Machine Learning Data Lifecycle in Production

In the second course of Machine Learning Engineering for Production Specialization, you will build data pipelines by gathering, cleaning, and validating datasets and assessing data quality; implement feature engineering, transformation, and selection with TensorFlow Extended and get the most predictive power out of your data; and establish the data lifecycle by leveraging data lineage and provenance metadata tools and follow data evolution with enterprise data schemas.

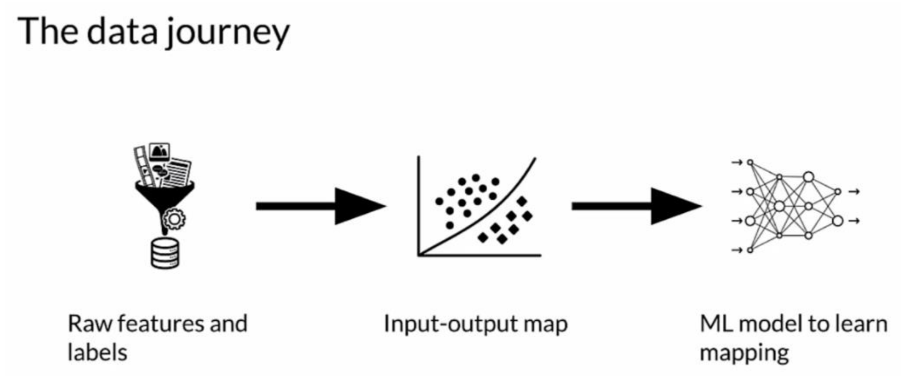
Understanding machine learning and deep learning concepts is essential, but if you're looking to build an effective AI career, you need production engineering capabilities as well. Machine learning engineering for production combines the foundational concepts of machine learning with the functional expertise of modern software development and engineering roles to help you develop production-ready skills.

Week 3: Data Journey and Data Storage

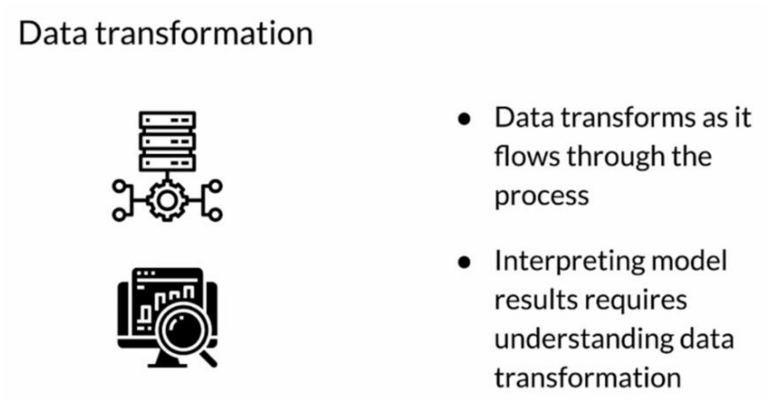
Contents

Compiled by **Kenneth Leung** Version: **202109**

Data Journey



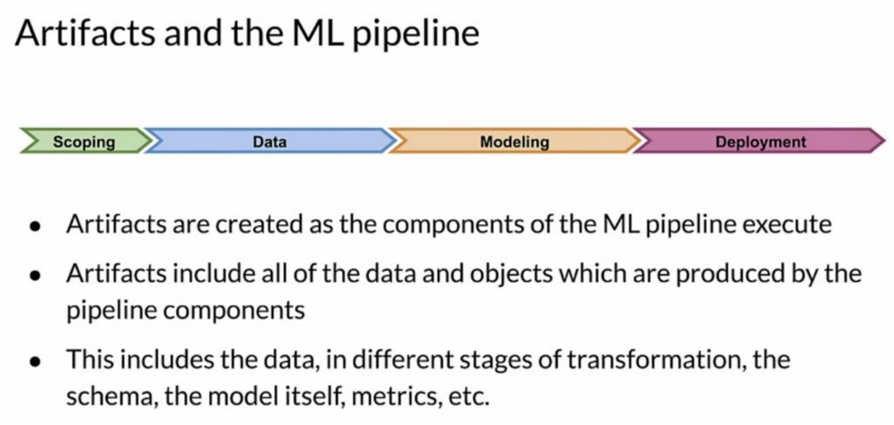
- Understanding data provenance requires understanding the Data journey throughout the life cycle of the production pipeline.
- More specifically, accounting for the evolution of data and models throughout the process.
- ML metadata is a versatile library to address these challenges, which helps with debugging and reproducibility.
- Concretely, it allows us to revisit and track data and model changes as they occur during training.
- Trying to understand data provenance begins with the Data journey. The journey starts with raw features and labels from whatever sources that we have.
- During training, the model learns the functional mapping from the input to the labels so as to be as accurate as possible.
- The Data transforms and changes as part of this training process.



Data transformation

- Data transforms as it flows through the process
- Interpreting model results requires understanding data transformation

- As the Data flows through the process, it transforms. Examples of this are, are changing data formats, applying feature engineering and training the model to make predictions.
- There's a dual connection between understanding these data transformations, and interpreting the model's results.
- Therefore, it's important to track and document these changes closely.

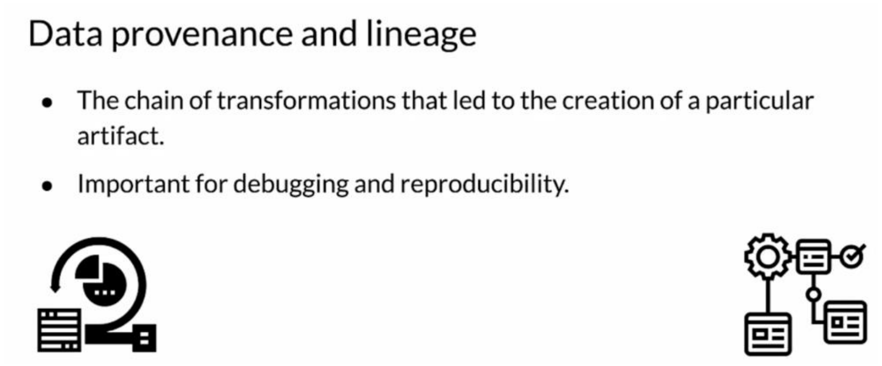


Artifacts and the ML pipeline

Scoping
Data
Modeling
Deployment

- Artifacts are created as the components of the ML pipeline execute
- Artifacts include all of the data and objects which are produced by the pipeline components
- This includes the data, in different stages of transformation, the schema, the model itself, metrics, etc.

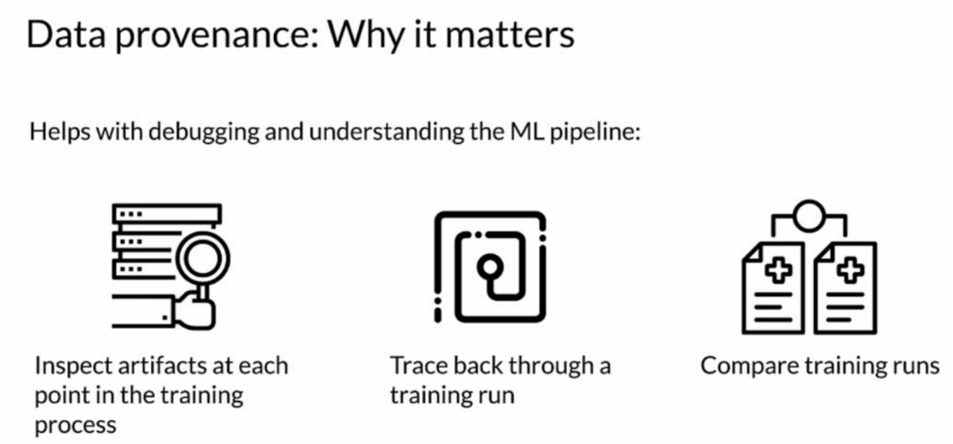
- Data artifacts are created as pipeline components execute.
- What exactly is an artifact? Each time a component produces a result, it generates an **artifact**.
- This includes basically everything that is produced by the pipeline, including the data in different stages of transformation, often as a result of feature engineering and the model itself, and things like the schema, and metrics and so forth.
- Basically, everything that's produced, every result that is produced, is an artifact.



Data provenance and lineage

- The chain of transformations that led to the creation of a particular artifact.
- Important for debugging and reproducibility.

- The **terms data provenance and data lineage are basically synonyms** and they're used interchangeably.
- Data provenance or lineage is a **sequence of artifacts** that are created as we move through the pipeline.
- Those artifacts are associated with code and components that we create.
- Tracking those sequences is key for debugging and understanding the training process and comparing different training runs that may happen months apart.



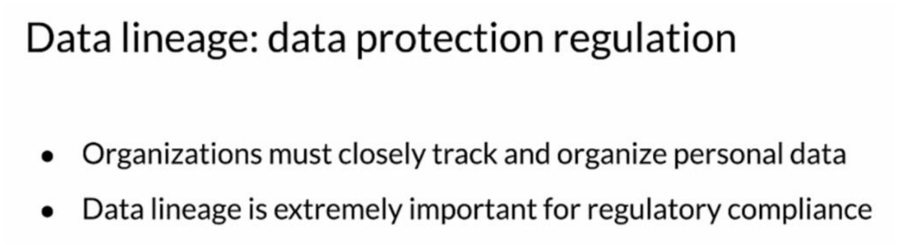
Data provenance: Why it matters

Helps with debugging and understanding the ML pipeline:

- Inspect artifacts at each point in the training process
- Trace back through a training run
- Compare training runs

- Machine learning pipelines for production have become prominent in several industries. They introduce complexity to the ML lifecycle due to the large amount of data, tools, and workflows involved. If data and models are not tracked properly during the life cycle, it becomes infeasible to recreate an ML model from scratch or to explain to stakeholders how it was created.

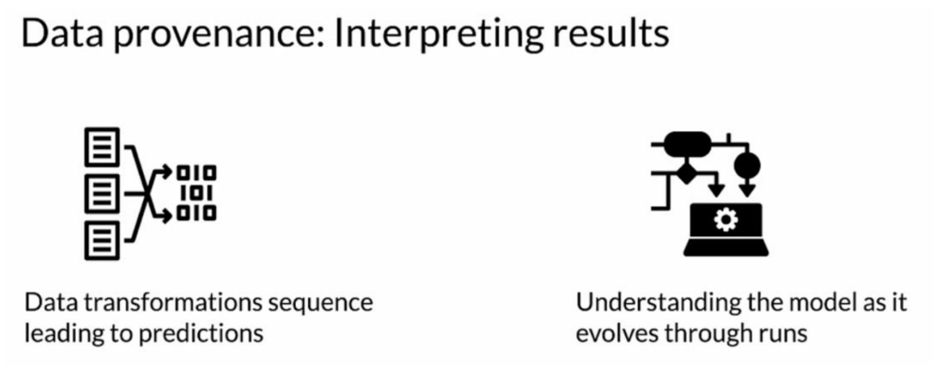
- Establishing data and model provenance tracking mechanisms help to prevent these shortcomings.
- Data provenance matters a great deal and it helps us understand the pipeline and perform debugging.
- Debugging and understanding requires inspecting those artifacts at each point in the training process, which can help us understand how those artifacts were created and what the results actually mean.
- Provenance will also allow you to track back through a training run from any point in the process.
- Also, provenance makes it possible to compare training runs, and understand why they produce different results.



