DATA+AI SUMMIT 2022

Building production-ready recommender systems with feature stores

bit.ly/feast-recsys-talk

Danny Chiao

Engineering Lead at Tecton / Feast (former lead at Google)

ORGANIZED BY databricks

Agenda

- Background
  - Recommender systems intro
  - What is a feature store / Feast?
- Recommender systems challenges
  - How teams typically run recommender systems
  - Optimizing performance / cost
  - Correctness
- Deploying Feast
- Key takeaways

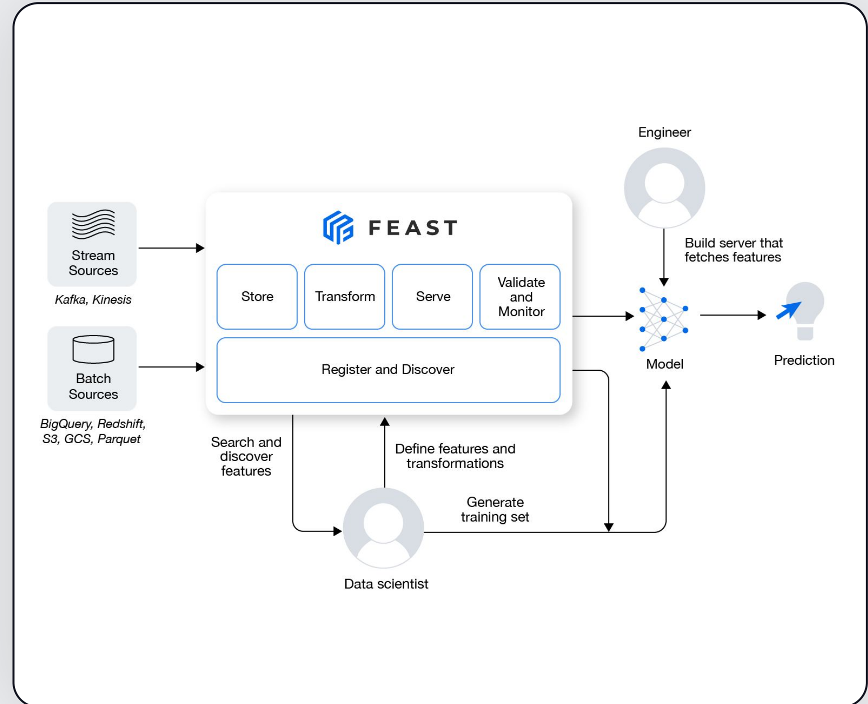
Background

Recommender systems

- **Use cases:** e-commerce, media streaming, social, ride-hailing, biomedical, etc
- **Who:** data engineers, data scientists, platform engineers
- **Trend:** Batch predictions → online predictions

What is Feast (FEAture STore)?

- A component to manage E2E lifecycle of a feature, including transformations and serving
- Helps ML platform teams build a platform to democratize feature engineering
- Manages ML lineage & metadata
- Generates training data
- Encourages feature re-use



Recommender system challenges

Batch recommender systems

Precompute recommendations for all users + load at request time

What is the most popular grocery item in NY?

Popularity model

(baseline)

Easiest, most interpretable

What items do other users like me buy?

Linear methods

(e.g. KNN, SVD, SLIM, LightFM)

Easy + ok interpretable, more complex feature engineering

How likely am I to buy this item?

