The Logical Feature Store: Data Management for Machine Learning

Published 31 July 2023 - ID G00792810 - 67 min read

By Analyst(s): Georgia O'Callaghan

Initiatives: Analytics and Artificial Intelligence for Technical Professionals; Data Management Solutions for Technical Professionals; Evolve Technology and Process Capabilities to Support D&A

Feature engineering for machine learning (ML) is often a siloed process, fraught with inefficiencies and risk of error. Data and analytics technical professionals can implement a logical feature store to better manage data for ML to ensure reusability, reliability and reproducibility of features.

Overview

Key Findings

- Productionizing ML models remains a significant challenge for enterprises looking to scale up ML operations, with only half of models making it into production.
- ML teams spend a disproportionate amount of their time sourcing data and engineering features for ML over actual model development and optimization. This is a siloed and inefficient process, leading to duplication of work and inconsistent feature definitions.
- For organizations early in their ML journey, feature transformation pipelines are created on a per-model basis, making them brittle and difficult to manage, govern and monitor. As a result, training-serving skew, data drift and poor data quality often impact model performance.
- Many organizations that attempt to build a feature management solution in-house do not fully understand the scope and required capabilities. This can result in over- or underestimating the complexity and comprehensiveness of the solution.

Recommendations

Data and analytics technical professionals seeking to accelerate model development and enable the smooth transition of models into production must:

- Manage features for ML across the enterprise by implementing a logical feature store appropriate for the organization's ML portfolio. Promote consistency of feature definitions and feature reuse to accelerate model development and productionization.
- Facilitate model audit and retraining by capturing version-controlled transformation logic, feature lineage and other metadata. This allows training and testing datasets to be reproduced with point-in-time correctness.
- Prevent ML model degradation by monitoring data that serves models in production. Detect and remediate factors that negatively impact model performance, such as data drift, outliers, poor data quality and missing data.
- Avoid attempting to build a comprehensive logical feature store from scratch unless the enterprise has considerable maturity and resources (expertise, time, funding, tools and infrastructure) and ML requirements that cannot be met by existing product offerings.

Analysis

Machine learning (ML) is a subset of artificial intelligence (AI) that utilizes mathematical models to extract knowledge and patterns from data. Businesses develop ML models to make predictions or to gain insights into a specific problem. In most circumstances, ML models will not perform well with raw data. Instead, data must be transformed and normalized, as appropriate. This process of sourcing, acquiring, cleaning and transforming raw data needed for ML model development is known as feature engineering. Features are the inputs to ML models and are provided to give the context the algorithm needs for analysis and learning.

When developing ML models, data scientists spend most of their time preparing and curating the data they need (empirical data suggests 47.4%, but some say upward of 60%) and less time creating and testing the models themselves. [1,2] Due to the time, effort and skill put into selecting and engineering features for ML, they are some of the most highly curated and refined data assets in the business, yet they are often the most poorly managed.

Enterprises with mature adoption of ML have put measures in place to facilitate smooth and efficient deployment of models into production, often designing and implementing custom technologies, infrastructures and procedures. However, for the typical organization in the early stages of ML experimentation, making the leap to productionizing their first models will not be an easy feat.

Indeed, according to Gartner's AI in Organizations Survey, the average proportion of ML prototypes that successfully make it to production is 54%. [3] Of organizations surveyed, only 10% could get 75% or more of their prototypes into production. The process from prototype to production was reported to take an average of 7.3 months per project. Breaking this down, only 12% of respondents were able to deploy models in less than three months, with 47% of respondents taking between six and 24 months. However, organizations that are good at deploying models are scaling to thousands or hundreds of thousands of models in production. So, what is the secret?

The Struggles of Feature Engineering

To accelerate time to production and maintain model performance, organizations need to tackle some of the main data-related challenges D&A teams face throughout the ML development life cycle:

- Lack of reusability
- Difficulty with reproducibility
- Issues with reliability

Within early ML adopters, data scientists are typically provided with organizational data to experiment with via one-time data dump into a sandbox or local environment. To supplement this data, they may also pull from external sources, such as publicly available or third-party datasets. Using this data, the data scientist will expend considerable effort selecting and engineering features. This is an iterative process that continues throughout model development while feature transformations are modified, new features are added, existing ones are dropped and so forth.