Data lineage: data protection regulation

- Organizations must closely track and organize personal data
- Data lineage is extremely important for regulatory compliance

- Under GDPR or the general data protection regulation, organizations are accountable for the origin, changes and location of personal data.
- Personal data is highly sensitive, so tracking the origins and changes along the pipeline are key for compliance. Data lineage is a great way for businesses and organizations to quickly determine how the Data has been used and which Transformations were performed as the Data moved through the pipeline.

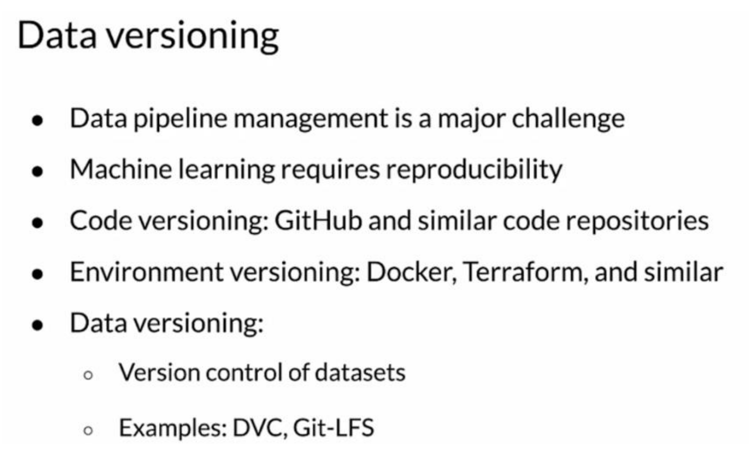


Data provenance: Interpreting results

Data transformations sequence leading to predictions

Understanding the model as it evolves through runs

- Data provenance is key to interpreting model results. Model understanding is related to this, but it's only part of the picture.
- The model itself is an expression of the data in the training set. In some sense, we can look at the model as a transformation of the Data.
- Provenance also helps us understand how the model evolves as it is trained and perhaps optimized.



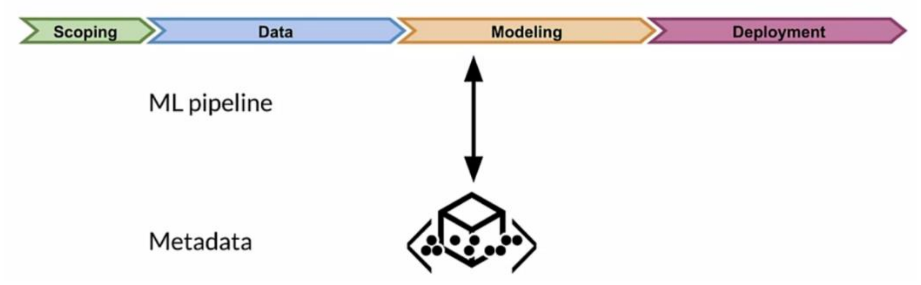
Data versioning

- Data pipeline management is a major challenge
- Machine learning requires reproducibility
- Code versioning: GitHub and similar code repositories
- Environment versioning: Docker, Terraform, and similar
- Data versioning:
  - Version control of datasets
  - Examples: DVC, Git-LFS

- Let's add an important ingredient here, tracking different Data versions.
- Managing a data pipelines is a big challenge as data evolves through the natural life cycle of a project, over many different training runs.
- Machine learning, when it's done properly, should produce results that can be reproduced fairly consistently. There will naturally be some variance, but the results should be close.
- ML pipelines should thus incorporate resilient mechanisms to deal with inconsistent data, and account for anomalies
- Code version control is probably something you're familiar with.
- GitHub is one of the most popular cloud-based code repositories, and there are others as well.
- Environment versioning is also important. Tools like Docker and terraform help us create repeatable environments and configuration.
- However, data versioning is also important, and plays a significant role for tracking provenance and lineage data versioning.
- It's essentially version control for data files so that you can trace changes over time and restore previous versions early.
- But the tools are somewhat different. For one thing, because of the size of files that we deal with, which are typically or can be any way, much larger than a code file would ever be.
- Tools for data versioning are starting to become available. These include DVC and Git LFS (large files storage)

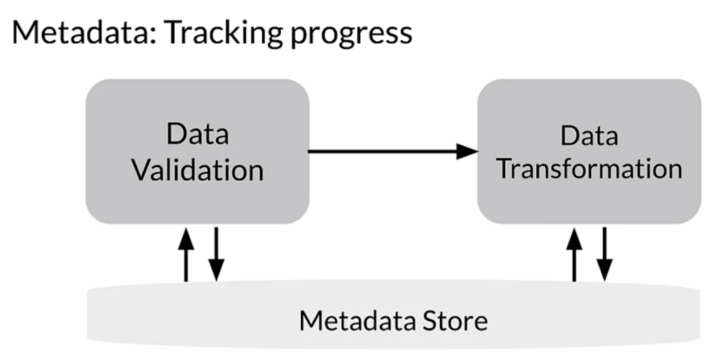
Introduction to ML Metadata

Metadata: Tracking artifacts and pipeline changes

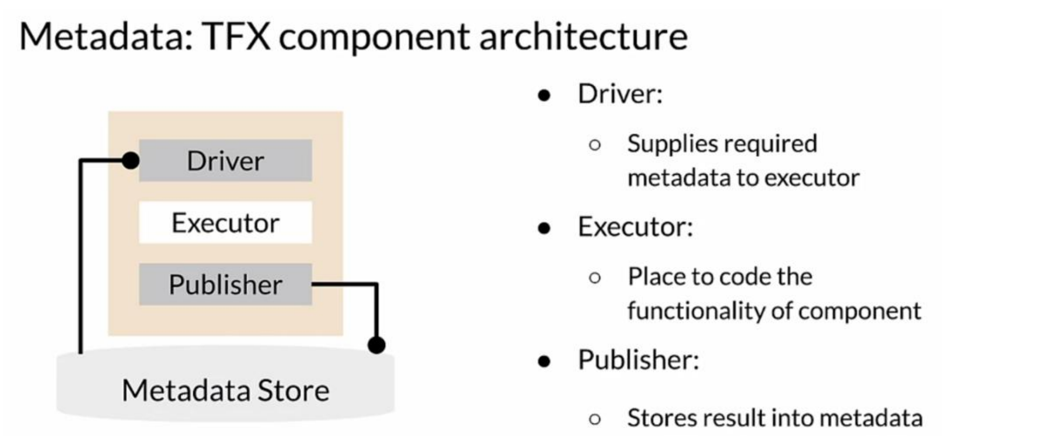


- As machine learning becomes increasingly used to make important business decisions, healthcare decisions and financial decisions, legal liability becomes a factor. Being able to interpret a model and being able to trace back the lineage or provenance of the data that was used to train the model becomes increasingly important to limiting exposure.

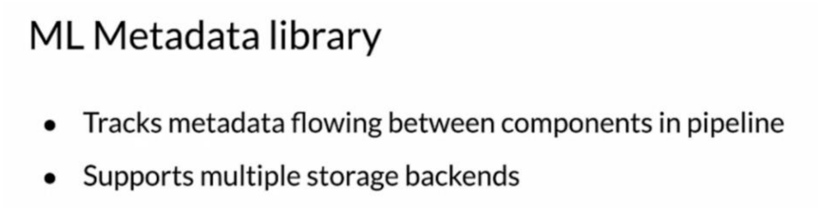
- Now let's start exploring how ML metadata (or **MLMD)** can help you with tracking artifacts and pipeline changes during a production life cycle.
- ML Metadata (MLMD) is a library for recording and retrieving metadata associated with ML production pipelines among other applications.
- Every run of a production ML pipeline generates metadata containing information about the various pipeline components and their executions or training runs and the resulting artifacts.
- In the event of unexpected pipeline behavior or errors, this metadata can be leveraged to analyze the lineage of pipeline components and to help you debug issues.
- Think of this metadata as the **equivalent of logging in software development**.
- MLMD helps you understand and analyze all the interconnected parts of your ML pipeline, instead of analyzing them in isolation.



- Now consider the two stages in ML engineering that you've seen so far. First you've done data validation, then you've passed the results onto data transformation or feature engineering. This is the first part of any model training process.
- But what if you had a centralized repository where every time you run a component, you store the result or the update or any other output of that stage into a repository.
- So whenever any changes made, which causes a different result, you don't need to worry about the progress you've made so far getting lost.
- You can examine your previous results to try to understand what happened and make corrections or take advantage of improvements.


Metadata: TFX component architecture
Driver
Executor
Publisher
Metadata Store
Driver:
Supplies required metadata to executor
Executor:
Place to code the functionality of component
Publisher:
Stores result into metadata

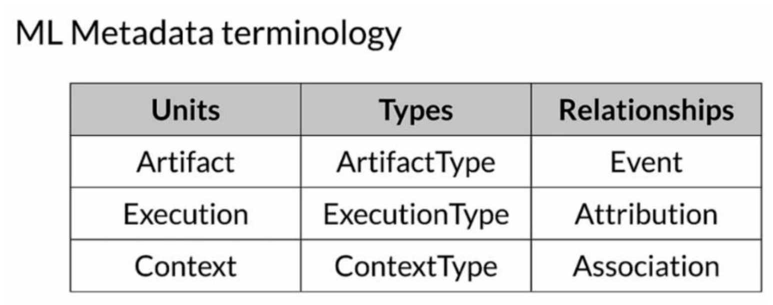
- Let's take a closer look. In addition to the executor where your code runs, each component also includes two additional parts, the driver and publisher.
- The executor is where the work of the component is done and that's what makes different components different. Whatever input is needed for the executor, is provided by the driver, which gets it from the metadata store.
- Finally, the publisher will push the results of running the executor back into the metadata store.
- Most of the time, you won't need to customize the driver or publisher.
- Creating custom components is almost always done by creating a custom executor.



