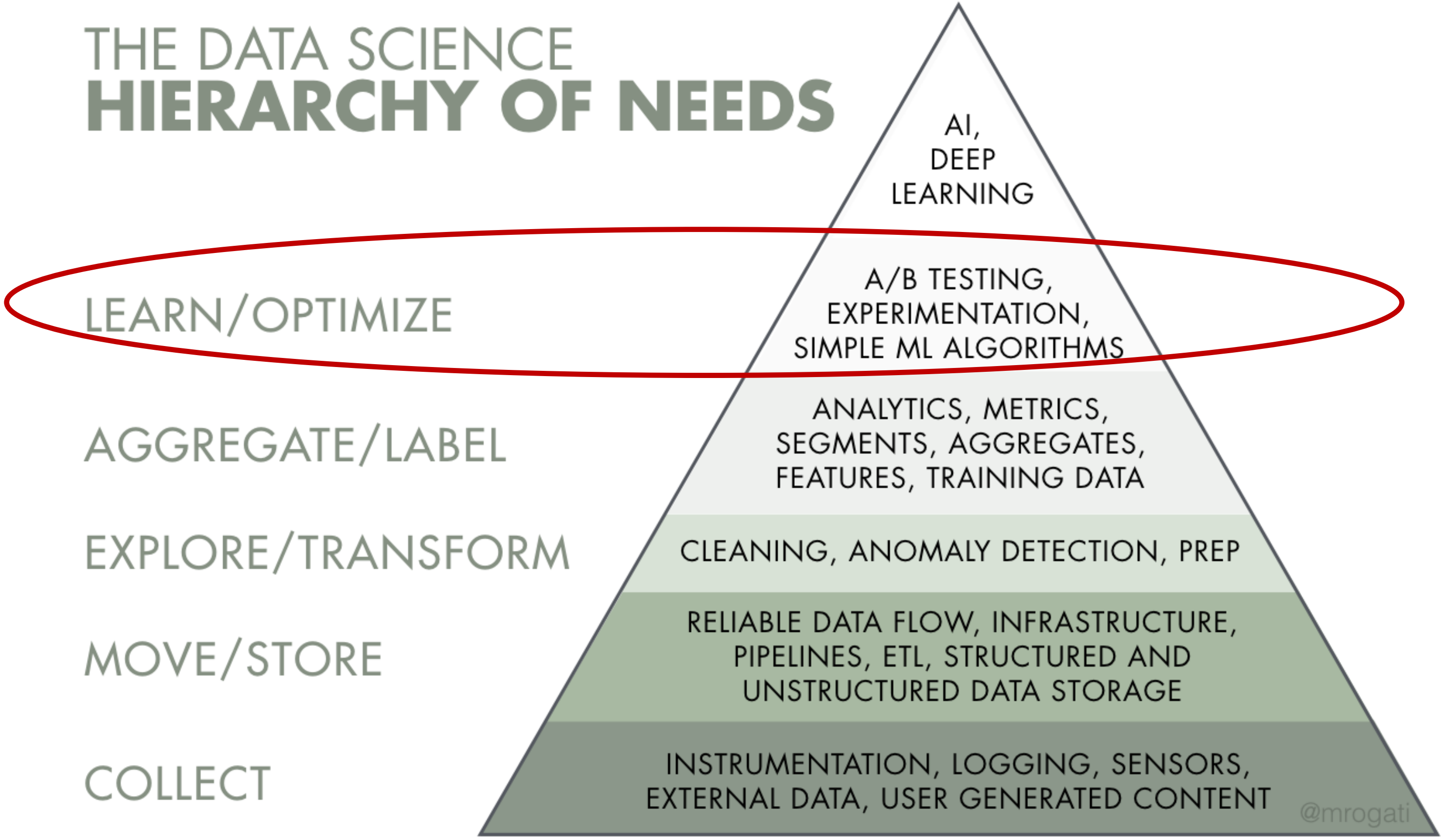
The AI Hierarchy of Needs (Monica Rogati, 2017)



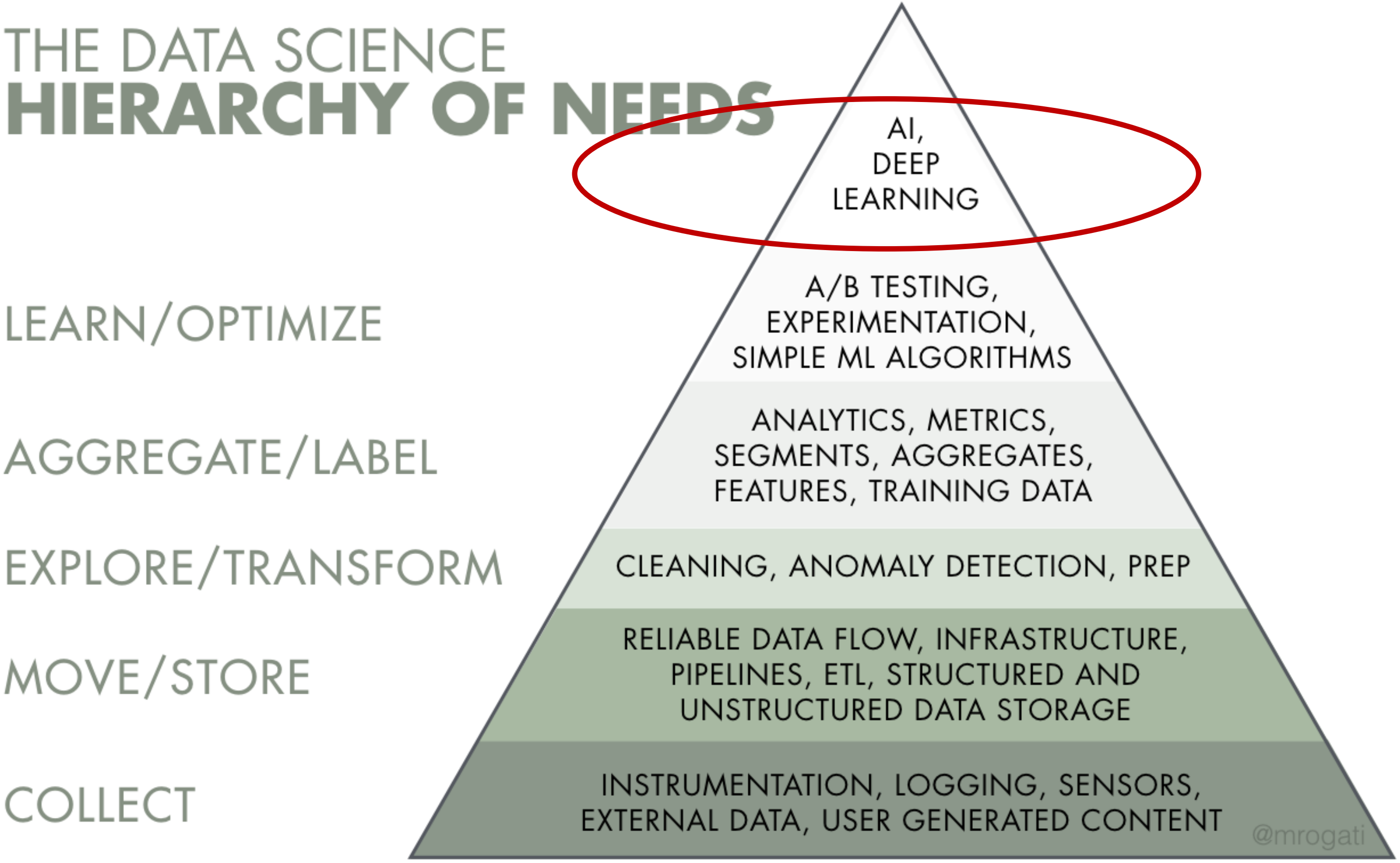
The AI Hierarchy of Needs (Monica Rogati, 2017)