This approach is not only time consuming, but it is extremely inefficient. It results in a lot of siloed work, and feature engineering efforts are not reusable across ML projects. Given how valuable data scientists are within organizations for their skills and expertise, it is wasteful for them to spend time duplicating efforts already made by others. There can be considerable overlap in features used by ML models. Therefore, reusing these features across models leads to faster development times. However, with the current siloed state of feature engineering practices, data scientists are unlikely to be cognizant of features developed by others within their organization. Even if data scientists seek out and discover relevant previous work, they are unlikely to reuse features because the absence of lineage and other metadata results in a lack of trust and understanding.

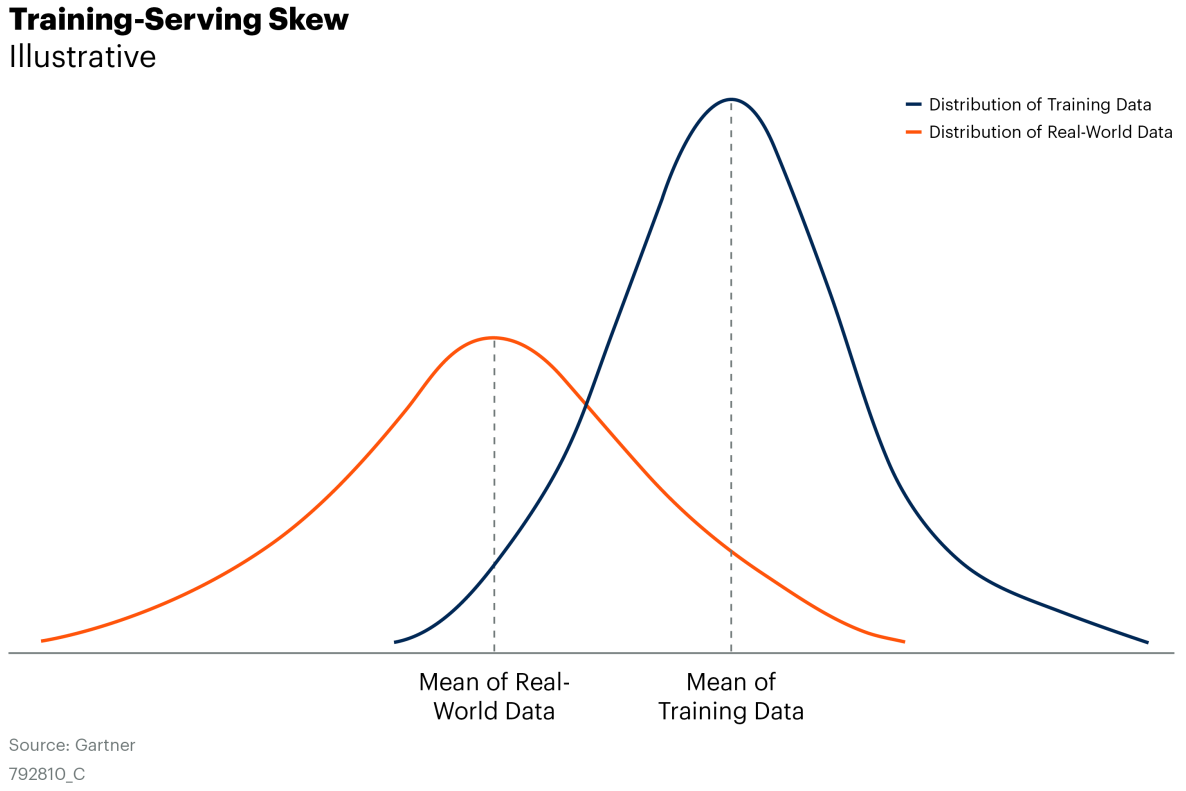
Following ML model prototype optimization, companies early in their ML adoption often rely on data engineers to build production data pipelines on a per-model basis or by building team-specific infrastructure. To do this, the scripts and related metadata are handed off to a data engineer to replicate in the production environment. The data engineer then establishes data pipelines that connect directly to source systems to serve transformed feature values to ML models in batch or real time, as appropriate for the use case. Once-off data pipelines are hard to maintain and keep consistent. Moreover, the lack of reusability of pipelines results in further duplication of work.