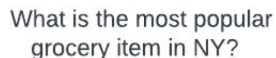
Deep learning

(e.g. rank candidate items)

Complex + not very interpretable, less complex feature engineering

Moving more online

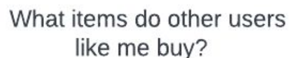
Moving online doesn't necessarily need a lot of new infrastructure



Popularity model

(baseline)

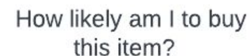
Need: fresh sets of most popular items



Linear methods

(e.g. KNN, SVD, SLIM, LightFM)

Need: fresh user x item interaction histories



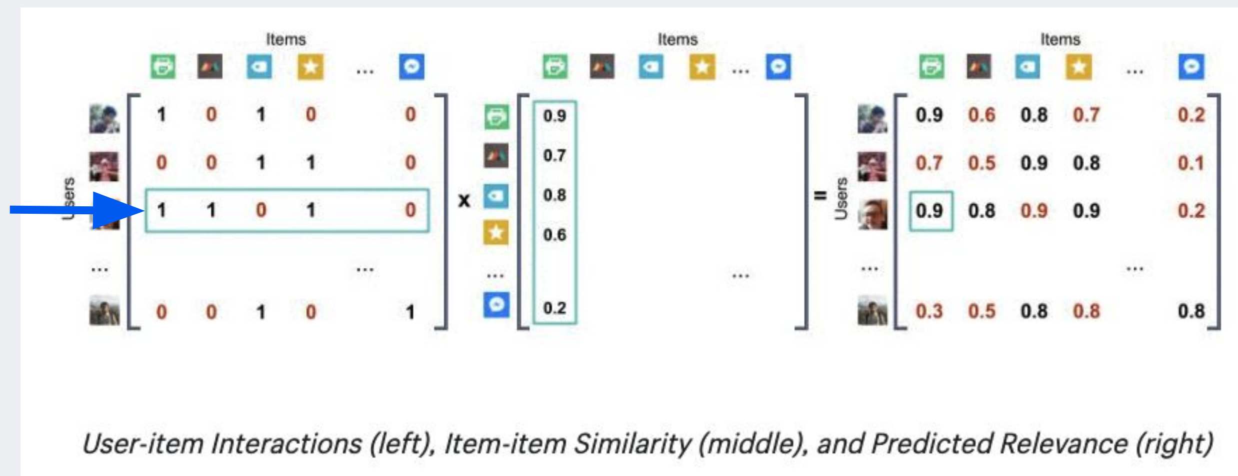
Deep learning

(e.g. rank candidate items)

Need: fresh features for users + items AND request time data

Moving more online

At serving time, need a **fresh user history vector** to get started with online inference (+ maybe resolving cold start problems)



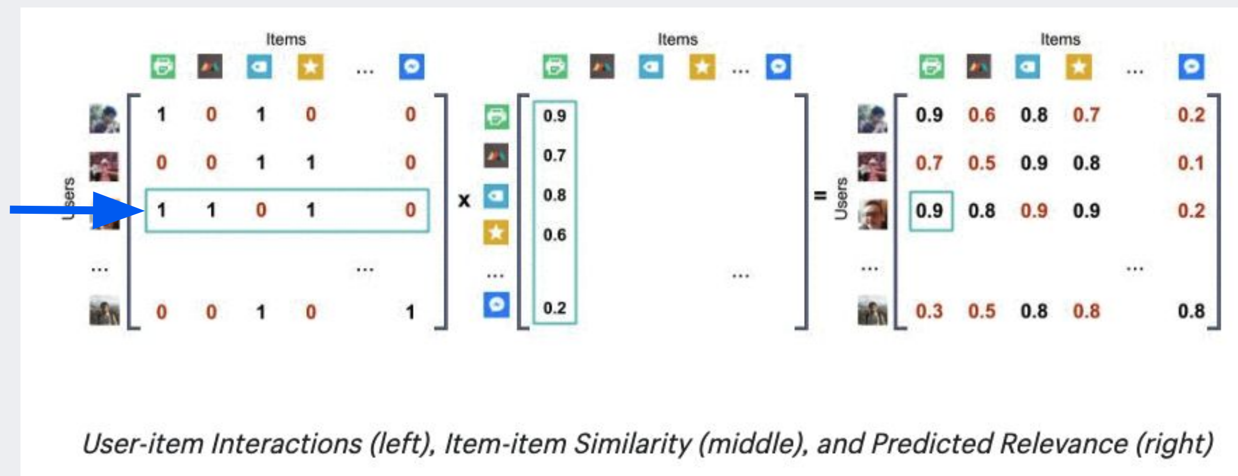
User-item Interactions (left), Item-item Similarity (middle), and Predicted Relevance (right)