ML Metadata library

- Tracks metadata flowing between components in pipeline
- Supports multiple storage backends

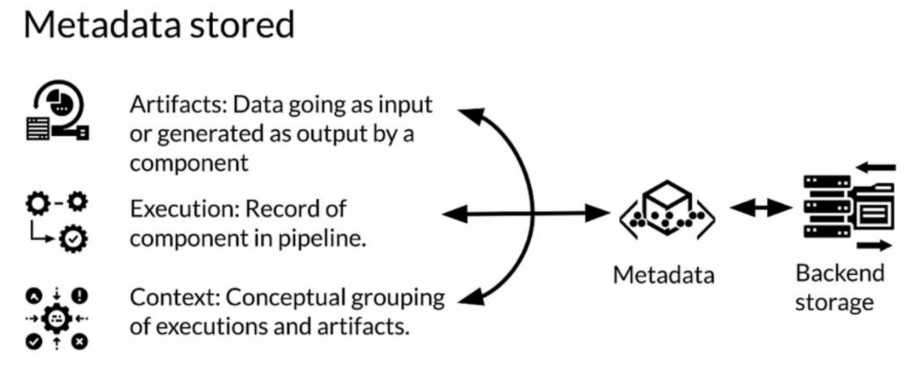
- Now, let's look specifically at ML Metadata or MLMD.
- **MLMD is a library for tracking and retrieving metadata** associated with ML developer and data scientist workflows.
- MLMD can be used as an integral part of an ML pipeline or it can be used independently.
- However, when integrated with an ML pipeline, you may not even explicitly interact with MLMD.
- Objects which are stored in MLMD are referred to as **artifacts**.
- MLMD stores the properties of each artifact in a relational database and stores large objects like data sets on disc or in a file system or block store.



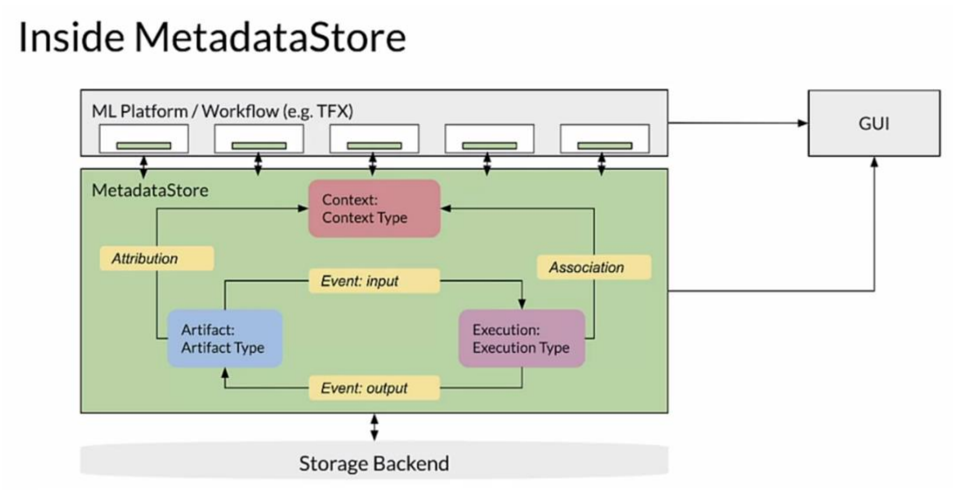
ML Metadata terminology

Units	Types	Relationships
Artifact	ArtifactType	Event
Execution	ExecutionType	Attribution
Context	ContextType	Association

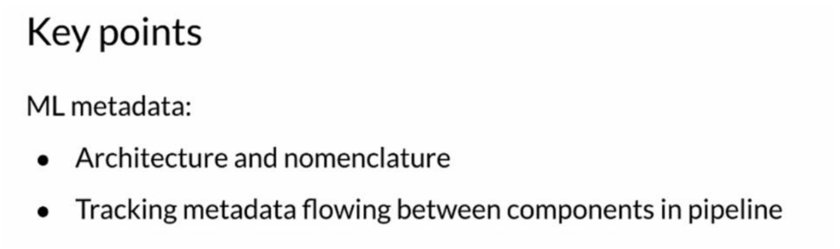
- When you're working with ML metadata, you need to know how data flows between different successive components.
- Each step in this data flow is described through an **entity** that you need to be familiar with. At the highest level of MLMD, there are some data entities that can be considered as **units**.
- First, there are artifacts. An artifact is an elementary unit of data that gets fed into the ML metadata store and as the data is consumed as input or generated as output of each component.
- Next there are **executions**. Each execution is a record of any component run during the ML pipeline workflow, along with its associated runtime parameters.
- Any artifact or execution will be associated with only one type of component.
- Artifacts and executions can be clustered together for each type of component separately. This grouping is referred to as the context.
- A context may hold the metadata of the projects being run, experiments being conducted, details about pipelines, etc.
- Each of these units can hold additional data describing it in more detail using properties.
- Next there are types. Previously, you've seen several types of units that get stored inside the ML metadata. Each type includes the properties of that type.
- Lastly, we have relationships. Relationships store the various units getting generated or consumed when interacting with other units.
- For example, **an event is the record of a relationship between an artifact and an execution**.
- An association is a record of the relationship between executions and context
- An attribution is a record of the relationship between artifacts and context
- Check the ML metadata documentation for more information


Metadata stored
Artifacts: Data going as input or generated as output by a component
Execution: Record of component in pipeline.
Context: Conceptual grouping of executions and artifacts.
Metadata
Backend storage

- So, ML metadata stores a wide range of information about the results of the components and execution runs of a pipeline.
- It stores artifacts and it stores the executions of each component in the pipeline.
- It also stores the lineage information for each artifact that is generated.
- All this information is represented in metadata objects and are stored in a backend storage solution.



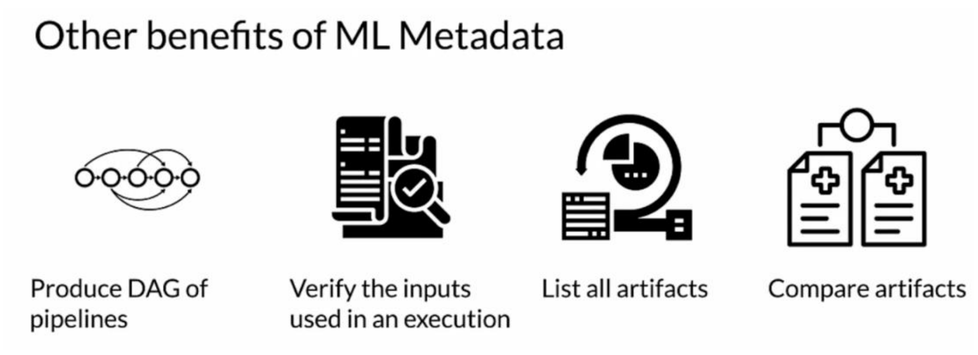
- Let's take a look at the architecture of ML metadata or MLMD.
- On the top are the various components present in any ML pipeline.
- All of these components are individually connected to a centralized metadata store of ML metadata, so that each component can independently access the metadata at any stage of the pipeline.
- An ML pipeline may optionally have a GUI console that can access the data from the metadata store directly to track the progress of each component.
- At the heart of the metadata store is the artifact which is described by its corresponding artifact type.
- Artifacts become the inputs to any pipeline components which depend on them. And the corresponding use of artifacts by components is recorded in executions.
- The input of an artifact into a component is described by an input event and the corresponding output of a new artifact from the component is described by an output event.
- This interaction between artifacts and executions is represented by context through the relationships of attribution and association.
- Lastly, all of the data generated by the metadata store is stored in various types of back end storage like SQLite and MySQL, and large objects are stored in a file system or block store.


Key points

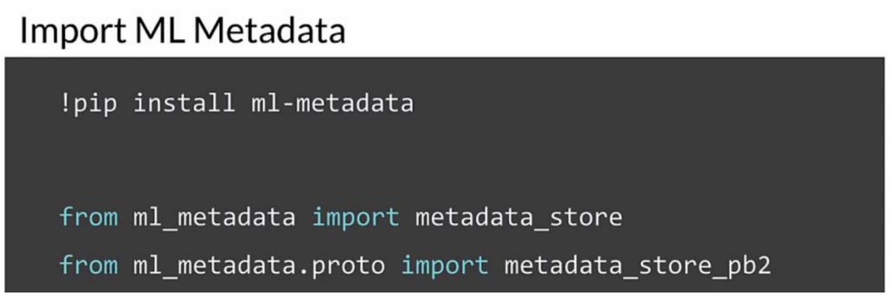
ML metadata:

- Architecture and nomenclature
- Tracking metadata flowing between components in pipeline

ML Metadata in Action

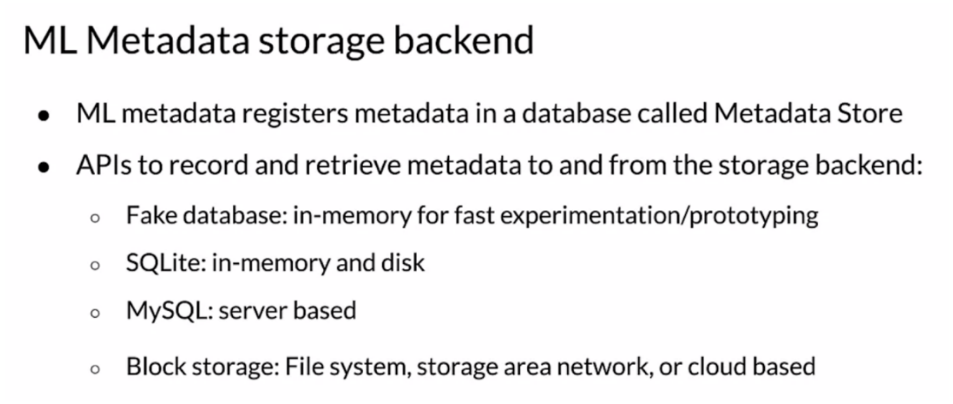

Other benefits of ML Metadata

- Now, let's take a look at using ML Metadata with a coding example.
- Besides data lineage and provenance tracking, you get several other benefits through ML Metadata.
- This includes the ability to construct a directed acyclic graph or DAG, of the component executions occurring in a pipeline, which can be useful for debugging purposes.
- Through this, you can verify which inputs have been used in an execution.
- You can also summarize all artifacts belonging to a specific type generated after a series of experiments.
- For example, you can list all of the models that have been trained. You can then compare them to evaluate your various training runs.


Import ML Metadata

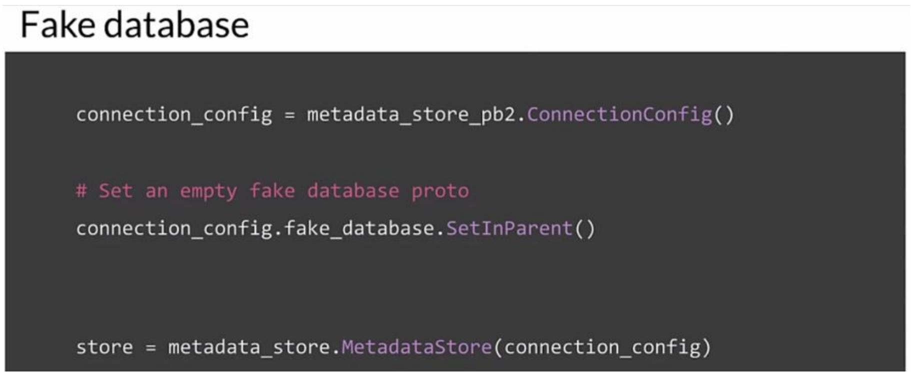
```
!pip install ml-metadata


from ml_metadata import metadata_store
from ml_metadata.proto import metadata_store_pb2
```

- Now, let's get started with writing code for ML Metadata. You may have to install ML Metadata, which you can do using PIP as shown here.
- Then there are two imports from ML Metadata which are used frequently, the ML Metadata store itself and the ML Metadata store PB2, which is a protocol buffer or protobuf.