In addition to these issues with reusability, these practices make it extremely difficult to reproduce the exact datasets that were used for training or testing models, or data used to generate a specific prediction. Even if data scientists retain the code used for feature engineering, they also need details about how the data was sourced and sampled, time windows to establish point in time correctness, and so on to recreate a feature set. The ability to reproduce datasets is relevant for model retraining, model audit, explainable AI and AI governance use cases. With the rise in interest in explainable AI and with data that suggests organizations retrain their ML models at least once a quarter, the ability to recreate datasets with point-in-time correctness has become increasingly important. [1]

Once models have been implemented in production, predictions may not be consistent with the prototype model — a problem that is very difficult to diagnose and fix. One common cause is training-serving skew, where data in production differs from the data used for model training, which manifests as significant differences in the distributions of the data (as illustrated in Figure 1). When the assumption of training-serving data consistency is broken, model accuracy decreases.

Data scientists can see an immediate drop in the performance of their model when launched into production due to factors like poor data sampling. This is where data used for training did not accurately represent the real-world data. Model performance can also drop when data engineers translate feature engineering code from languages favored by data scientists (e.g., Python) to production standard code (typically SQL). Even when the greatest care is taken during the reengineering process, subtle differences in these languages can result in differences in feature calculations that are difficult to diagnose. Additionally, training-serving skew can emerge over time due to poor data quality that makes its way into the data pipeline (like missing data or outliers) or because aspects of the real world or business change and the data drifts.

Figure 1: Training-Serving Skew



Feature Management: The Logical Feature Store Approach

Features for ML are some of the most highly curated data assets in organizations, yet they are often the most poorly managed. Therefore, the goal for all enterprises seeking to mature their ML operations must be to implement a feature management practice to achieve the following:

- **Reusability:** The ability to share features and use them across multiple ML workloads. This includes sharing within and across D&A teams in the organization.
- **Reproducibility:** The ability to recreate entire training and test datasets used for ML model development with point-in-time correctness.
- **Reliability:** The ability to maintain ML model performance by serving data in production that is consistent with the data used for ML model development and by detecting when these diverge.

The logical feature store is a feature management solution that aims to break down silos, promote collaboration, reduce time spent on feature engineering and avoid duplication of work by sharing resources among data analysts. It's a flexible term because it allows organizations to define and enable just the capabilities they need to meet the requirements of their ML portfolio. Having a common understanding of how data was transformed to create features for ML can increase consistency, trust and transparency; promote explainability; and enable direct comparisons between models.

The logical feature store is a feature management solution that allows organizations to define and implement capabilities proportionate to the requirements of their ML portfolio.

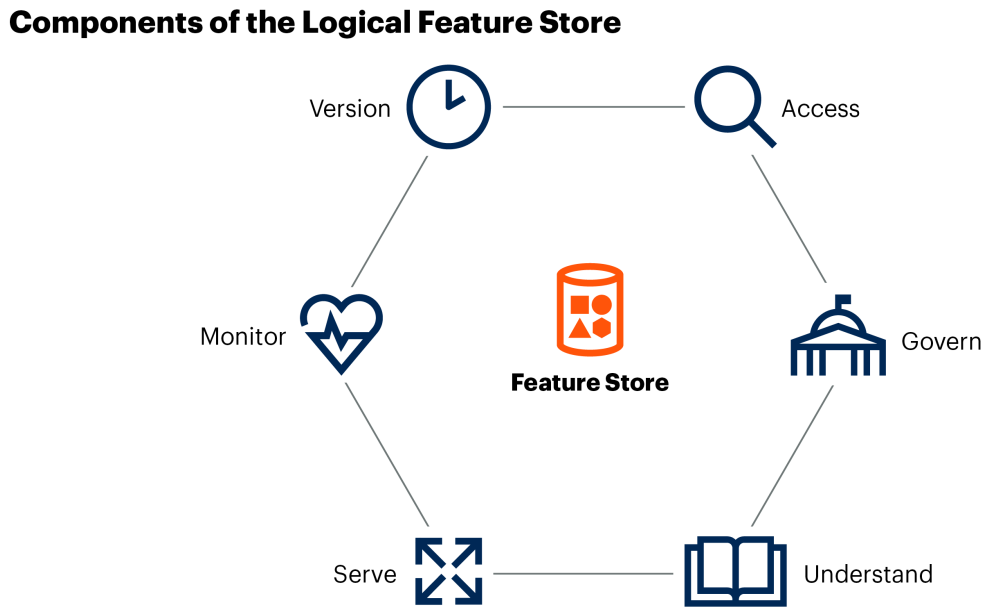
Technical professionals should seek to introduce the following components to their feature management solution:

- **Store:** The feature management solution can contain a database or repository that meets the enterprise's storage requirements for scalability of capacity, high availability, high throughput and low latency of retrieval. Features in the store must have associated time stamps to allow datasets to be created without data leakage and to recreate datasets with point-in-time correctness.
- **Access:** Features must be accessible to data scientists and other privileged people within the organization to use for future model development. For example, providing a UI with a searchable catalog of features allows them to be discovered, shared and reused across ML workloads.
- **Govern:** Policies and processes need to accompany this solution to govern access to features in a way that strikes a balance between democratizing and securing the data.
- **Understand:** The catalog must be populated by relevant metadata that makes features understandable to promote trust in the data, such as feature descriptions and definitions, lineage, versioned transformation code, dependencies, and feature transformation cadence.

- **Serve:** The feature management system should enable users to easily pull data to create training and test datasets for ML model development. At the same time, the system needs to serve and accommodate the organization's different ML use cases in production, which may be batch or on-demand.
- **Monitor:** Features should be constantly validated and monitored to prevent poor quality from disrupting models in production, thus decreasing accuracy and overall performance. Outgoing features serving models in production must also be monitored for data drift and training-serving skew by comparing up-to-date feature descriptive statistics (e.g., means, standard deviations and metrics of normality) to those collected during model training.
- **Version:** The ability to version data used for ML model development is a key component of managing the continuous cycle of ML model debugging, retraining and improvements that occur over time. The system must also have the ability to serve the right version of the right features to the right models in production.

These capabilities, summarized in Figure 2, make up the logical feature store approach. The logical feature store approach is technology agnostic, and yet several pieces of technology are employed to compose this solution, as needed to meet the organization's needs. Implementation of the logical feature store approach is discussed in the next section.