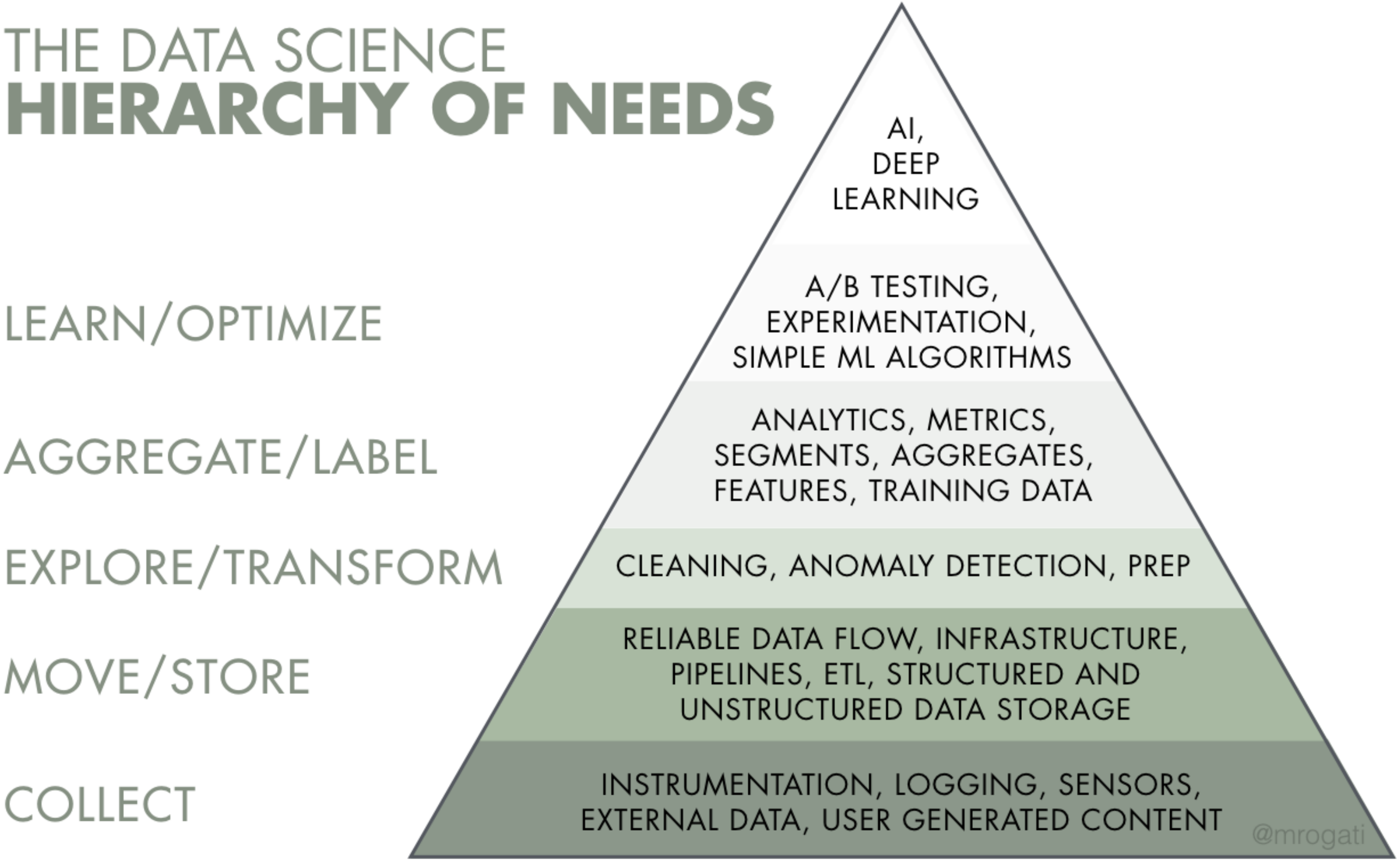
The AI Hierarchy of Needs (Monica Rogati, 2017)



The AI Hierarchy of Needs (Monica Rogati, 2017)



[The AI Hierarchy of Needs] (Monica Rogati, 2017)



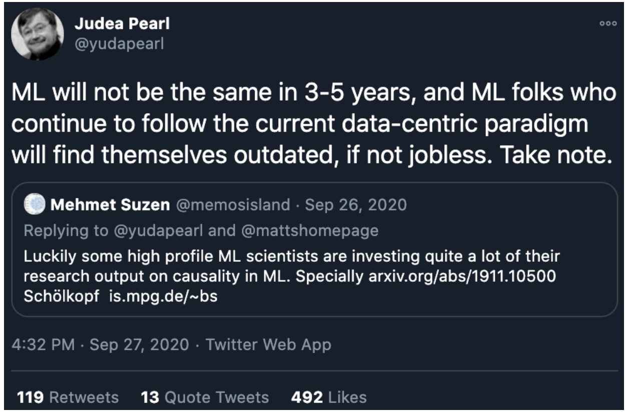
CAUTION! The data science hierarchy does not mean that you spend a long time to build disconnected, over-engineered infrastructure for a year.

The AI Hierarchy of Needs (Monica Rogati, 2017)

Mind

“Data is profoundly dumb.”

Judea Pearl, [Mind over data - The Book of Why]



“Huge computation and massive amount of data, … with simple learning device, … [create] incredibly bad learners. … Structure allows us to design systems that can learn more from less data.”

Chris Manning, [Deep Learning and Innate Priors]

Data

“General methods that leverage computation are ultimately the most effective, and by a large margin … Human-knowledge approach tends to complicate methods in ways that make them less suited to taking advantage of general methods leveraging computation.”
Richard Sutton, Bitter Lesson

“We don’t have better algorithms. We just have more data.”
Peter Norvig, The Unreasonable Effectiveness of Data

“Imposing structure requires us to make certain assumptions, which are invariably wrong for at least some portion of the data.”
Yann LeCun, Deep Learning and Innate Priors

Data is necessary.