ML Metadata storage backend

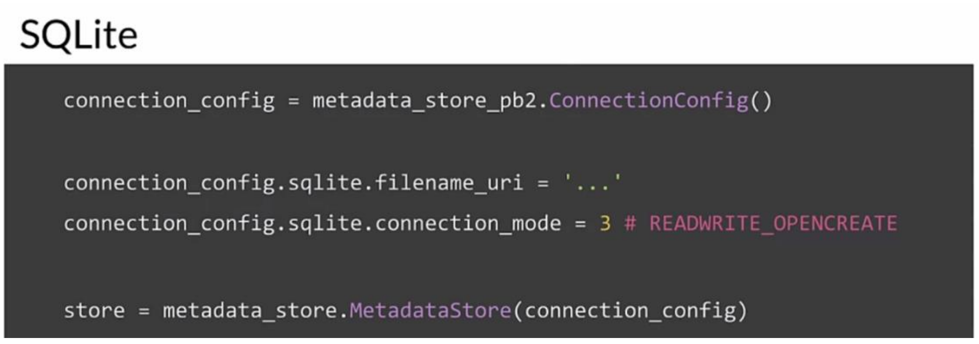
- ML metadata registers metadata in a database called Metadata Store
- APIs to record and retrieve metadata to and from the storage backend:
  - Fake database: in-memory for fast experimentation/prototyping
  - SQLite: in-memory and disk
  - MySQL: server based
  - Block storage: File system, storage area network, or cloud based

- Start by setting up the storage backend. ML Metadata store is the database where ML Metadata registers all of the metadata associated with your project.
- ML Metadata provides APIs to connect with a fake database for quick prototyping, SQLite and MySQL.
- We also need a block store or file system where ML Metadata stores large objects like datasets.


Fake database

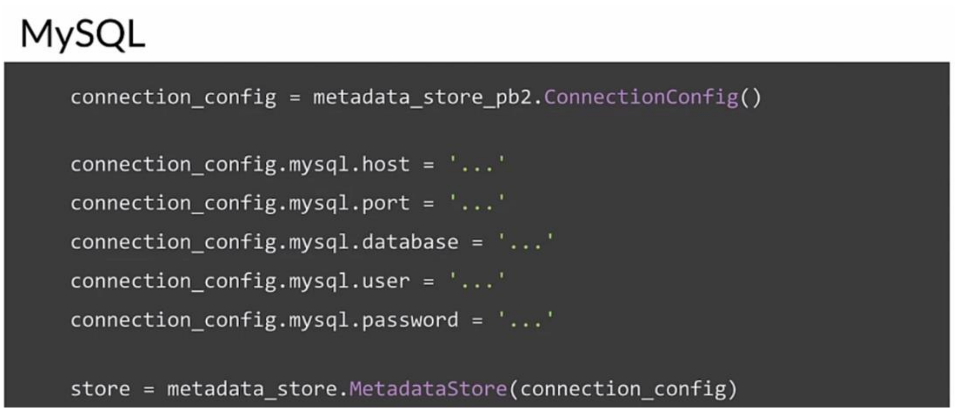
```
connection_config = metadata_store_pb2.ConnectionConfig()

# Set an empty fake database proto
connection_config.fake_database.SetInParent()


store = metadata_store.MetadataStore(connection_config)
```

- Let's quickly explore the first three options. For any storage backend, you'll need to create a connection_config object using the metadata store PB2 protobuf.
- Then, based on your choice of storage backend, you need to configure this connection object.
- Here you're signaling that your fake database is in parent, which is the primary memory of the system on which it's running.
- Finally, you create the store object passing in the connection config. Regardless of what storage or database you use, the store object is a key part of how you interact with ML Metadata.
- **The MetaDataStore object receives a connection configuration that corresponds to the storage backend used.**


SQLite

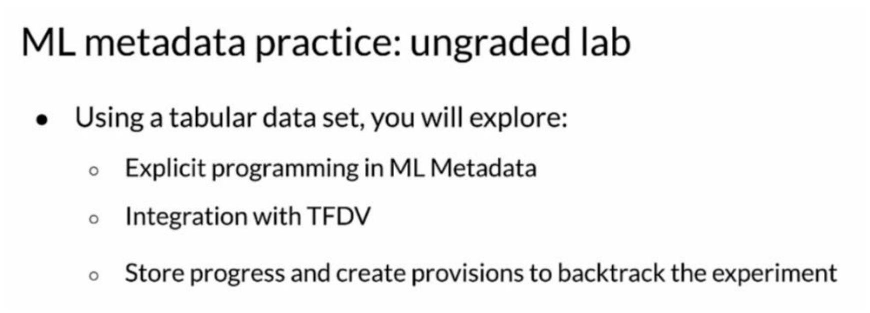
```
connection_config = metadata_store_pb2.ConnectionConfig()

connection_config.sqlite.filename_uri = '...'
connection_config.sqlite.connection_mode = 3 # READWRITE_OPENCREATE

store = metadata_store.MetadataStore(connection_config)
```

- To use SQLite, you start by creating a connection config again then you configure the connection config object with the location of your SQLite file.
- Make sure to provide that with the relevant read and write permissions based on your application.
- Finally, you create the store object passing in the connection config.


MySQL

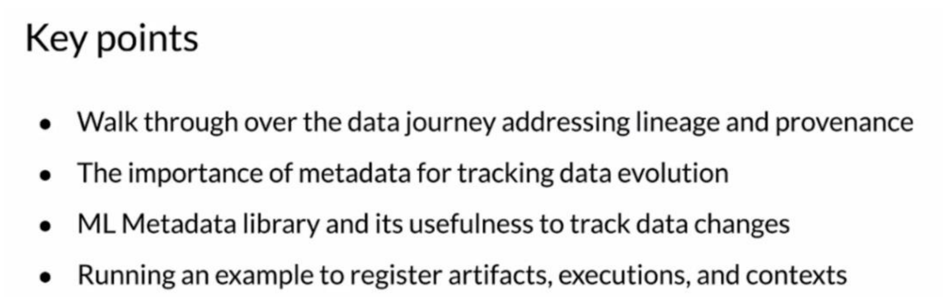
```
connection_config = metadata_store_pb2.ConnectionConfig()

connection_config.mysql.host = '...'
connection_config.mysql.port = '...'
connection_config.mysql.database = '...'
connection_config.mysql.user = '...'
connection_config.mysql.password = '...'

store = metadata_store.MetadataStore(connection_config)
```

- As you might imagine, using MySQL is very similar, you start by creating a connection config.
- Your connection config object should be configured with the relevant host name, the port number, the database location, username, and password of your MySQL database user, and that's shown here.
- Finally, you create the store object, passing in the connection config.


ML metadata practice: ungraded lab

- Using a tabular data set, you will explore:
  - Explicit programming in ML Metadata
  - Integration with TFDV
  - Store progress and create provisions to backtrack the experiment

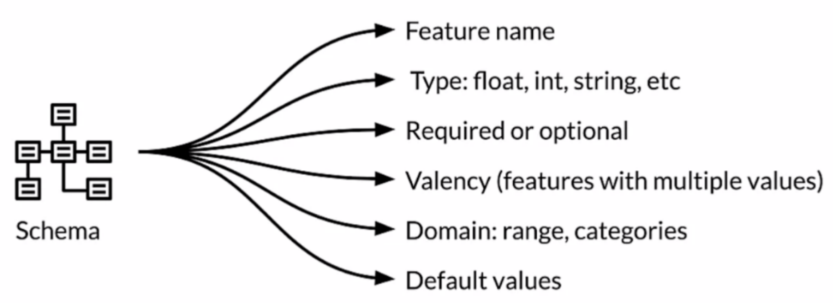
- Now, that the storage backend is configured, it's time to use ML Metadata to solve a problem. Let's go over a workflow using **TFDV**. We're going to use that in conjunction with ML Metadata.
- The choice here is a tabular dataset containing many features. In the lab, ML Metadata is explicitly programmed because in a full-fledged ML Pipeline, ML Metadata is intrinsically capable enough to understand the flow of data between various components and perform its necessary duties.
- To help you better understand ML Metadata, you'll be using it outside of a pipeline.
- The lab also shows ML Metadata integration with TensorFlow Data Validation or TFDV.
- By the end of the lab, you should have some intuition about how ML Metadata can keep track of progress and how you can use ML Metadata to track your training process and pipeline.


Key points

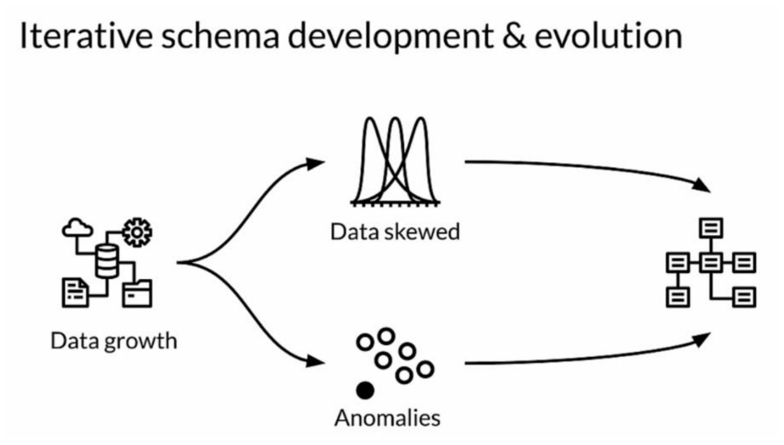
- Walk through over the data journey addressing lineage and provenance
- The importance of metadata for tracking data evolution
- ML Metadata library and its usefulness to track data changes
- Running an example to register artifacts, executions, and contexts

Schema Development

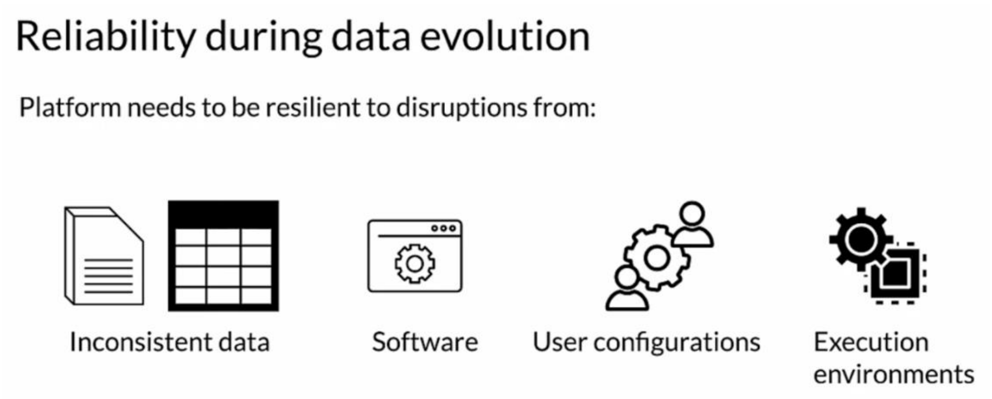
Review: Recall Schema


Schema
Feature name
Type: float, int, string, etc
Required or optional
Valency (features with multiple values)
Domain: range, categories
Default values