Figure 2: Components of the Logical Feature Store



Source: Gartner
792810_C

Gartner

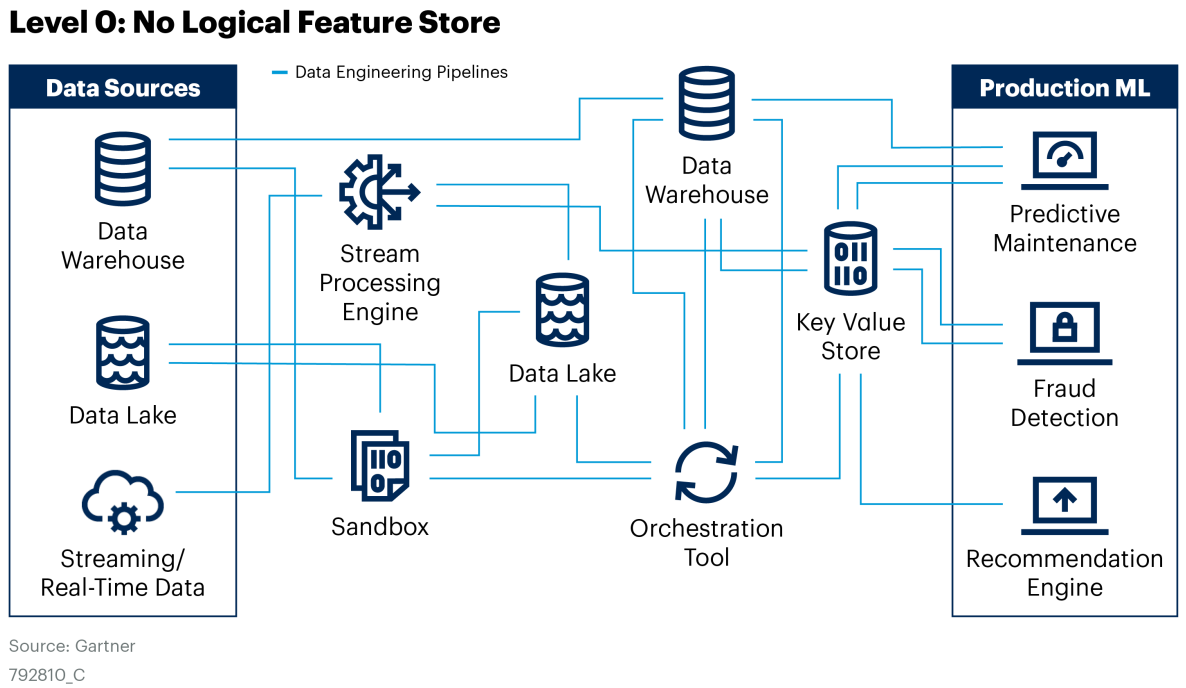
Implementing the Logical Feature Store

The complexity and scale of ML use cases within the organization will drive logical feature store requirements. The more comprehensive the logical feature store needs to be, the harder it will be to build in-house. This section provides four sample architectures ranging from no feature management capabilities to more comprehensive solutions. Some large enterprises may even utilize combinations of these approaches.

Level 0: No Logical Feature Store

Companies early in their ML adoption often rely on data engineers to build production data pipelines on a per-model basis or by building team-specific infrastructure. Figure 3 illustrates this scenario, where each line represents a separate data engineering pipeline feeding a model in production.

Figure 3: Level 0: No Logical Feature Store



Custom-made pipelines, as illustrated in Figure 3, are brittle, error-prone and difficult to scale. They are time consuming to implement and manage, resulting in massive engineering overhead, high costs and technical debt. In some cases, data scientists package up the feature transformation code within the model function themselves (e.g., containerized or carried out within an AWS Lambda layer in Keras). This means that the preprocessing code is carried along with the model and executed at runtime. As a result, raw data is transformed into features every time the model is run. This can be slow and computationally expensive. This approach is only suitable for batch transformations and often results in duplication of pipelines across training and inference pipelines, and across ML use cases.

However, there are still situations where attempting to create a logical feature store solution would be excessive given the ML use cases within the organization. A logical feature store is not needed, for example, for ML use cases where features are:

- Not time dependent
- Needed only for batch serving
- Computationally inexpensive

- Graph-based features (because these can't be precomputed)
- Known by the client requesting the ML prediction

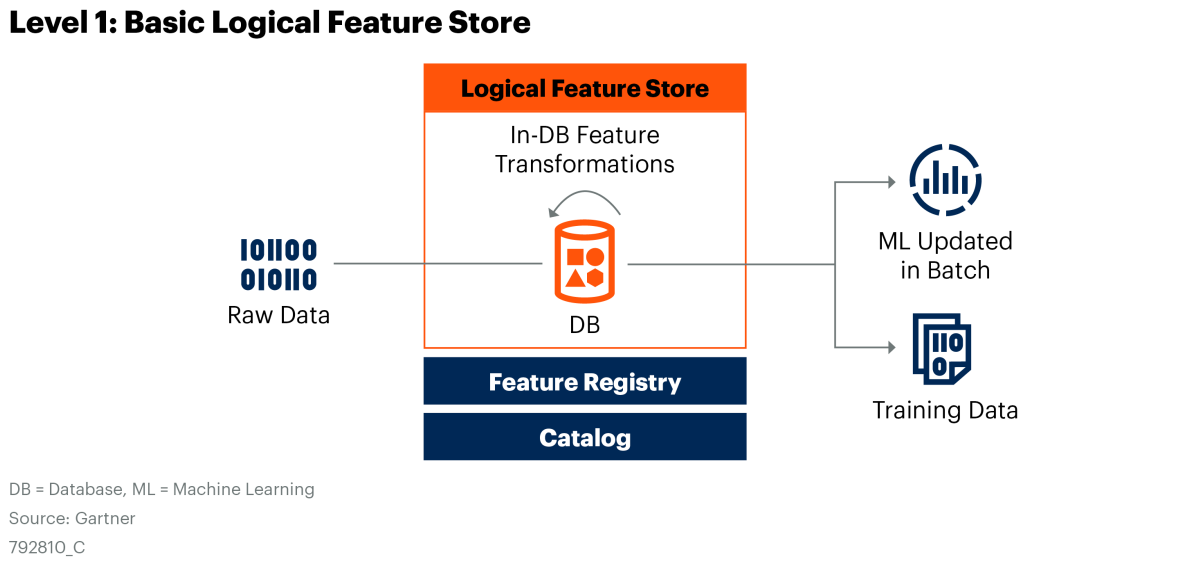
The number of models in production is often the best indicator of the need for a logical feature store. Issues with manually managing once-off data pipelines and associated artifacts only become more pronounced as organizations attempt to scale up ML operations. Creating custom-made data pipelines or wrapping preprocessing code up in the model might be sustainable with one, 10 or possibly 20 models in production. However, beyond that, teams should mature into developing at least a basic logical feature store.