Source: https://shopify.engineering/how-shopify-uses-recommender-systems-to-empower-entrepreneurs

Moving more online

At serving time, need a **fresh user history vector** to get started with online inference (+ maybe resolving cold start problems)

ALSO: Can use new meaningful features that rely on data available at <u>request time</u> (e.g. session data, timestamp of request, location of request, etc)



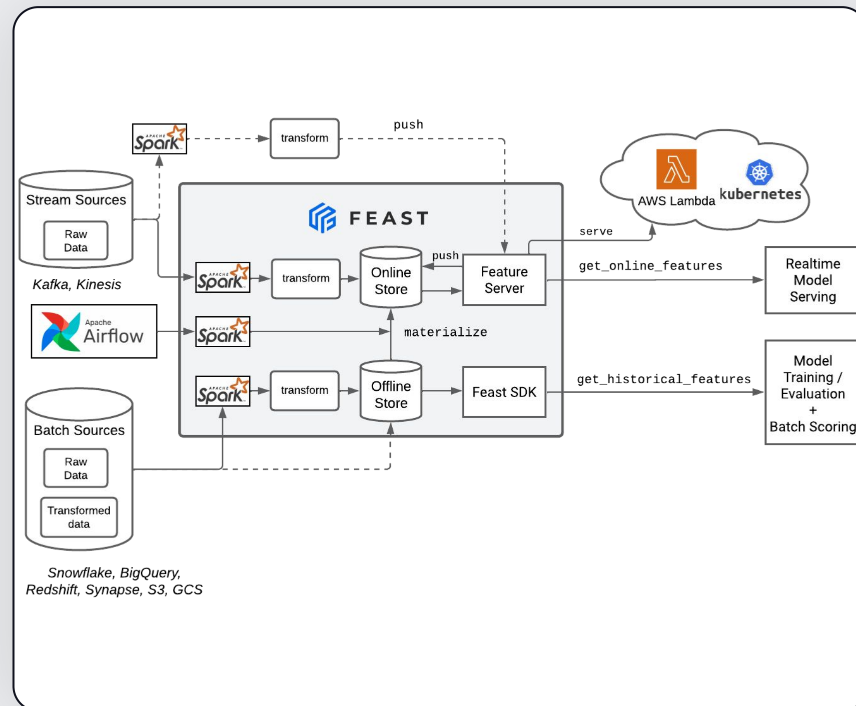
User-item Interactions (left), Item-item Similarity (middle), and Predicted Relevance (right)

Source: https://shopify.engineering/how-shopify-uses-recommender-systems-to-empower-entrepreneurs

Examples of where Feast fits in

Generating fresh online features

- Unifying batch + stream sources
  - low latency online retrieval (for online inference)
  - historical retrieval (for training dataset generation & batch scoring)
- Abstracting away data model for writing and reading into the low latency online store