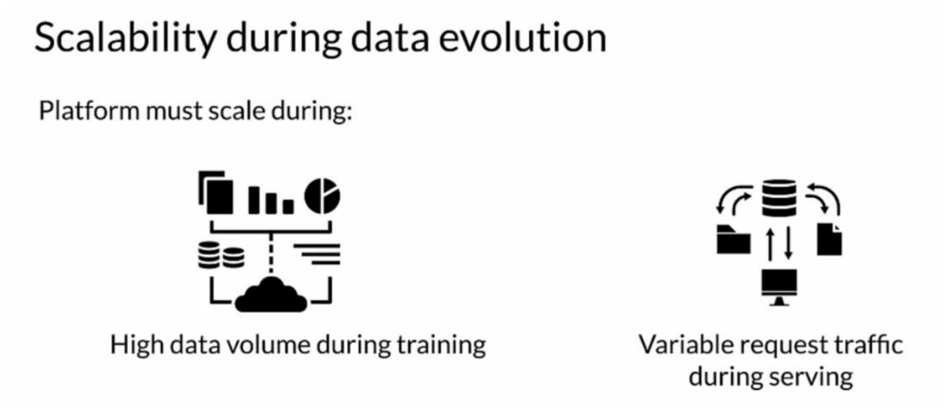
- Now let's discuss schema development. In this lesson on evolving data, you'll learn about developing enterprise schema environments and how to iteratively create and maintain enterprise data schemas.
- You'll be using schemas quite extensively. Let's quickly review what a schema is and how helpful they are in the context of production ML.
- **Schemas are relational objects summarizing the features in a given dataset or project.** They include the feature name, the feature of variable type.
- For example, an integer, float, string or categorical variable. Whether or not the feature is required. The valency of the feature, which applies to features with multiple values like lists or array features, and expresses the minimum and maximum number of values.
- Information about the range and categories and feature default values.



- Schemas are important, because your data and feature set evolves over time.
- From your experience, you know that data keeps changing and this change often results in change distributions. Let's focus on how to observe data that has changed as it keeps evolving.
- **Changing data often results in a new schema being generated.**
- However, there are some special use cases. Imagine that even before you assess the dataset, you have an idea or information about the expected range of values for your features.
- The initial dataset that you've received is covering only a small range of those values.
- In that case, it makes sense to adjust or create your schema to reflect the expected range of values for your features.
- A similar situation may exist for the expected values of categorical features.
- Besides that, your schema can help you find problems or anomalies in your dataset, such as missing required values or values of the wrong type.
- All these factors have to be considered when you're designing the schema of your ML pipeline.


Reliability during data evolution
Platform needs to be resilient to disruptions from:
Inconsistent data
Software
User configurations
Execution environments

- As data keeps evolving, there are some requirements which must be met in production deployments. Let's consider some important ones.
- The first factor is **reliability**. The ML platform of your choice should be **resilient to disruptions** from inconsistent data.
- There's no guarantee that you'll receive clean and consistent data every time. In fact, I can almost guarantee you that you won't.
- Your system needs to be designed to handle that efficiently.
- Also, your software might generate unexpected runtime errors and your pipeline needs to gracefully handle that also.
- Problems with misconfiguration should be detected and handled gracefully.
- Above all, the orchestration of execution among different components in the pipeline should happen smoothly and efficiently.

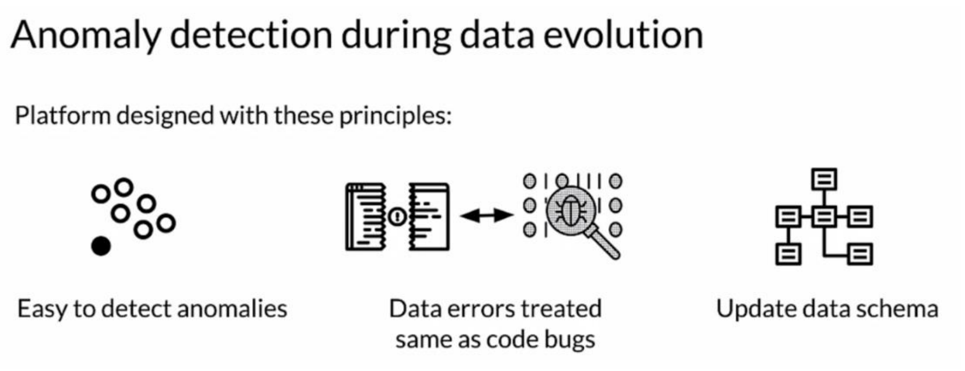

Scalability during data evolution

Platform must scale during:

High data volume during training

Variable request traffic during serving

- Another factor to consider in your ML pipeline platform is **scalability** during data evolution.
- During training, your pipeline may need to handle a large amount of training data well, including keeping expensive accelerators like GPUs and TPUs busy.
- When you serve your model, especially in online scenarios such as running on a server, you'll almost always have varying levels of request traffic.
- Your infrastructure needs to **scale up and down to meet those latency requirements** while minimizing cost.


Anomaly detection during data evolution

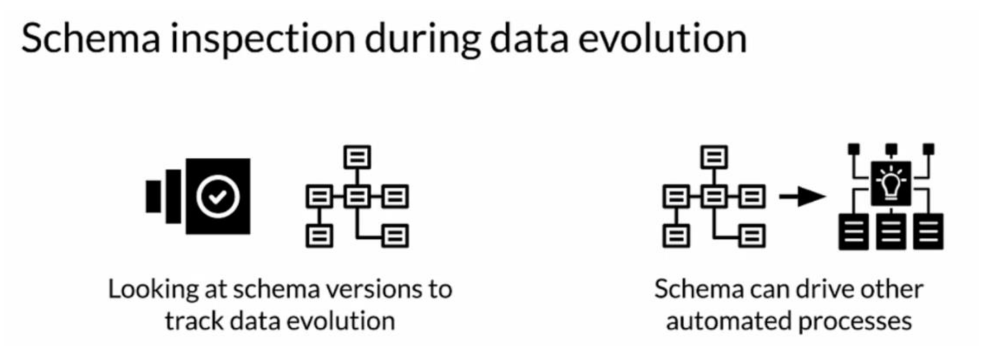
Platform designed with these principles:

Easy to detect anomalies

Data errors treated same as code bugs

Update data schema

- If your system isn't designed to handle data evolution, it will quickly run into problems.
- These include the introduction of anomalies in your dataset. Will your system detect those anomalies? Your system and your development process should be designed to **treat data errors as first-class citizens** in the same way that bugs in your code are treated.
- In some cases, those anomalies are alerting you that you need to update the schema and accommodate valid changes in your data.

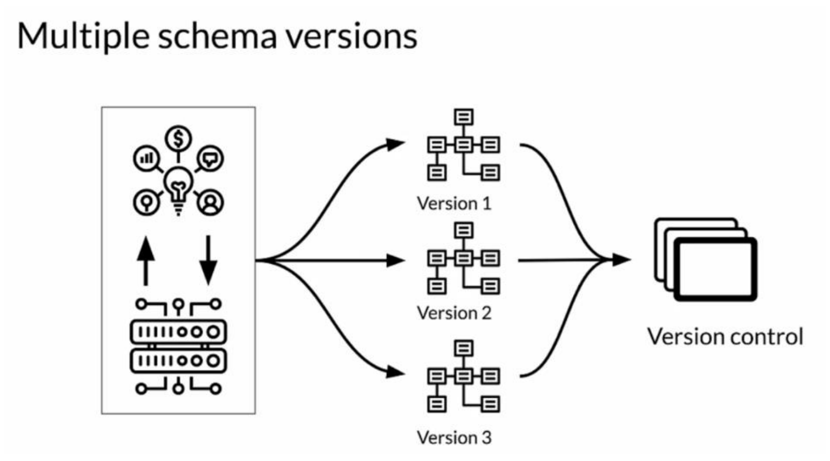

Schema inspection during data evolution

Looking at schema versions to track data evolution

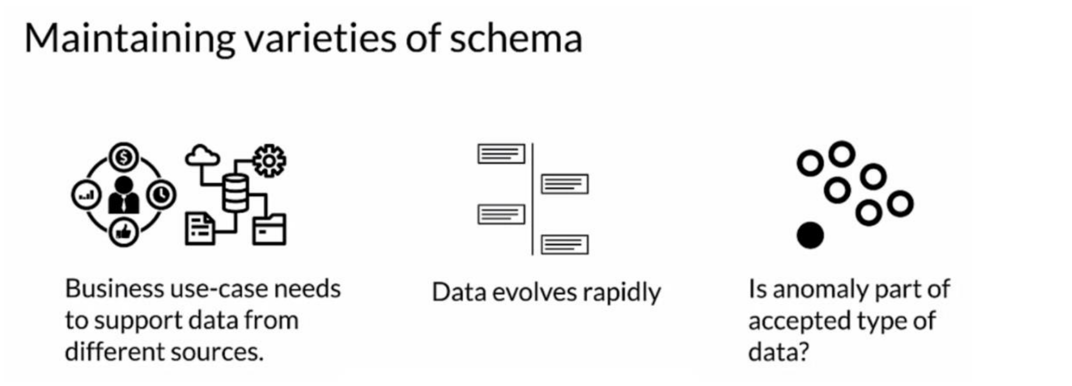
Schema can drive other automated processes

- The evolution of your schemas can be a useful tool to understand and track the evolution of your data.
- Also, schemas can be used as key inputs into automated processes that work with your data like automatic feature engineering.