Level 1: Basic Logical Feature Store

For organizations with only batch features and batch ML scoring use cases, a basic logical feature store implementation will suffice. A basic implementation is a scalable repository that can hold feature data. It is known in the industry as an offline feature store. This could be any relational database (DB) technology already in use within the organization. For this, data scientists can collaborate with data engineers, data architects and database administrators (DBAs) to create feature tables for their use cases within existing DBs. These tables serve as the "store" component of the logical feature store capabilities shown in Figure 2.

Provisioning feature tables within a DB allows features to be defined once and be production ready as soon as they have been committed to the store. From this store, data scientists can pull training sets for model development and run model scoring jobs in batch, as needed (see Figure 4). This creates a consistent view of features across development and production, reducing the likelihood of training-serving skew.

Figure 4: Level 1: Basic Logical Feature Store



Provisioning feature tables within a DB appears to be a relatively simple solution. However, it has not yet addressed some of the challenges a logical feature store attempts to address:

- How are new features added to the repository?
- Who creates and orchestrates the data pipelines, and who is responsible for maintaining them?
- How do you ensure good data quality?
- How are features discoverable and understandable?
- How are features versioned?

Ideally, the responsibility of creating production-ready feature transformation code is taken away from the data engineer and kept with the data scientist who developed the feature. However, this is not always feasible in this implementation.

Certain modern database (DB) technologies allow for in-DB analytics. Google BigQuery and AWS Redshift, for example, have capabilities for data scientists to create feature tables through automated or manual feature engineering. Both BigQuery and Redshift have functionality that allows data scientists to perform feature engineering tasks and model development using SQL and a series of commands and functions the vendors have created. BigQuery ML also integrates with Vertex AI, and Redshift ML integrates with SageMaker (Google and AWS's respective data science and machine learning [DSML] platforms) for more advanced model building and deployment capabilities, including model and feature registries. In the case of BigQuery ML, data does not leave the DB, but for Redshift, data is first exported to an S3 bucket for processing before it is returned to the DB and stored as features. See Google's Automatic Feature Preprocessing and Manual Feature Preprocessing for further information on feature engineering in BigQuery. See Amazon's Using Machine Learning in Amazon Redshift for further information on feature engineering in Redshift.

In both BigQuery and Redshift, data scientists require SQL skills to utilize all of the capabilities provided. In the case of Snowflake, however, data scientists can use Python, Java or Scala to conduct feature engineering and model development through Snowpark. Snowpark supports external notebooks and certain DSML and machine learning operationalization (MLOps) platforms, including Dataiku, Deepnote, Hex, SageMaker Studio Lab, DataRobot, Domino Data Lab, and H2O.ai. These applications access Snowflake via the Snowpark Application Programming Interface (API), which translates Python code to SQL queries to retrieve and transform the necessary data. Transformed feature data within data frames can be written back to Snowflake tables to store the data for historical purposes and for reuse by future data scientists.

Data science teams should investigate whether their existing DB technologies have built-in analytics capabilities. For many modern cloud DBs, in-DB analytics capabilities are missing. This drives data science teams to consider alternative solutions, such as depending on data engineers to create feature tables for them, or adopting new DB technologies. In addition to the options mentioned above, organizations adopting new technologies to enable data scientists to create their own feature tables can also consider DBs that have been specifically optimized for feature storage and retrieval, with data-scientist-friendly interfaces, like FeatureBase and RonDB (by Hopsworks).

It is also possible that in-DB analytics capabilities are present, but data scientists are not allowed to use them. In these cases, data scientists will still rely on data engineers and DBAs to create tables for them. From an agility perspective, this is not ideal. However, DBAs and other data management professionals may restrict or block access to these capabilities for many reasons, including governance of data, cost management, and controlling storage volume and compute workloads. For more on the impact of leveraging existing DB technologies for the storage of feature data, see the Reducing Impact on Data Management section.

To create the logical feature store, data science and supporting data management teams need to resolve these issues around who will create features and manage feature pipelines. They must also create a process to monitor, identify and resolve data quality issues as they arise. When conducting in-DB analytics, data quality is more likely to fall under the purview of data engineers and DBAs supporting the main DB. However, to complete the logical feature store, these teams still need to address versioning, accessibility and understanding.