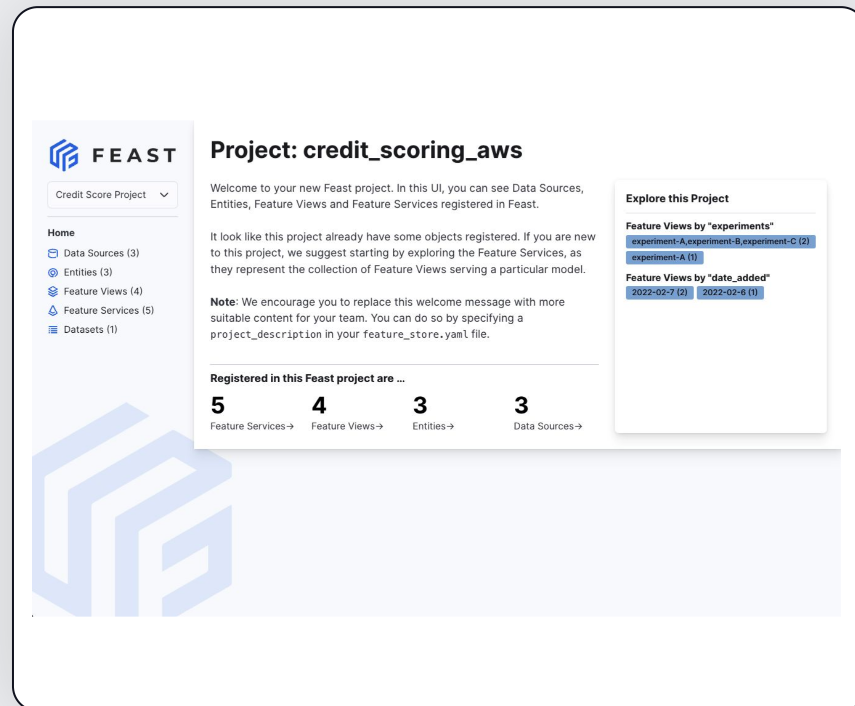
Examples of where Feast fits in

Re-using features

- ```
  store.get_historical_features(
    features=[
      "fv:time_since_last_purchase"]
    ...)
  ```
- ```
  store.get_online_features(
    features=[
      "fv:time_since_last_purchase"]
    ...)
  ```

Model versioning

- ```
  store.get_X_features(features=store.get_feature_service("ranking_model_v2"))
  ```


  ```

Examples of where Feast fits in

DS author production-ready features

- Iterate quickly and reduce training / serving skew
- On demand features
  - Combining entity values, request data, batch (pre-computed) features, and streaming features
  - e.g. `user_has_bought_category_before`
  - e.g. generate fresh user history by combining batch + stream features
- Stream transformations:
  - e.g. geohash features
- WIP: Batch transformations:
  - e.g. batch joins `last_n_item_categories`