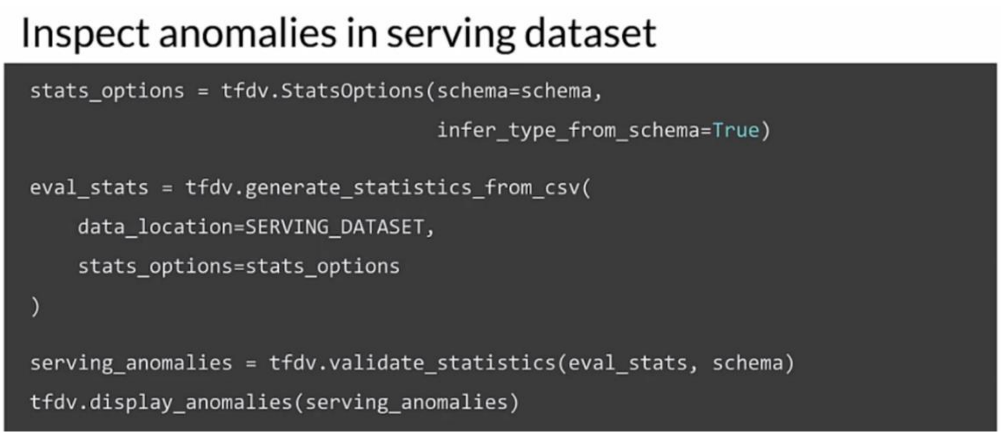
Schema Environments



- Let's discuss schema environments. Your business and data will evolve throughout the lifetime of your production pipeline. It's often the case that as your data evolves, your schema evolves also.
- As you're developing your code to handle changes in your schema, you may have multiple versions of your schema all active at the same time.
- You may have one schema being used for development, one currently in test, and another currently in production.
- Having version control for your schemas, just like you do for your code, helps make this manageable.


Maintaining varieties of schema
Business use-case needs to support data from different sources.
Data evolves rapidly
Is anomaly part of accepted type of data?

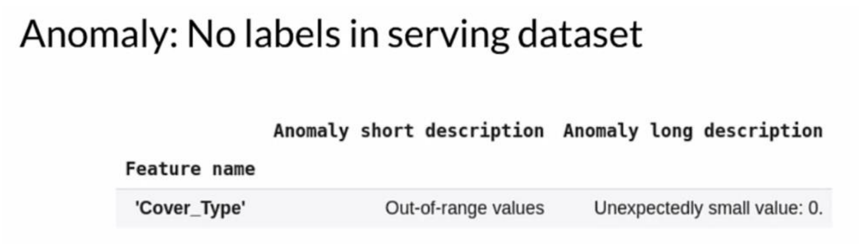
- In some cases, you may need to have different schemas to support multiple training and deployment scenarios for different data environments.
- For example, you may want to use the same model on a server and in a mobile application but imagine that a particular feature is different in those two environments.
- Maybe in one case it's an integer and the other case it's a float.
- You need to have a **different schema for each to reflect the difference in the data**.
- Along with that, your data's evolving. Potentially, in all your different data environments at once.
- At the same time you also needed to check your data for problems or anomalies, and **schemas are a key part of checking for anomalies.**



Inspect anomalies in serving dataset

```
stats_options = tfdv.StatsOptions(schema=schema,
                                  infer_type_from_schema=True)

eval_stats = tfdv.generate_statistics_from_csv(
    data_location=SERVING_DATASET,
    stats_options=stats_options
)

serving_anomalies = tfdv.validate_statistics(eval_stats, schema)
tfdv.display_anomalies(serving_anomalies)
```

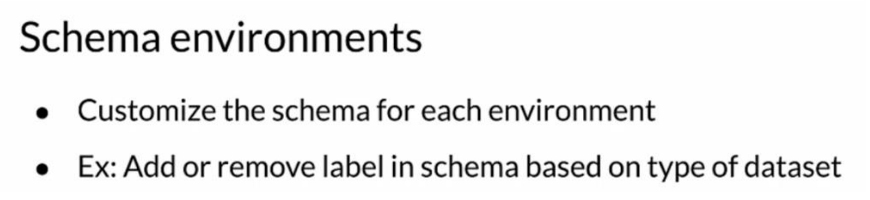
- Let's look at an example of how a schema can help you detect errors in your serving request data and why multiple versions of your schema are important.
- We'll start by inferring the serving schema and we'll use TensorFlow Data Validation or TFDV to do that.
- Then we're going to generate statistics for the serving dataset. Then we'll use TFDV to find if there are any problems with this data and visualize the result in a notebook.



Anomaly: No labels in serving dataset

	Anomaly short description	Anomaly long description
Feature name		
'Cover_Type'	Out-of-range values	Unexpectedly small value: 0.

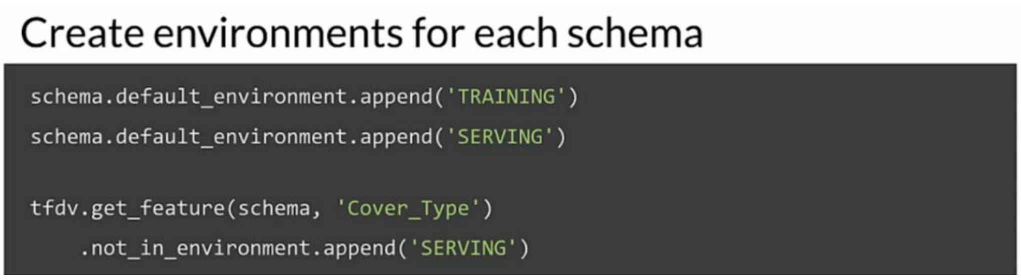
- Look at that. TFDV reports back that there are anomalies in the serving data.
- Since this is a dataset that contains prediction requests, that's actually not surprising.
- The **label** which is cover type is missing, but the schema is telling TFDV that the cover type feature is required. So it's flagging this as an anomaly. How do we fix this problem?



Schema environments

- Customize the schema for each environment
- Ex: Add or remove label in schema based on type of dataset

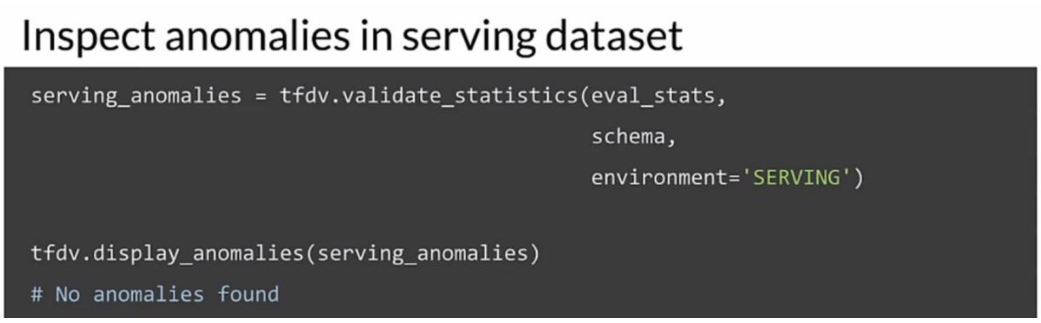
- In scenarios where you need to maintain multiple types of the same schemas, you often need to keep the **schema environment**.
- This is most commonly true of the difference between training and serving data.
- You can choose to customize your schema based on the situation you're going to handle.
- For example, in this case, the setup is to maintain two schemas. One for training data, where the label is required and the other for serving where we know we won't have the label.



Create environments for each schema

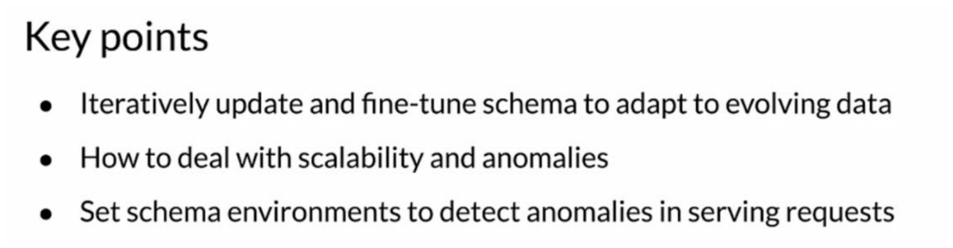
```
schema.default_environment.append('TRAINING')
schema.default_environment.append('SERVING')

tfdv.get_feature(schema, 'Cover_Type')
    .not_in_environment.append('SERVING')
```

- The code for multiple schema environments is fairly straightforward. In our existing environment, we already have a schema for training.
- We then create two named environments called training and serving.
- We modify our serving environment by removing the cover type feature. Since we know that in serving, we won't have that in our feature set.


Inspect anomalies in serving dataset

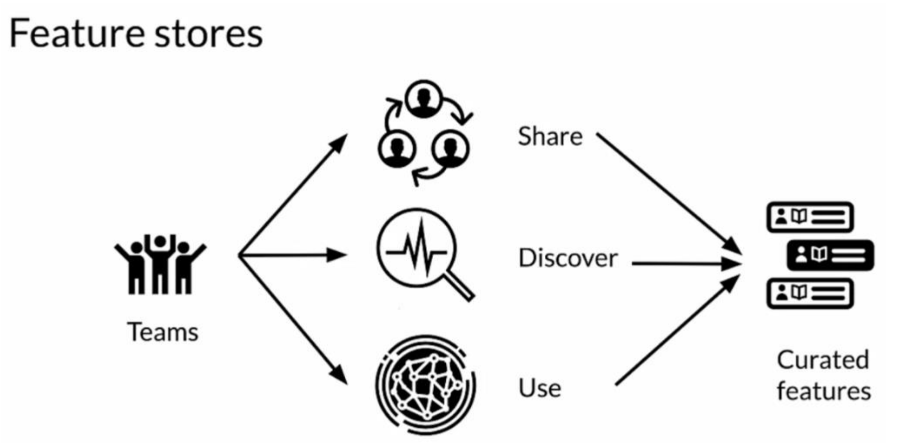
```
serving_anomalies = tfdv.validate_statistics(eval_stats,
                                             schema,
                                             environment='SERVING')

tfdv.display_anomalies(serving_anomalies)
# No anomalies found
```

- Lastly, the code sets up the serving environment and uses it to validate the serving data.
- Now, there are no anomalies found since we're using the correct schema for our data.