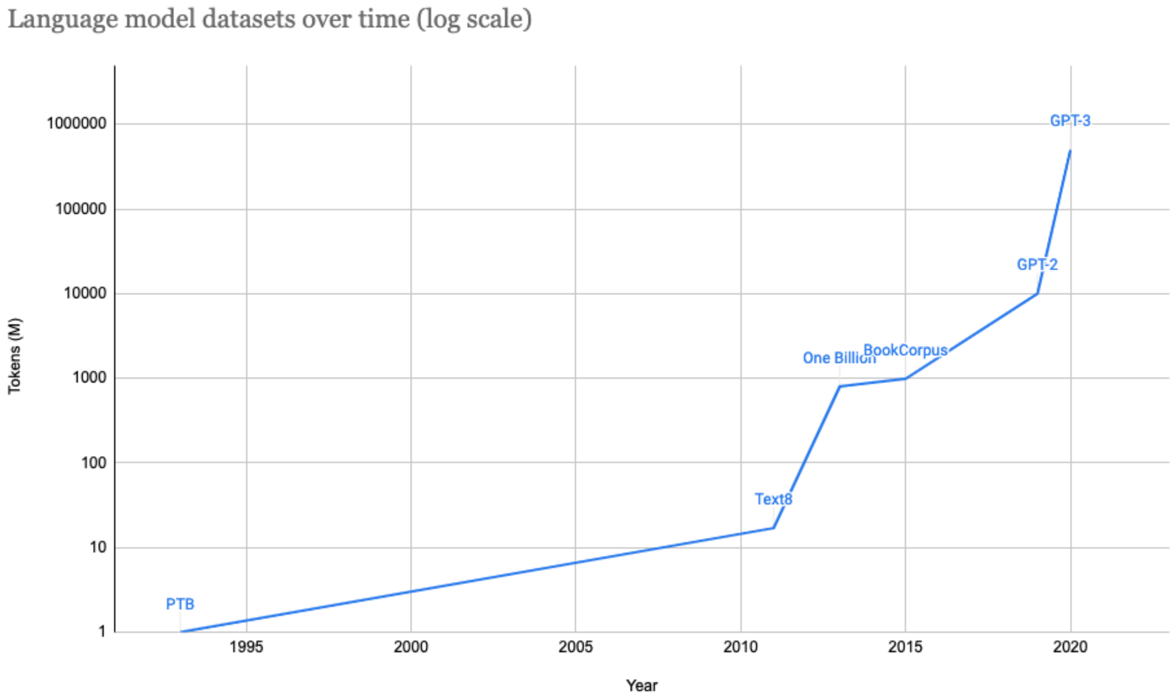
The debate is whether *finite** data is sufficient.

* If we had infinite data (and infinite memory), we could solve arbitrarily complex problems by just looking up the answers.

A lot of data ≠ infinite data.

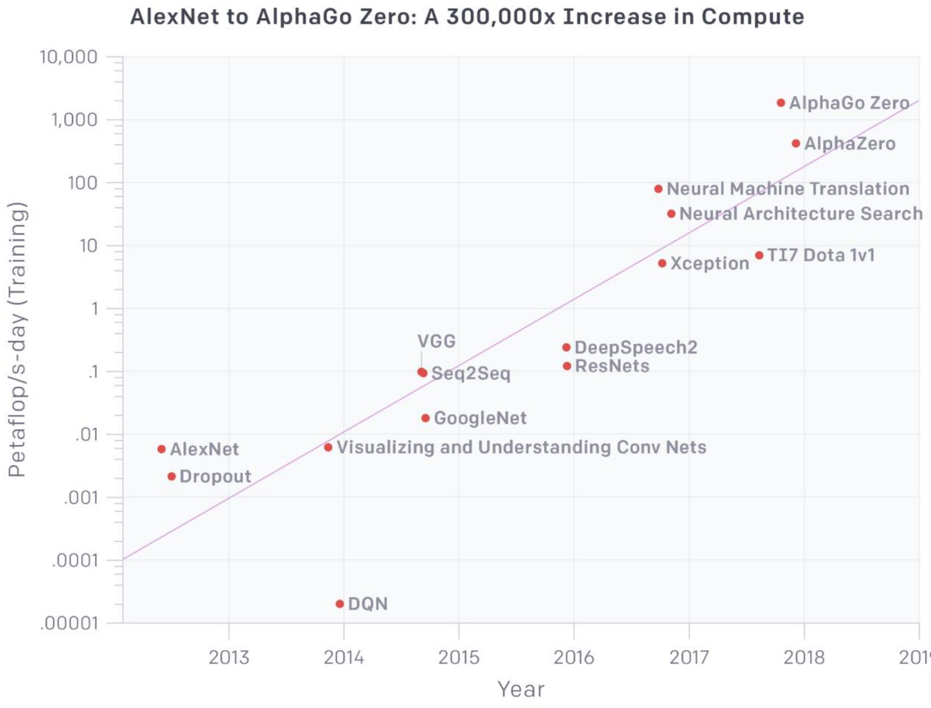
Datasets for language models

Dataset	Year	Tokens (M)
Penn Treebank	1993	1
Text8	2011	17
One Billion	2013	800
BookCorpus	2015	985
GPT-2 (OpenAI)	2019	10,000
GPT-3 (OpenAI)	2020	500,000