```python
request_source=RequestSource(
    name="request_data",
    schema=[Field(name="current_time", dtype=UnixTimestamp)]
)
# Transforming (user, item) pair feature + request data ("current_time")
@on_demand_feature_view(
    inputs={request_source, user_fv, item_feature},
    schema=[
      Field(name="time_since_purchased", dtype=Int64),
      Field(name="previously_purchased_item_cat", dtype=Int64),
      Field(name="purchased_item_ids", dtype=Array(Int32)),
    ]
)
def purchase_on_demand_features(inputs: pd.DataFrame):
   from keras.utils.np_utils import to_categorical
   import numpy as np
   df = pd.DataFrame()
   df["time_since_purchase"] = inputs["current_time"] - inputs["last_purchase_time"]
   df["previously_purchased_item_cat"] = df[["item_category", "prev_purchased_categories"]].apply(
       lambda x: x["item_category"] in x["prev_purchased_categories"],
       axis=1)
   df["purchased_item_ids"] = inputs.apply(
       lambda x: sorted(list(
           pd.unique(
               np.concatenate([x["last_1d_purchased_item_ids"], x["purchased_item_ids"]])))),
       axis=1)
   return df
```

Operational challenges with moving online

Operational challenges

Considerations when moving online

Among other requirements, an online recommender system often needs:

- fresh features (write heavy)
  - **Why?** e.g. user session activity for all users, precomputed features have delays
    - Different events update different features
- low latency access to features for many entities (read heavy)
  - **Why?** e.g. for a given user, need to rank 100s to 1000s of items
    - Typically, the faster the recommendation, the more likely users accept them.
    - The less time spent on data, the more time the model can spend inferring.
- low cost
  - **Why?** e.g. reads, writes, storage can be expensive, reducing value of moving online

Optimizing for the above can introduce significant data quality issues too.

Building a low latency online store

Consideration 1 (of 4)

Consideration

1. Balancing read vs write requirements
    a. update features independently (e.g. from streams)
    b. reading features for a specific model quickly

Example strategies

⇛ Collocate features from a stream / event together in both online store & offline store

⇛ Collocate features needed for a specific model

User Features
user_id
country
age
last_viewed_item_category
ts_country
ts_age
ts_last_5_viewed_item_category

User Metadata Features
user_id
country
age
timestamp

User Session Features
user_id
last_viewed_item_category
last_transaction_amt
timestamp

User Historical Features
user_id
28d_avg_transaction_amt
28d_top_item_category
timestamp

Embedding features
user_id
user_embedding
timestamp

Building a low latency online store

Consideration 2 (of 4)

Consideration

2. Managing type conversions for online store