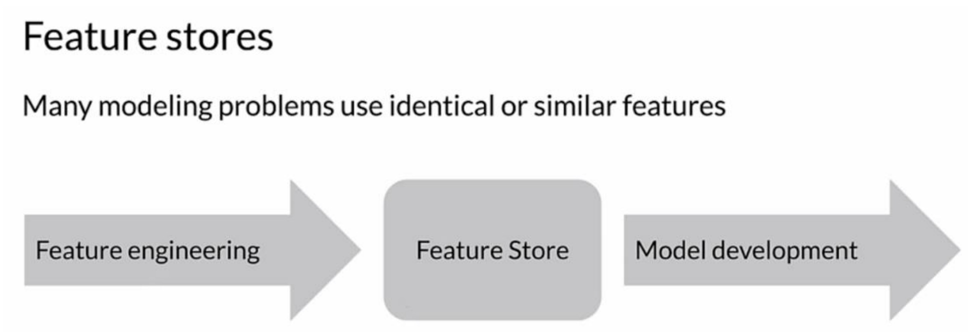

Key points

- Iteratively update and fine-tune schema to adapt to evolving data
- How to deal with scalability and anomalies
- Set schema environments to detect anomalies in serving requests

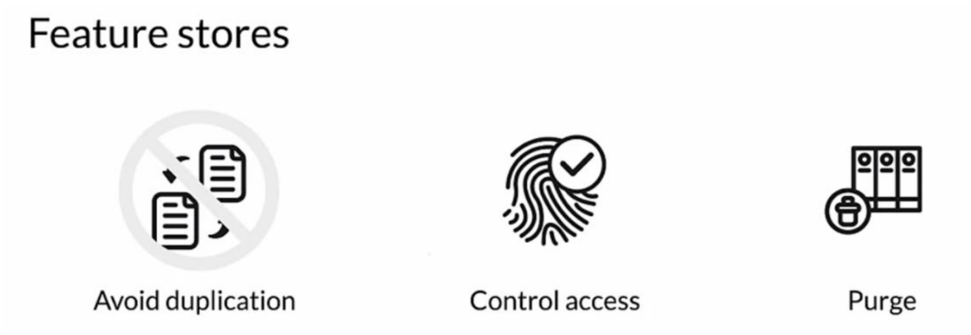
Feature Stores



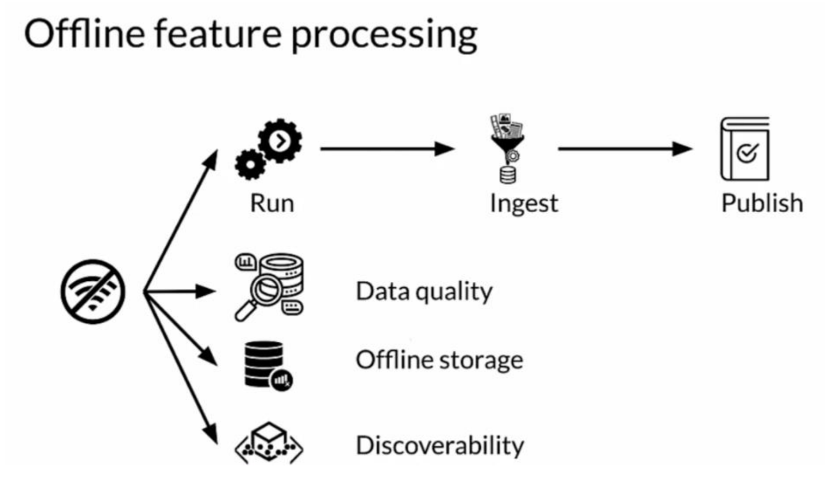
- Now let's turn to the question of where we should store our data. We'll start with a discussion of feature stores.
- A **feature store is a central repository for storing documented, curated and access controlled features.**
- Using a feature store enables teams to share, discover and use highly curated features.
- A feature store makes it easy to discover and consume that feature and that can be both online or offline for both serving and training.
- *Own Note – Good article on what feature stores are: https://www.phdata.io/blog/what-is-a-feature-store/*



- People often discover is that many modeling problems use identical or similar features. So often the same data is used in multiple modeling scenarios.
- In many cases, a feature store can be seen as the interface between feature engineering and model development.
- Feature stores are **valuable centralized feature repositories that reduce redundant work**. They are also valuable because they enable teams to share data and discover data that is already available.
- You may have different teams in an organization with different business problems that they're trying to solve. But they're using identical data or data that's very similar.
- For these reasons, **feature stores are becoming the predominant choice for enterprise data storage**, for machine learning, for large projects and organizations.

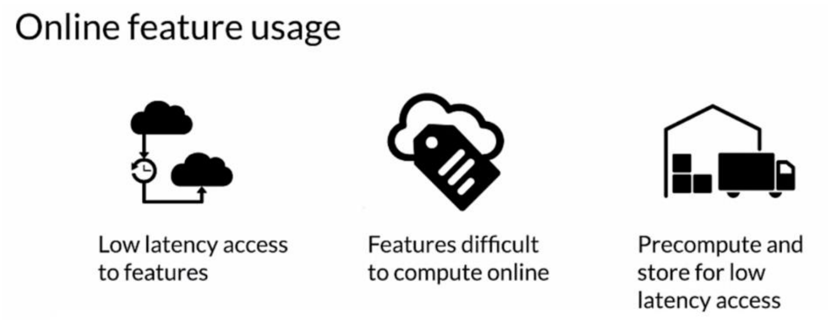


- Feature stores often allow transformations of data so that you can avoid duplicating that processing in different individual pipelines.
- The access to the data in feature stores can be controlled based on role based permissions.
- The data in the feature stores can be aggregated to form new features.
- It can also be anonymous sized and even purged for things like wipeouts for GDPR compliance



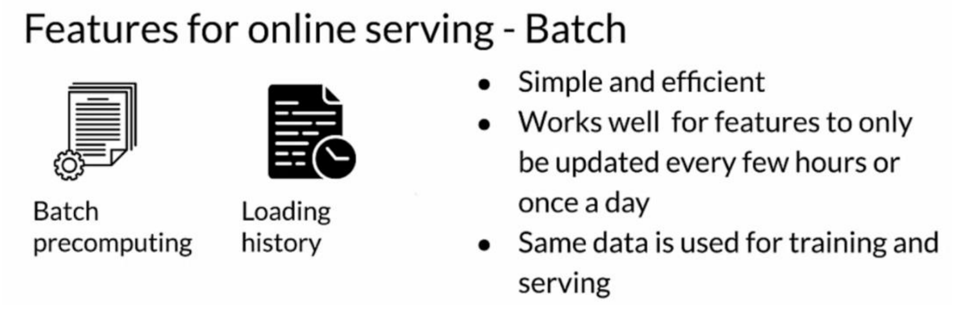
- Feature stores typically allow for feature processing **offline** that can be done on a regular basis, maybe on a CRON job, for example.

- Imagine that you're going to run a job to ingest data. And then maybe do some feature engineering on it and produce additional features from it. Maybe for feature crosses, for example
- These new features will also be published to the Feature store. You can integrate that with monitoring tools
- As you are processing and adjusting your data, you could be running monitoring offline.
- Those processed features are stored for offline use.
- They can also be part of a prediction request by doing a join with the data provided in the prediction request to pull in additional information.
- Feature metadata allows you to discover the features that you need. The metadata that describes the data that you are keeping is a tool (and often the main tool) for trying to discover the data you're looking for.


Online feature usage

Low latency access to features
Features difficult to compute online
Precompute and store for low latency access

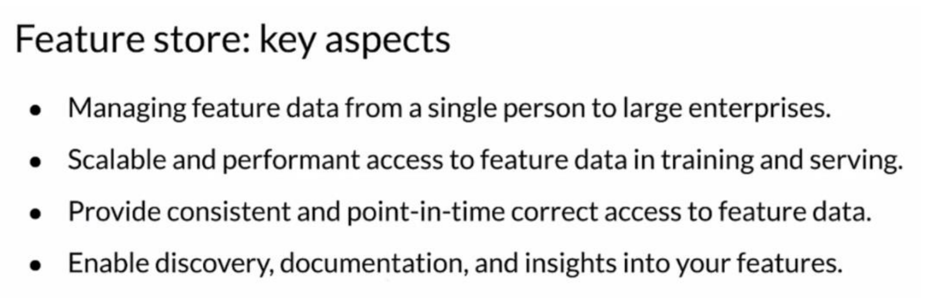
- For online feature usage where predictions must be **returned in real time**, the latency requirements are typically fairly strict.
- You're going to need to make sure that you have fast access to that data.
- If you're going to do a join, for example, maybe with user account information along with individual requests, that join has to happen quickly. That's good, but it's often difficult to compute some of those features in a performant manner online.
- So, having pre computed features is often a good idea. If you pre-compute and store those features then you can use them later. And typically that's at fairly low latency.


Features for online serving - Batch

- Simple and efficient
- Works well for features to only be updated every few hours or once a day
- Same data is used for training and serving

Batch precomputing
Loading history

- You can also do this in a batch environment. Again, you don't want the latency to be too long, but it probably isn't as strict as an online request.
- For pre computing and loading, especially things like historical features, it tends to be fairly simple. For historical features in a batch environment, it's also usually straightforward.
- However, when you're training your model, you need to **make sure that you only include data that will be available at the time that a serving request is made**.
- Including data that is only available at some time after a serving request is referred to as **time travel**, and many feature stores include safeguards to avoid that.
- You might do pre computing on a clock every few hours or once a day. You're going to use, of course, that same data for both training and serving in order to avoid training-serving skew.

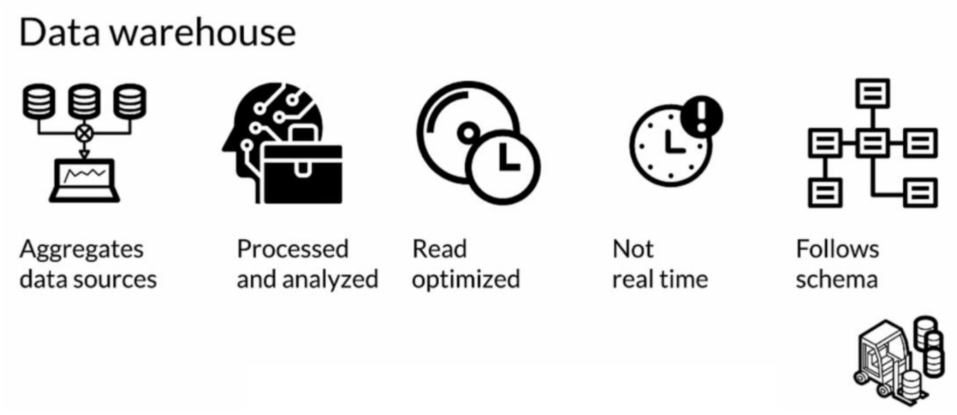


Feature store: key aspects

- Managing feature data from a single person to large enterprises.
- Scalable and performant access to feature data in training and serving.
- Provide consistent and point-in-time correct access to feature data.
- Enable discovery, documentation, and insights into your features.

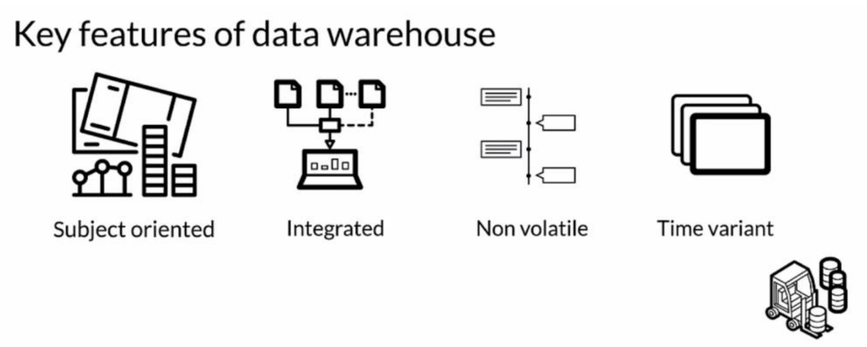
- The goals of most feature stores are providing a unified means of managing featured data that can scale from a single person up to large enterprises.
- It needs to be performant and you want to try to use that same data, both when you're training and serving your models.
- You want consistency and point in time correct access to feature data. You want to avoid making a prediction, for example, using data that will only be available in the future when serving your model.
- In other words, if you're trying to predict something that will happen tomorrow, you want to make sure that you are not including data from tomorrow. It should only be data from before tomorrow.
- Most feature stores provide tools to enable discovery and to allow you to document and provide insights into your features.