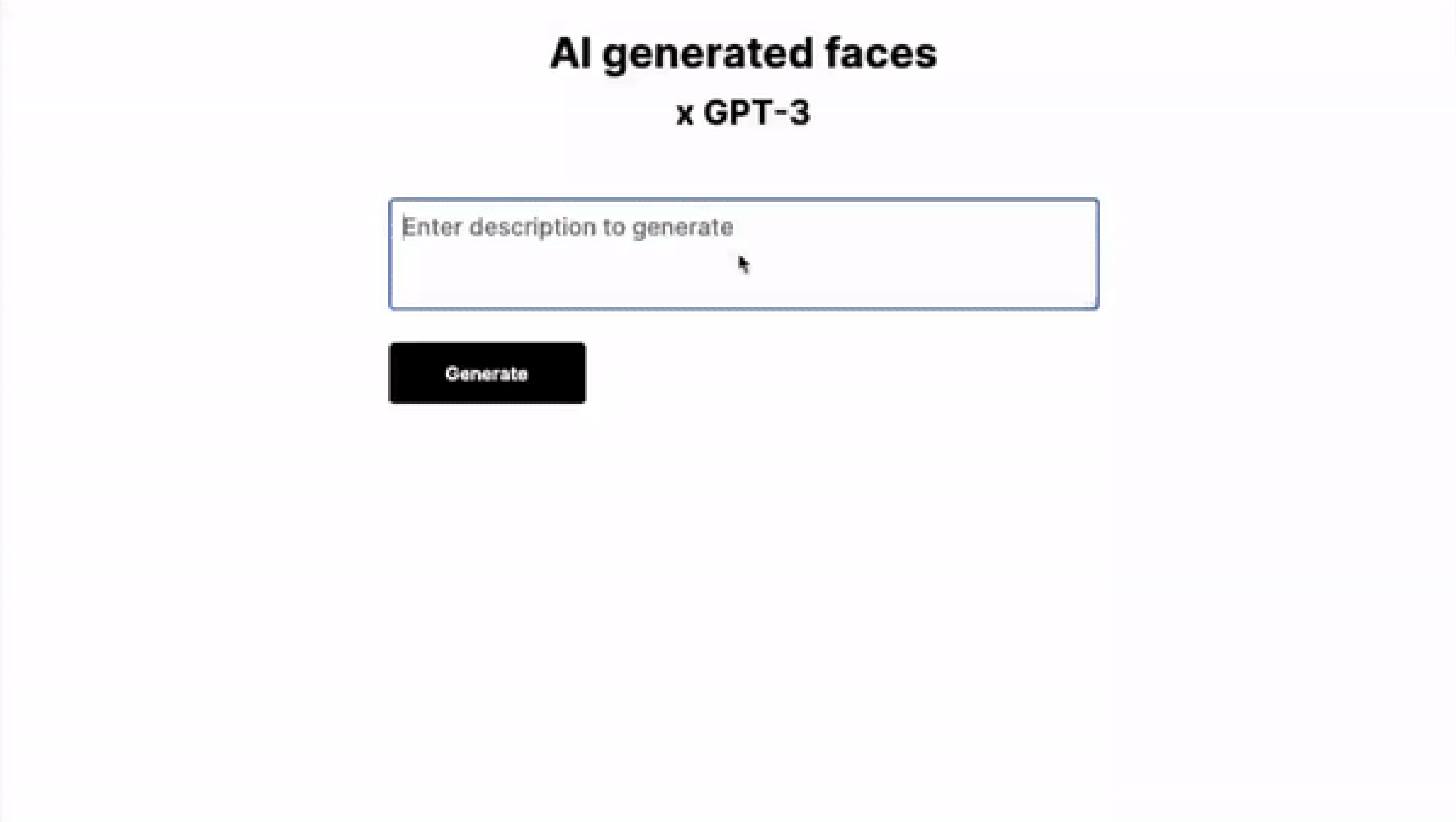
More data (generally) needs more compute

"amount of compute used in the largest AI training runs has doubled every 3.5 months"



AI and Compute (OpenAI 2018)

Deep Learning in NLP



Source: https://www.auxiliary.tools/

Data basics: formats to store

- How to store both data and labels?
    - `{‘image’: [[200,155,0], [255,255,255], ...], ‘label’: ‘car’, ‘id’: 1}`
- How to store a model?
- How to store any complex object?

Data basics: data serialization

Converting a data structure or object state into a format that can be stored or transmitted and reconstructed later

	Format	Binary/Text	Human-readable?	Example use cases
	JSON	Text	Yes	Everywhere
Row-based	CSV	Text	Yes	Everywhere
Column-based	Parquet	Binary	No	Hadoop, Amazon Redshift
	Avro	Binary primary	No	Hadoop
	Protobuf	Binary primary	No	Google, TensorFlow (TFRecord)
	Pickle	Text, binary	No	Python, PyTorch serialization

Data basics: column-based vs. row-based

Column-based:
- Stored and retrieved column-by-column
- Good for accessing features

Row-based:
- Stored and retrieved row-by-row
- Good for accessing samples

	Column 1	Column 2	Column 3
Sample 1	…	…	…
Sample 2	…	…	…
Sample 3	…	…	…

Data basics: text vs. binary files

Benefits of column-based:

flexible data access: can access only columns required

	Column 1	Column 2	Column 3
Sample 1	...	...	...
Sample 2	...	...	...
Sample 3	...	...	...

Data basics: column-based vs. row-based

Pandas DataFrame: column-based

- accessing a row much slower than accessing a column and NumPy

```
# Get the column `date`, 1000 loops
%timeit -n1000 df["Date"]

# Get the first row, 1000 loops
%timeit -n1000 df.iloc[0]
```

```
1.78 µs ± 167 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
145 µs ± 9.41 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

NumPy ndarray: row-based by default

- can specify to be column-based

```
df_np = df.to_numpy()
%timeit -n1000 df_np[0]
%timeit -n1000 df_np[:,0]
```

```
147 ns ± 1.54 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
204 ns ± 0.678 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```


https://github.com/chiphuyen/just-pandas-things

Data basics: text vs. binary files

	Text files	**Binary files**
Examples	CSV, JSON	Parquet
Pros	Human readable	Compact
To store the number 1000000?	7 characters -> 7 bytes	If stored as int32, only 4 bytes

Data basics: text vs. binary files

```
In [2]: df = pd.read_csv("data/interviews.csv")
        df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17654 entries, 0 to 17653
Data columns (total 10 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Company     17654 non-null  object
 1   Title       17654 non-null  object
 2   Job         17654 non-null  object
 3   Level       17654 non-null  object
 4   Date        17652 non-null  object
 5   Upvotes     17654 non-null  int64
 6   Offer       17654 non-null  object
 7   Experience  16365 non-null  float64
 8   Difficulty  16376 non-null  object
 9   Review      17654 non-null  object
dtypes: float64(1), int64(1), object(8)
memory usage: 1.3+ MB

In [3]: Path("data/interviews.csv").stat().st_size

Out[3]: 14200063

In [4]: df.to_parquet("data/interviews.parquet")
        Path("data/interviews.parquet").stat().st_size

Out[4]: 6211862
```

Data basics: text vs. binary files

- JSON is a very common human-readable format.
- Supported by many programming languages support it.
- Its key-value pair paradigm allows you to structure your data as you want.

```
{
  "firstName": "Boatie",
  "lastName": "McBoatFace",
  "isVibing": true,
  "age": 12,
  "address": {
    "streetAddress": "12 Ocean Drive",
    "city": "Port Royal",
    "postalCode": "10021-3100"
  }
}
```

```
{
  "text": "Boatie McBoatFace, aged 12, is vibing, at 12 Ocean Drive,  Port Royal, 10021-3100"
}
```

Data basics: column-based vs. row-based

Column-based:
- Stored and retrieved column-by-column
- Good for accessing features
- Good for using data for analytic tasks

Row-based:
- Stored and retrieved row-by-row
- Good for accessing samples
- Good for managing transactions as they come in