    a. Data source types and Pandas / Python types (in data scientist notebook)
    b. Conversions are expensive

Pandas dtype	Python type	NumPy type
object	str or mixed	string_, unicode_, mixed types
int64	int	int_, int8, int16, int32, int64, uint8, uint16, uint32, uint64
float64	float	float_, float16, float32, float64
bool	bool	bool_
datetime64	NA	datetime64[ns]
timedelta[ns]	NA	NA
category	NA	NA

GSHEET... TABLE123

table123 SHARE COPY DELETE

SCHEMA DETAILS PREVIEW

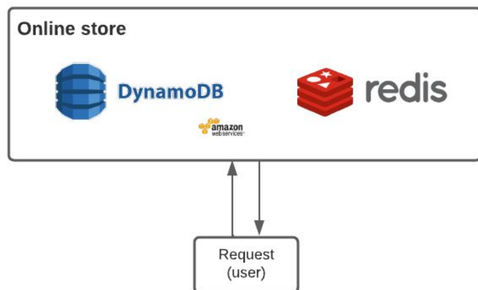
Row	string_field_0	string_field_1
1	INT64	12345
2	NUMERIC	520000000000
3	BIGNUMERIC	5.2e+37
4	FLOAT65	5.4321
5	BOOLEAN	false
6	STRING	555
7	BYTES	coupler_io
8	DATE	2021-05-01
9	DATE	2021-05-01-3.00
10	TIME	5:59:12.0422
11	DATETIME	2021-05-01 21:32:45
12	TIMESTAMP	2021-05-27 8:05:01-3:00
13	GEOGRAPHY	51.500989020415034, -0.12471081312336843
14	ARRAY	name, 123, 2021-01-01
15	STRUCT	555,'name'

Building a low latency online store

`Consideration 3 (of 4)`

Consideration

3. Optimizing for batch retrieval
   a. Large batch sizes (i.e. number of entities to score in the sample request)
   b. Online store specific optimizations.



Example strategies

⇛ Co-locating entities
⇛ Caching
⇛ E.g. Redis pipelines & `mget` vs `hmget` vs `hgetall`

Example: fetch features for all stores in a region

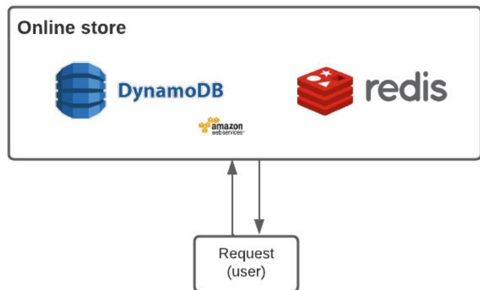
Store features				
geohash	store_id	feature_1	...	feature_N
...	...	...	...	...
...	...	...	..	...

Building a low latency online store

Consideration 4 (of 4)

Consideration

4. Cost
   a. Write cost
   b. Read cost
   c. Storage cost



Example strategies

⇛ incremental data processing
⇛ in-memory or out-of-process caching
⇛ online store TTL (warning: multiple models)

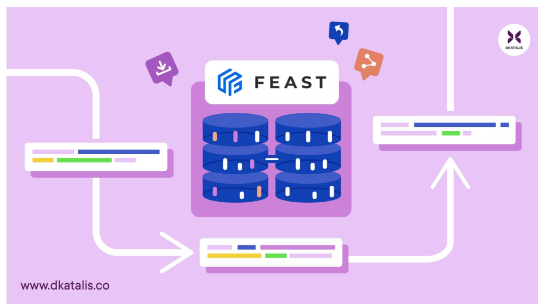
Correctness

Feature iteration

How to iterate on features safely

Challenges

1. How to avoid breaking model versions in production
2. Reproducible model training



[Blog: how DKatalis tackles this]

Example strategies

- ⇛ Feature + model lineage / versioning
- ⇛ Dev vs staging vs prod folders or branches
- ⇛ CI/CD checks + lints to enforce immutability
- ⇛ Feast `SavedDatasets` or using DVC to manage retrieved training data

Handling bad data

Data quality, data cleaning, drift

Example sources of bad data

- upstream systems change
- faulty feature transformation logic or messy data that has not been properly cleaned
- streams can publish bad data (or fail to publish data)

Mitigations

- ⇛ Implement data quality monitoring
  - e.g. see Feast DQM and versioned datasets via `SavedDatasets`
  - e.g. Great Expectations integration
  - can easily go wrong with false alerts
- ⇛ Visualize feature statistics
- ⇛ Fallback to old / default values or impute values for missing / faulty data.

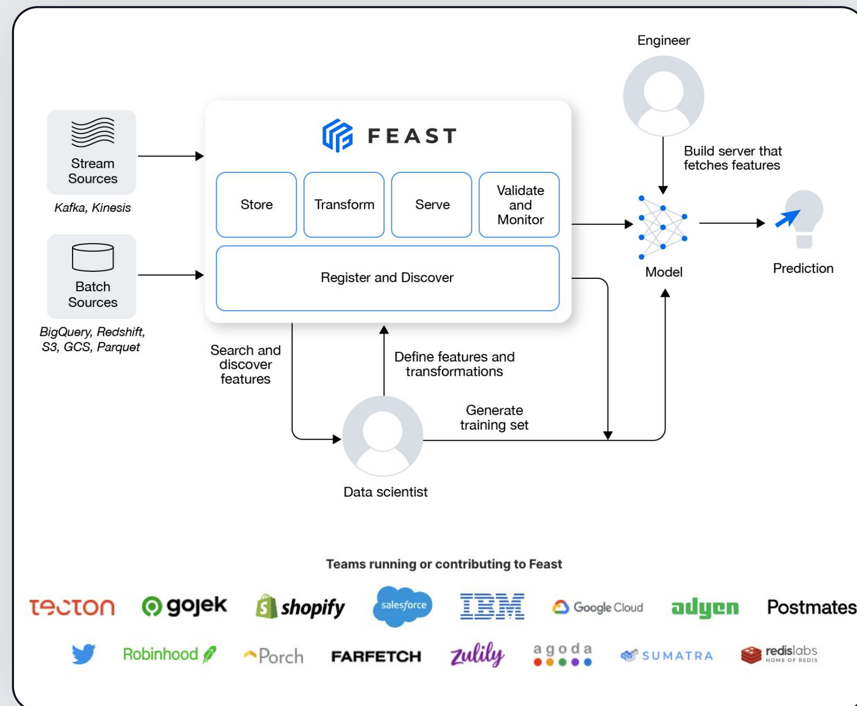
```python
DELTA = 0.1  # controlling allowed window in fraction of the value on