Data Warehouse

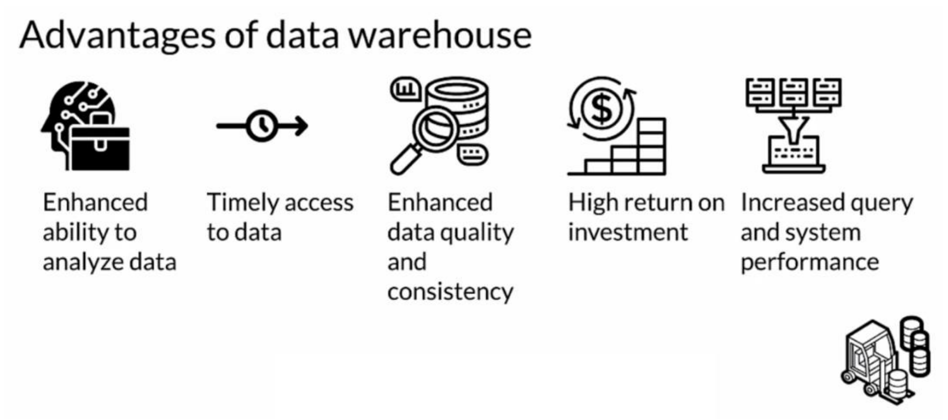


Data warehouse

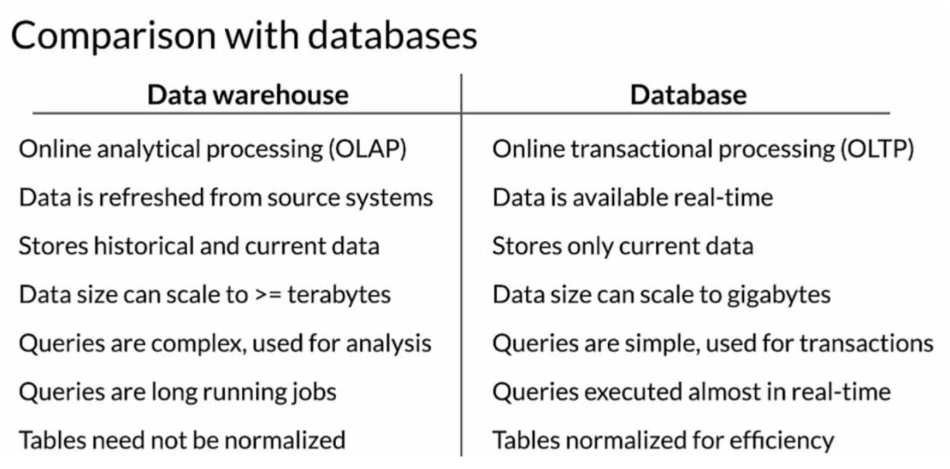
- Now let's discuss another popular enterprise data storage solution – Data warehouses.
- Data warehouses were originally developed for big data and business intelligence applications, but they're also valuable tools for production ML.
- A data warehouse is a technology that **aggregates data from one or more sources** so that it can be processed and analyzed.
- A data warehouse is usually meant for long running batch jobs and their storage is optimized for read operations.
- Data entering into the warehouse may not be in real time.
- When you're storing data in a data warehouse, your data needs to **follow a consistent schema**.



- A data warehouse is **subject oriented**, and the information that's stored in it revolves around a topic.
- For example, data stored in a data warehouse may be focused on organization's customers or vendors etc.
- The data in data warehouse may be collected from multiple types of sources, such as relational databases or files and so forth.
- Data collected in a data warehouse is usually time-stamped to maintain the context of when it was generated.
- Data warehouses are **nonvolatile**, which means the previous versions of data are **not** erased when new data is added. That means that you can access the data stored in a data warehouse as a function of time and understand how that data has evolved.


Advantages of data warehouse
Enhanced ability to analyze data
Timely access to data
Enhanced data quality and consistency
High return on investment
Increased query and system performance

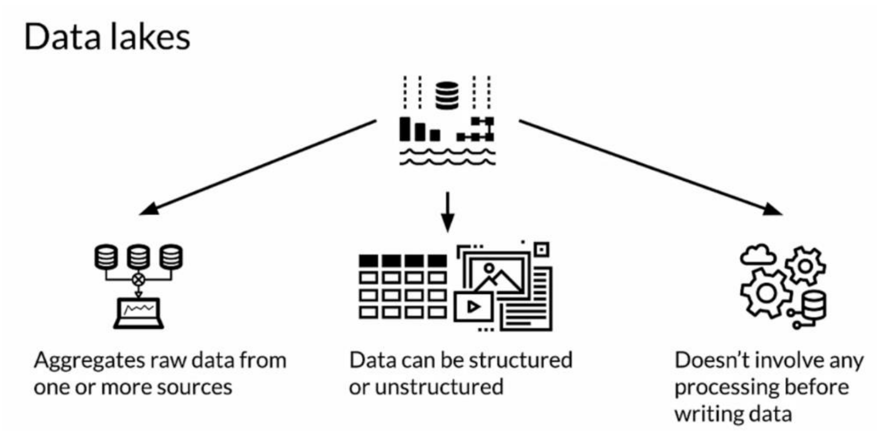
- Data warehouses offer enhanced ability to analyze your data. By time stamping your data, a data warehouse can help maintain contexts.
- When you store your data in a data warehouse, it follows a consistent schema and that helps improve the data quality and consistency.
- Studies have shown that the **return on investment for data warehouses tend to be fairly high for many use cases.**
- Lastly, the read and query efficiency from data warehouses is typically high, giving you **fast access** to your data.



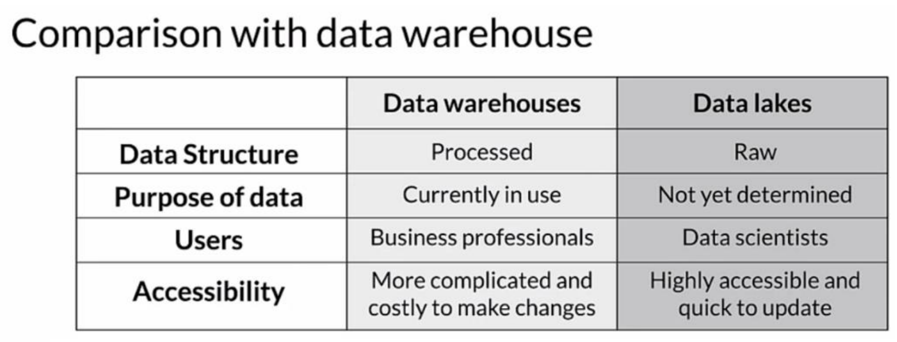
Comparison with databases

Data warehouse	Database
Online analytical processing (OLAP)	Online transactional processing (OLTP)
Data is refreshed from source systems	Data is available real-time
Stores historical and current data	Stores only current data
Data size can scale to >= terabytes	Data size can scale to gigabytes
Queries are complex, used for analysis	Queries are simple, used for transactions
Queries are long running jobs	Queries executed almost in real-time
Tables need not be normalized	Tables normalized for efficiency

- **A natural question is, what's the difference between a data warehouse and a database?**
- **Data warehouses are meant for analyzing data, whereas databases are often used for transaction purposes.**
- Inside a data warehouse, there may be a delay between storing the data and the data getting reflected in the system. But in a database, **data is usually available immediately after it's stored**.
- Data warehouses store data as a function of time, and therefore, historical data is also available.
- Data warehouses are typically capable of storing a larger amount of data compared to databases.
- Queries in data warehouses are complex in nature and tend to run for a long time, whereas queries in database are simple and tend to run in real time.
- **Normalization** is not necessary for data warehouses, but it should be used with databases.


Data lakes
Aggregates raw data from one or more sources
Data can be structured or unstructured
Doesn't involve any processing before writing data

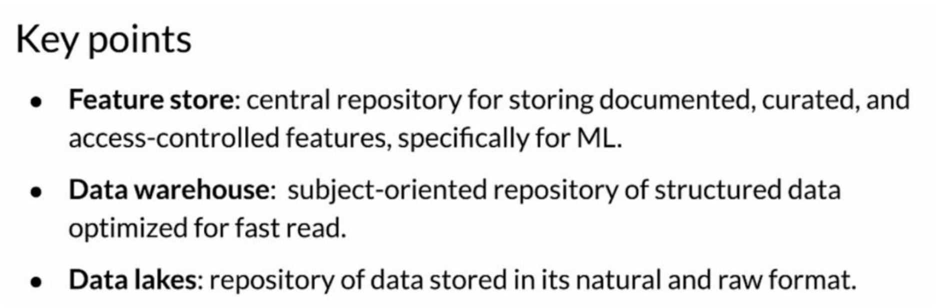
- A data lake is a system or repository of data stored in its **natural and raw format**, which is usually in the form of **blobs** or **files**.
- A data lake, like a data warehouse, aggregates data from various sources of enterprise data.
- A data lake can include structured data like relational databases or semi-structured data like CSV files, or unstructured data like a collection of images or documents, and so forth.
- Since a data lake store data in its raw format, they **don't do any processing**, and they usually **don't follow a schema.**



Comparison with data warehouse

	Data warehouses	Data lakes
Data Structure	Processed	Raw
Purpose of data	Currently in use	Not yet determined
Users	Business professionals	Data scientists
Accessibility	More complicated and costly to make changes	Highly accessible and quick to update

- How does a data lake differ from a data warehouse? Let's compare the two.
- The primary difference between them is that in a data warehouse, data is stored in a consistent format which follows a schema, whereas in data lakes, the data is usually in its raw format.
- In data lakes, the reason for storing the data is often not determined ahead of time. This is usually not the case for a data warehouse, where it's usually stored for a particular purpose.
- Data warehouses are often used by business professionals as well, whereas **data lakes are typically used only by data professionals such as data scientists**.
- Since the data in data warehouses is stored in a consistent format, changes to the data can be complex and costly. Data lakes however are more flexible, and make it easier to make changes to the data.



Key points

- **Feature store**: central repository for storing documented, curated, and access-controlled features, specifically for ML.
- **Data warehouse**: subject-oriented repository of structured data optimized for fast read.
- **Data lakes**: repository of data stored in its natural and raw format.

References

Data Versioning:
https://dvc.org/
https://git-lfs.github.com/

ML Metadata:
https://www.tensorflow.org/tfx/guide/mlmd#data_model
https://www.tensorflow.org/tfx/guide/understanding_custom_components

Chicago taxi trips data set:
https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew/data
https://archive.ics.uci.edu/ml/datasets/covertype

Feast:
https://cloud.google.com/blog/products/ai-machine-learning/introducing-feast-an-open-source-feature-store-for-machine-learning
https://github.com/feast-dev/feast
https://www.gojek.io/blog/feast-bridging-ml-models-and-data