	Column 1	Column 2	Column 3
Sample 1	…	…	…
Sample 2	…	…	…
Sample 3	…	…	…

OnLine Transaction Processing vs. **OnLine Analytical Processing**

OLTP: OnLine Transaction Processing

- How to handle a large number of small transactions?
  - e.g. ordering food, ordering rides, buying things online, transferring money
- Requirements:
  - Atomicity: all the steps in a transaction fail or succeed as a group
    - If payment fails, don't assign a driver
  - Isolation: concurrent transactions happen as if sequential
    - Don't assign the same driver to two different requests that happen at the same time
  - Fast response time (e.g. milliseconds)
- Operations:
  - INSERT, UPDATE, DELETE

See ACID: Atomicity, Consistency, Isolation, Durability

Row

```
INSERT INTO RideTable(RideID, Username, DriverID, City, Month, Price)
VALUES ('10', 'memelord', '3932839', 'Stanford', 'July', '20.4');
```

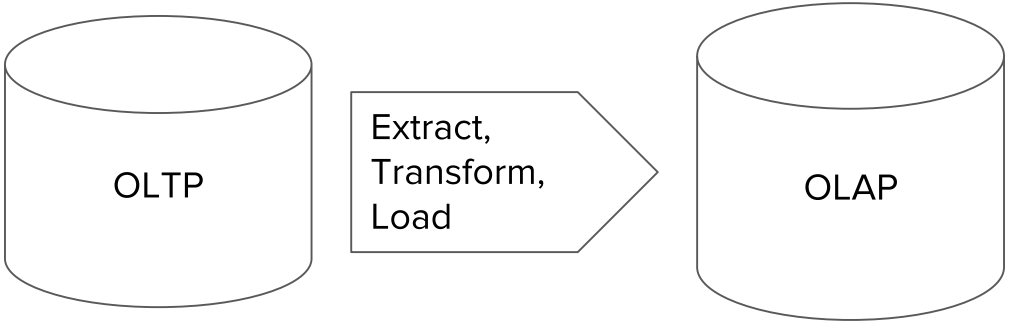
OLAP: OnLine Analytical Processing

- How to get aggregated information from a large amount of data?
  - e.g. what's the average ride price last month for riders at Stanford?
- Requirements:
  - Can handle complex queries on large volumes of data
  - Okay response time (seconds, minutes, even hours)
- Operations:
  - Mostly SELECT

Column

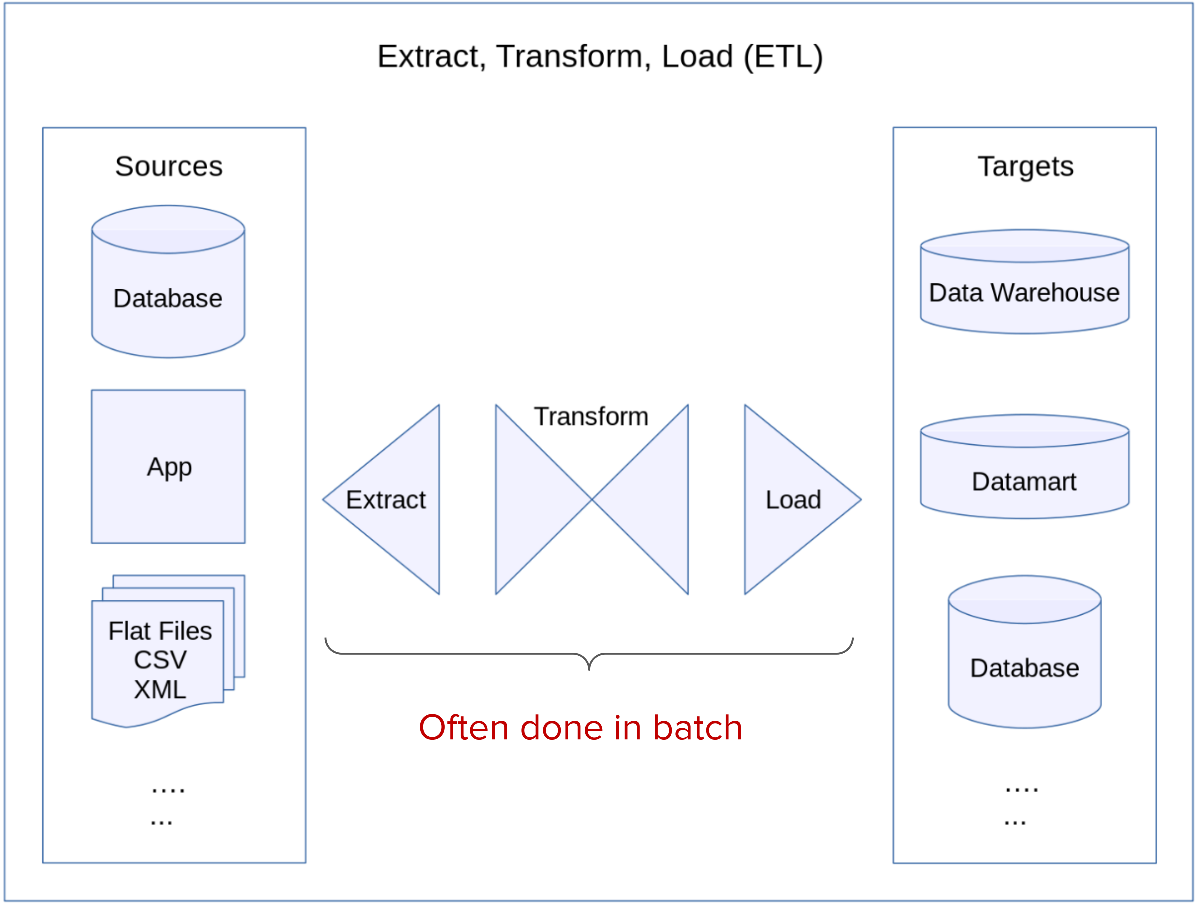
```
SELECT AVG(Price)
FROM RideTable
WHERE City = 'Stanford' AND Month = 'July';
```

Data basics: ETL (Extract, Transform, Load)



Transform: the meaty part

- Cleaning
- Validating
- Transposing
- Deriving values
- Joining from multiple sources
- Deduplicating
- Splitting
- Aggregating, etc.



https://commons.wikimedia.org/wiki/File:Extract,_Transform,_Load_Data_Flow_Diagram.svg

Data basics: structured vs. unstructured data

Structured	Unstructured
Schema clearly defined	Whatever
Easy to search and analyze	Fast arrival (e.g. no need to clean up first)
Can only handle data with specific schema	Can handle data from any source
Schema changes will cause a lot of trouble	No need to worry about schema changes
Data warehouse	Data lake

Data basics: structured vs. unstructured data

Structured	Unstructured
Schema clearly defined	Whatever
Easy to search and analyze	Fast arrival (e.g. no need to clean up first)
Can only handle data with specific schema	Can handle data from any source
Schema changes will cause a lot of trouble	No need to worry about schema changes
Data warehouse	Data lake

Structured -> unstructured -> structured

want more flexibility tools & infra standardized

ETL -> ELT -> ETL

Slide from: CS 329 | Chip Huyen | cs329s.stanford.edu

Real time pipeline: ride-sharing example

Real-time pipeline: a pipeline that can process data, input it into model, and return a prediction in real-time

To detect whether a transaction is fraud, need features from:

- this transaction
- user's recent transactions (e.g. 7 days)
- credit card's recent transactions
- recent in-app frauds

Real time pipeline: ride-sharing example

To detect whether a transaction is fraud, need features from:

- The current transaction
- user's recent transactions (e.g. 7 days)
- credit card's recent transactions
- recent in-app frauds
- etc.