@ge_profiler
def stats_profiler(ds: PandasDataset) -> ExpectationSuite:
    # simple checks on data consistency
    ds.expect_column_values_to_be_between(
        "avg_speed",
        min_value=0,
        max_value=60,
        mostly=0.99  # allow some outliers
    )

    ds.expect_column_values_to_be_between(
        "total_miles_travelled",
        min_value=0,
        max_value=500,
        mostly=0.99  # allow some outliers
    )
```

Source: [Feast data quality monitoring tutorial]

Feast x RecSys

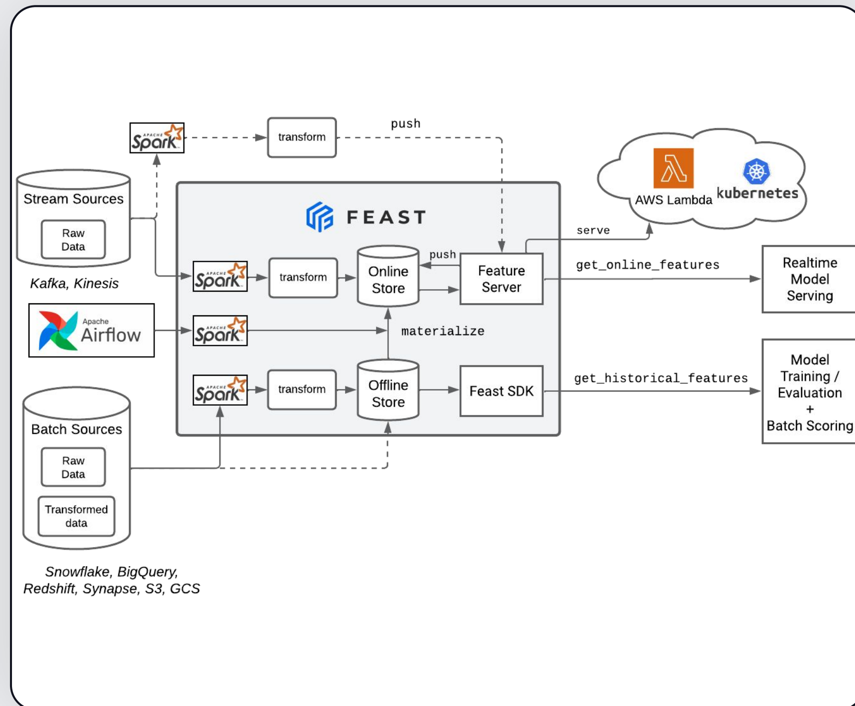
Feast

- Feast is an open-source pluggable feature store that connects to
  - **Batch sources** (via Spark, BigQuery, Redshift, Snowflake, Azure Synapse Analytics, Hive)
  - **Stream sources** (via push API or Spark)
- Active community with 3k+ Slack and bi-weekly community calls
- **Goal**: to simplify & reduce overhead of generating and managing ML features



Deploying Feast

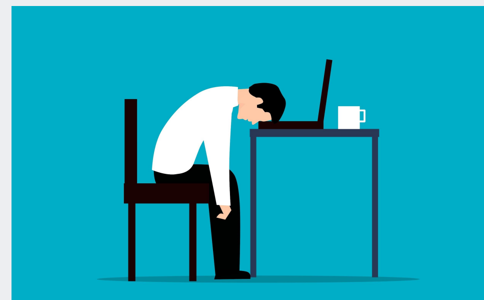
- Airflow for scheduled materialization of online features from batch sources
  - Stream processors leverage DS defined transforms or push to online store directly
- Embed SDK or deploy feature server
  - Serverless (e.g. Using Feast's AWS Lambda integration)
  - Kubernetes (e.g. Feature Server docs)
- Versioning models with feature service
- Pushing features in via push API
- Everything is pluggable



DATA+AI SUMMIT 2022

Takeaways

1. Incrementally move batch RecSys online (e.g make fresher features). Prove business value first.
2. Managing fresh features in an online store is not trivial
    - E.g. low latency reads vs write throughput, batch reads, iterating safely, bad data, cost
3. Feast abstracts complexity away, and is pluggable so you can incrementally solve more issues
4. Consistent + performant streaming & on demand transformations are key to online RecSys



Questions?

This talk

- https://bit.ly/feast-recsys-talk

Useful resources

- https://feast.dev/
- https://github.com/feast-dev/feast
- https://slack.feast.dev/