Stream storage

APACHE kafka®

972 companies reportedly use **Kafka** in their tech stacks, including **Uber**, **Spotify**, and **Shopify**.

Uber | Spotify | Shopify | Slack | Robinhood | LaunchDarkly | Nubank | The New York Times | Alibaba Travels

amazon Kinesis

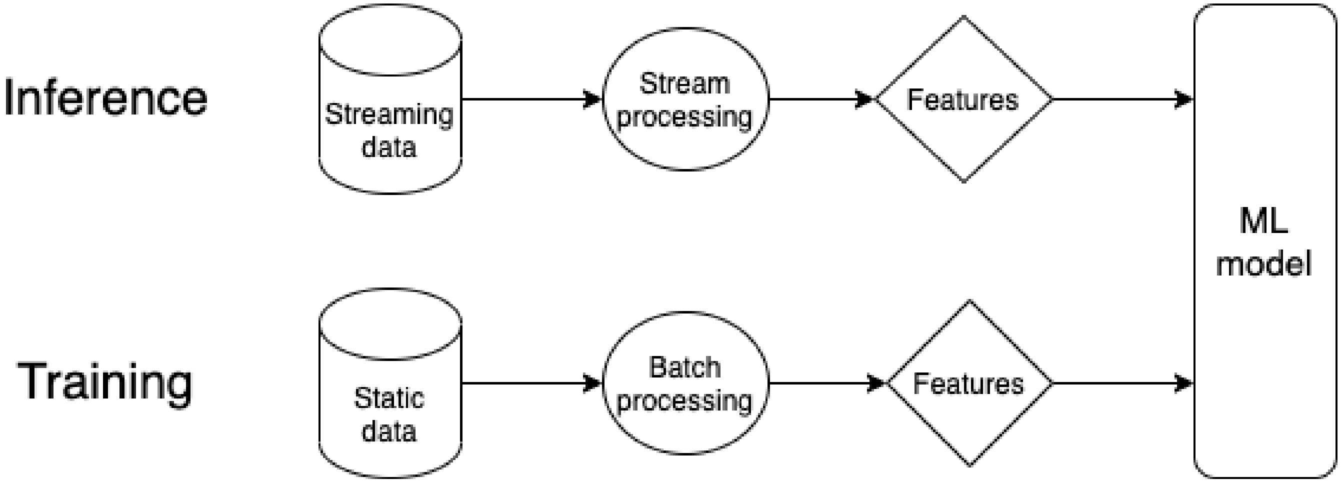
233 companies reportedly use **Amazon Kinesis** in their tech stacks, including **Amazon**, **Instacart**, and **Lyft**.

Amazon | Instacart | Lyft | LaunchDarkly | Accenture | Figma | trivago | N26 | Pratilipi

Need static + dynamic features

Static data	**Streaming data**
CSV, PARQUET, etc.	Kafka, Kinesis, etc.
Bounded: know when a job finishes	Unbounded: never finish
Static features: ● age, gender, job, city, income ● when account was created ● rating	Dynamic features ● locations in the last 10 minutes ● recent activities
Can be processed in batch ● e.g. SQL, MapReduce	Processed as events arrive ● e.g. Apache Flink, Samza

One model, two pipelines



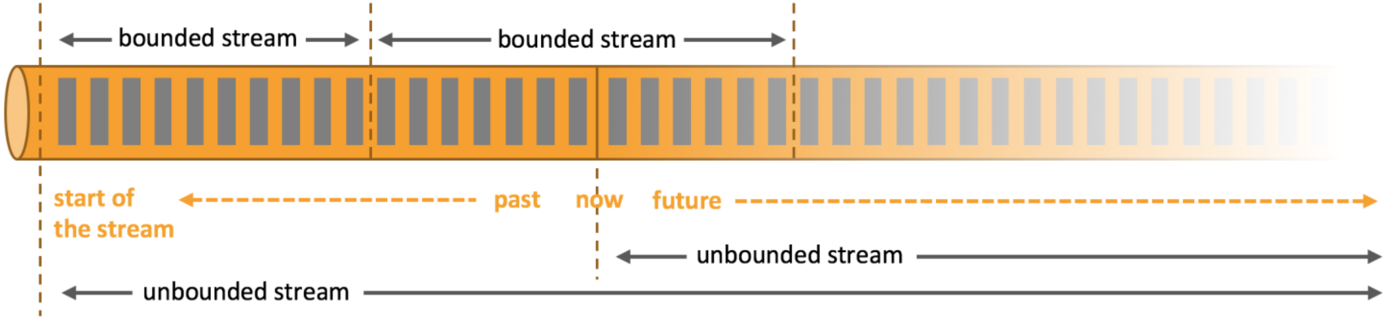
A common source of errors in production

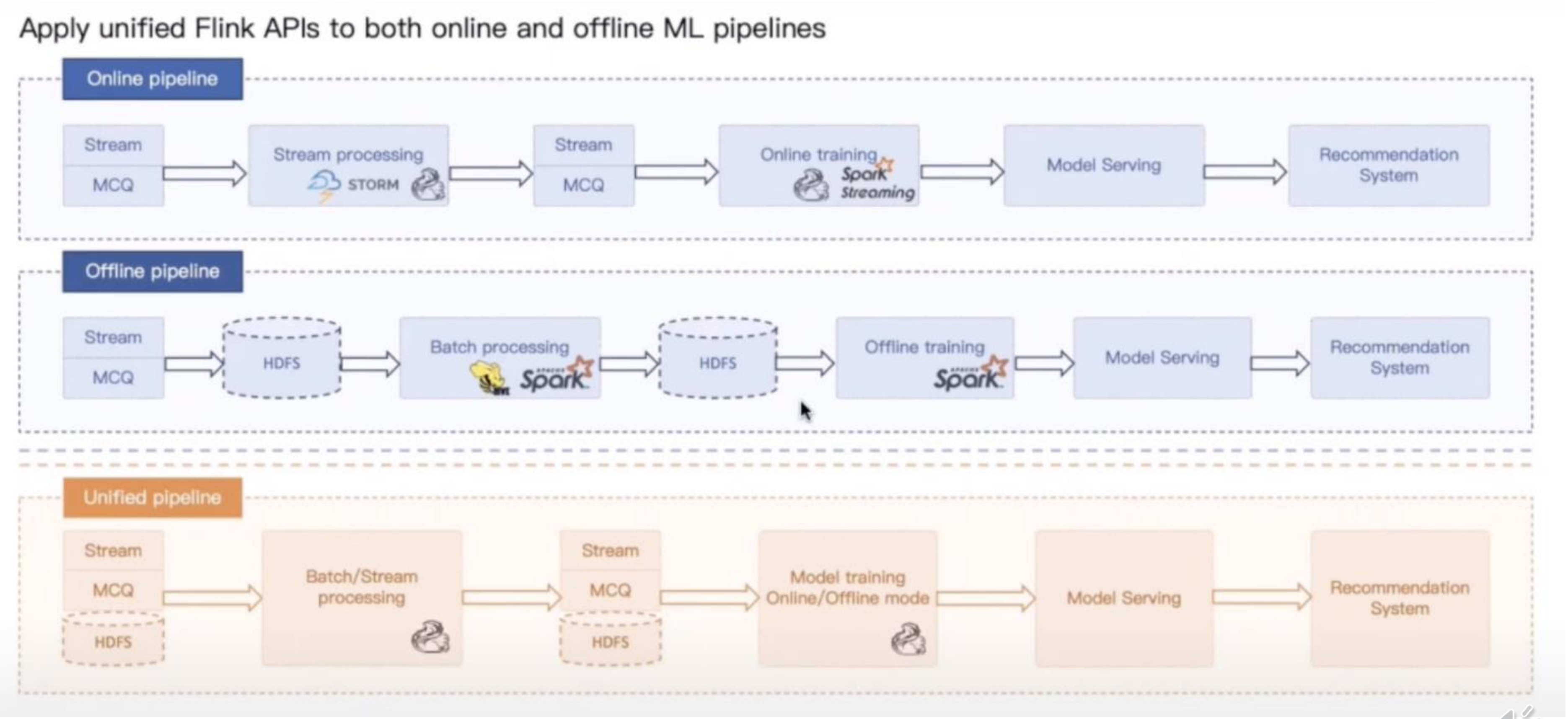
Apache Flink

- Apache Flink is a framework and distributed processing engine for stateful computations over *unbounded and bounded* data streams.
- Flink has been designed to run in *all common cluster environments*, perform computations at *in-memory speed* and at *any scale*.

Apache Flink

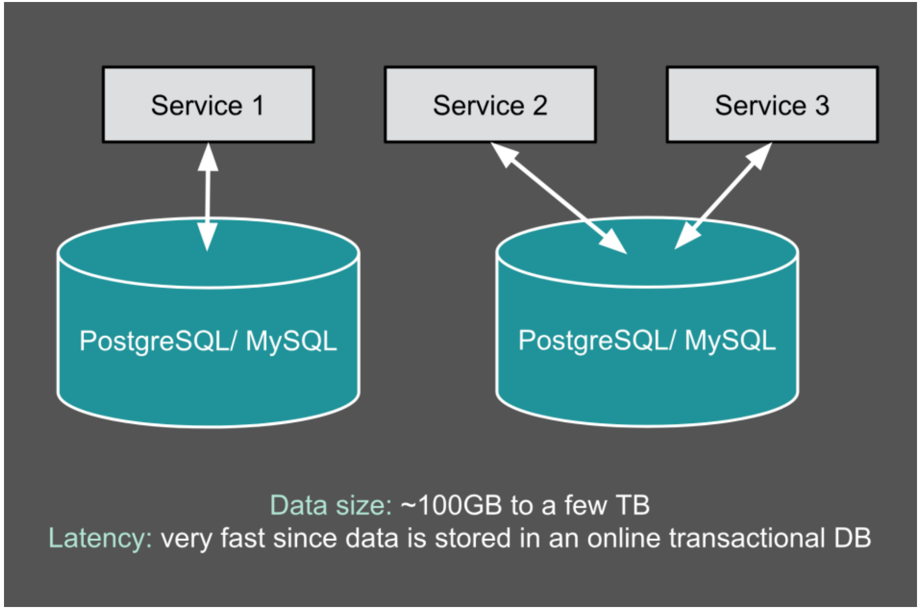
- Data can be processed as *unbounded* or *bounded* streams.
- **Unbounded streams** have a start but no defined end.
    - They do not terminate and provide data as it is generated.
    - Unbounded streams must be continuously processed, i.e., events must be promptly handled after they have been ingested.
    - It is not possible to wait for all input data to arrive because the input will not be complete at any point in time.
    - Processing unbounded data often requires that events are ingested in a specific order, such as the order in which events occurred, to be able to reason about result completeness.
- **Bounded streams** have a defined start and end.
    - Bounded streams can be processed by ingesting all data before performing any computations.
    - Ordered ingestion is not required to process bounded streams because a bounded data set can always be sorted.
    - Processing of bounded streams is also known as batch processing.





Machine learning with Flink in Weibo (Qian Yu, QCon 2019)

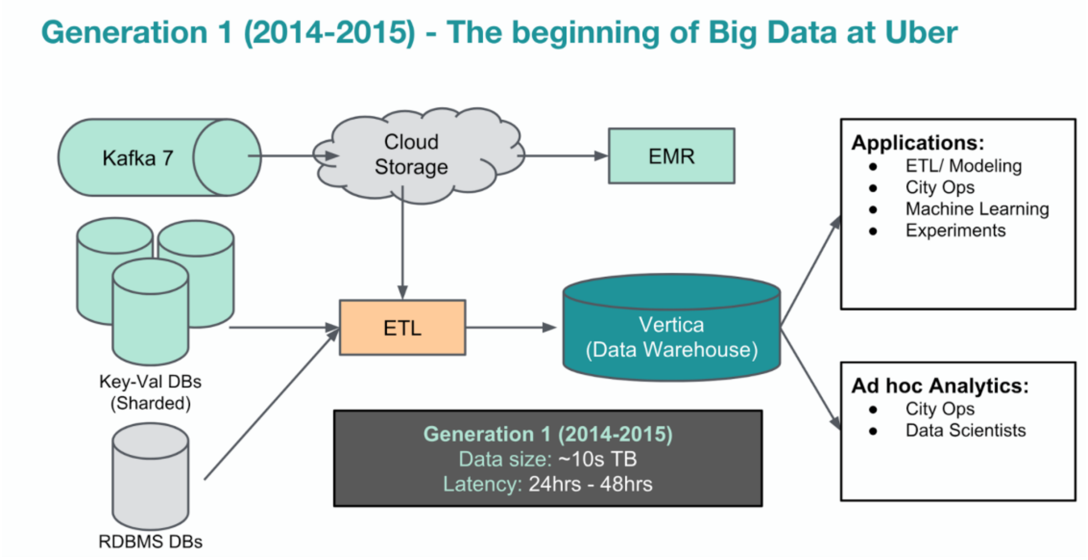
Data pipeline: case study with Uber



Before 2014, the total amount of data stored at Uber was small enough to fit into a few traditional OLTP databases.
There was no global view of the data, and data access was fast since each database was queried directly.

Uber's Big Data Platform: 100+ Petabytes with Minute Latency (Reza Shiftehfar, Uber Engineering Blog 2018)

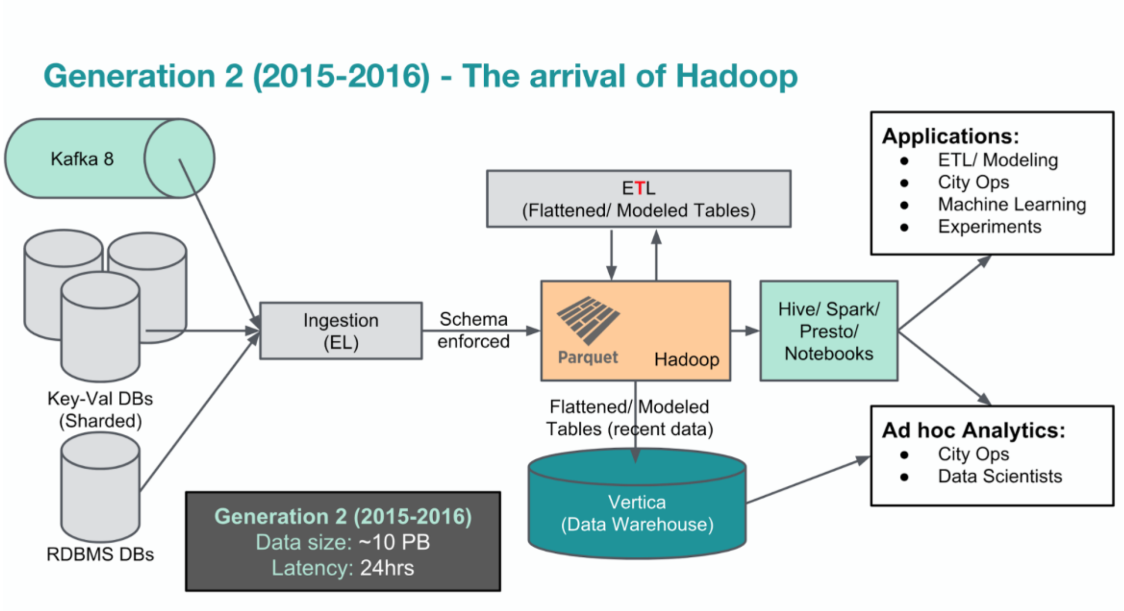
Data pipeline: case study with Uber



Before 2014, the total amount of data stored at Uber was small enough to fit into a few traditional OLTP databases.
There was no global view of the data, and data access was fast since each database was queried directly.

Uber's Big Data Platform: 100+ Petabytes with Minute Latency (Reza Shiftehfar, Uber Engineering Blog 2018)

Data pipeline: case study with Uber



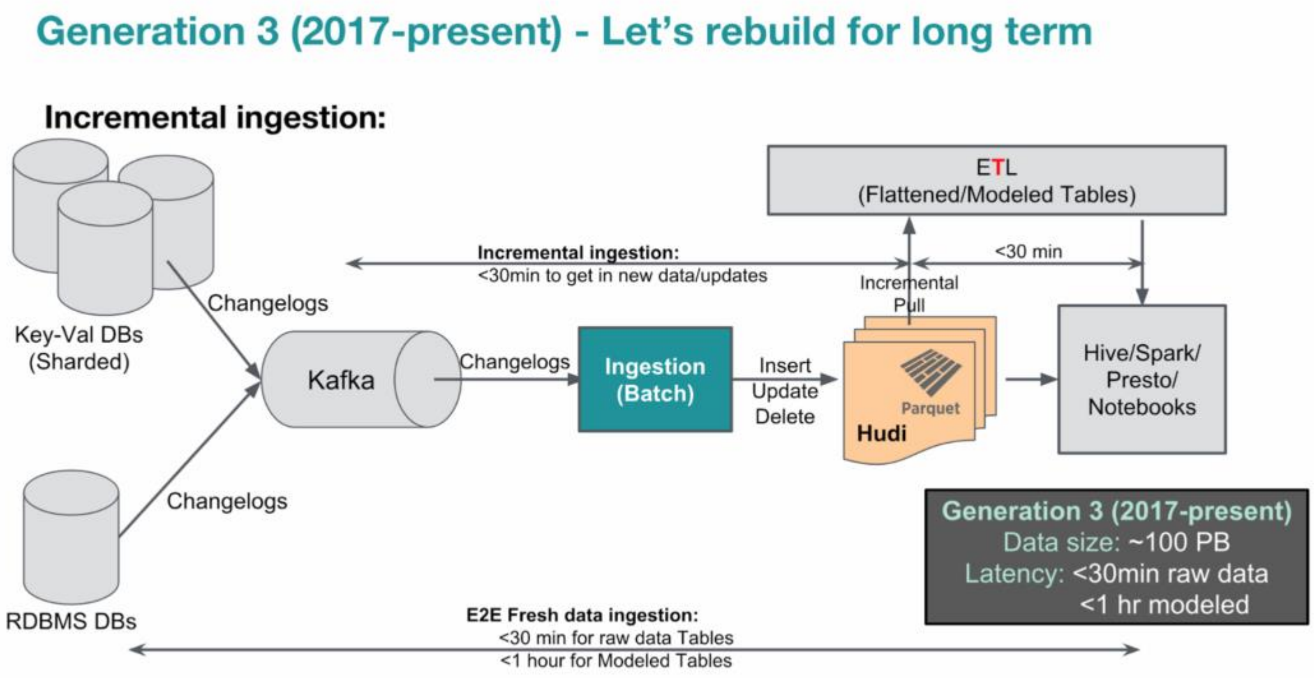
The second generation of our Big Data platform leveraged Hadoop to enable horizontal scaling.
Incorporating technologies such as Parquet, Spark, and Hive, tens of petabytes of data was ingested, stored, and served.

Uber's Big Data Platform: 100+ Petabytes with Minute Latency (Reza Shiftehfar, Uber Engineering Blog 2018)

Data pipeline: case study with Uber

e.g.
JSON blobs

e.g.
transactions



Uber's Big Data Platform: 100+ Petabytes with Minute Latency (Reza Shiftehfar, Uber Engineering Blog 2018)