For versioning, the team may create a repository of versioned feature transformation code, using something like Git, to go alongside the dedicated tables of feature data created in their existing database. Definitions, in this case, refer to the description of the feature in a human-readable format. Although transformation code shows the exact mode of calculating the feature (e.g., the code used to average, aggregate or derive certain values), the feature definition is an equivalent explanation in words. This provides transparency and traceability, which helps avoid silent changes to definitions or transformations that can impact downstream models. Versioning also allows data scientists and data engineers to roll back to earlier versions of features, if necessary.

Finally, data scientists need a way to discover existing features in the store in order to reuse them for future development projects. The easiest way to do that is to leverage a data catalog or metadata management system already in use within the organization. Examples of these include Alex Solutions Data Platform, Collibra Data Intelligence Cloud, data.world, Informatica Enterprise Data Catalog and OvalEdge Data Catalog. A more comprehensive list of representative vendors can be found in Table 3 of Deploying Effective Metadata Management Solutions.

Integrating a data catalog into the logical feature store involves:

1. Connecting the data catalog to the DB in question that holds the feature tables

2. On-boarding the relevant technical information about these tables

3. Tasking data scientists with curating the metadata and enriching it with additional relevant information, such as feature definitions

However, not every organization has a data catalog, and not every organization with a data catalog has an agile process that allows it to onboard data rapidly as new tables are created. There can also be resistance from data management professionals who maintain the data catalog, who may not see data-scientist-derived data as assets that should be included. In such cases, the data science team will need to create an alternative means of cataloging their features to create a data dictionary and glossary of terms and definitions. They may be able to achieve this through custom code that links the feature transformation code stored in Git to table information extracted from the DB and enrich this with other relevant information. They may also lean on open-source cataloging tools that they can customize to serve a similar purpose.

Cataloging the data (whether manually, through a custom solution or using an out-of-the-box tool) makes it searchable and understandable to other data scientists, and can give data management experts insight into the tables created for ML use cases. As with all analytics use cases that involve a data catalog as their starting point, accessing data for ML model development will be a two-step process:

1. Searching the catalog to discover relevant data

2. Accessing the DB directly to pull relevant data for offline analysis or running models directly against the DB

The approach described in this section outlines the components necessary to establish a logical feature store for batch scoring and batch feature use cases by leveraging existing technologies within the organization. However, there are still some challenges that this solution does not overcome.

For example, in-DB analytics helps only in cases where required data assets are already available in a DB. In many cases, data scientists need to perform experimental data analysis on previously untransformed or unexplored data in order to determine its relevancy or usefulness in solving a business problem with ML. Also, as previously mentioned, this setup only allows data scientists to run batch model scoring on batch data. To support more advanced ML use cases that require streaming data, on-demand model scoring and real-time feature calculations, data science teams should consider the more comprehensive logical feature store approaches described in the following sections.

Level 2: Moderate Logical Feature Store

ML model scoring can be run on-demand (e.g., manually refreshed by an individual), in batch (i.e., scheduled to automatically update on a set schedule — this could be every quarter, month, week, day, etc.) or in real-time (e.g., when some event/application/etc. requests an immediate response based on new information). These use cases represent increasing levels of complexity in ML serving. With these use cases, the logical feature store must now also be capable of serving the most up-to-date feature values to ML models on-demand and with extremely low latency.

Due to limitations of today's database technologies, such as data warehouses and data lakes, no single database technology can do both scalable storage and low-latency retrieval to the level required to support these use cases. Therefore, in addition to the offline feature store described in the previous section, the logical feature store now requires infrastructure to support an "online" feature store. This must be optimized for low-latency serving of features for on-demand ML scoring or even low-latency serving of prescored ML models.

To meet requirements for storage capacity and scalability, the offline feature store can be architected in much the same way as described above: with tables within a traditional data warehouse (such as Snowflake, BigQuery or Redshift) or within a dedicated partition data lake (like Amazon S3). This serves as a source of data for new ML model development (training and test datasets), in addition to serving ML models that are updated in batch or on-demand. The ability to create and recreate datasets with point-in-time correctness from the offline feature store is critical for training, retraining and model audit. For more on point-in-time correctness see Note 1.

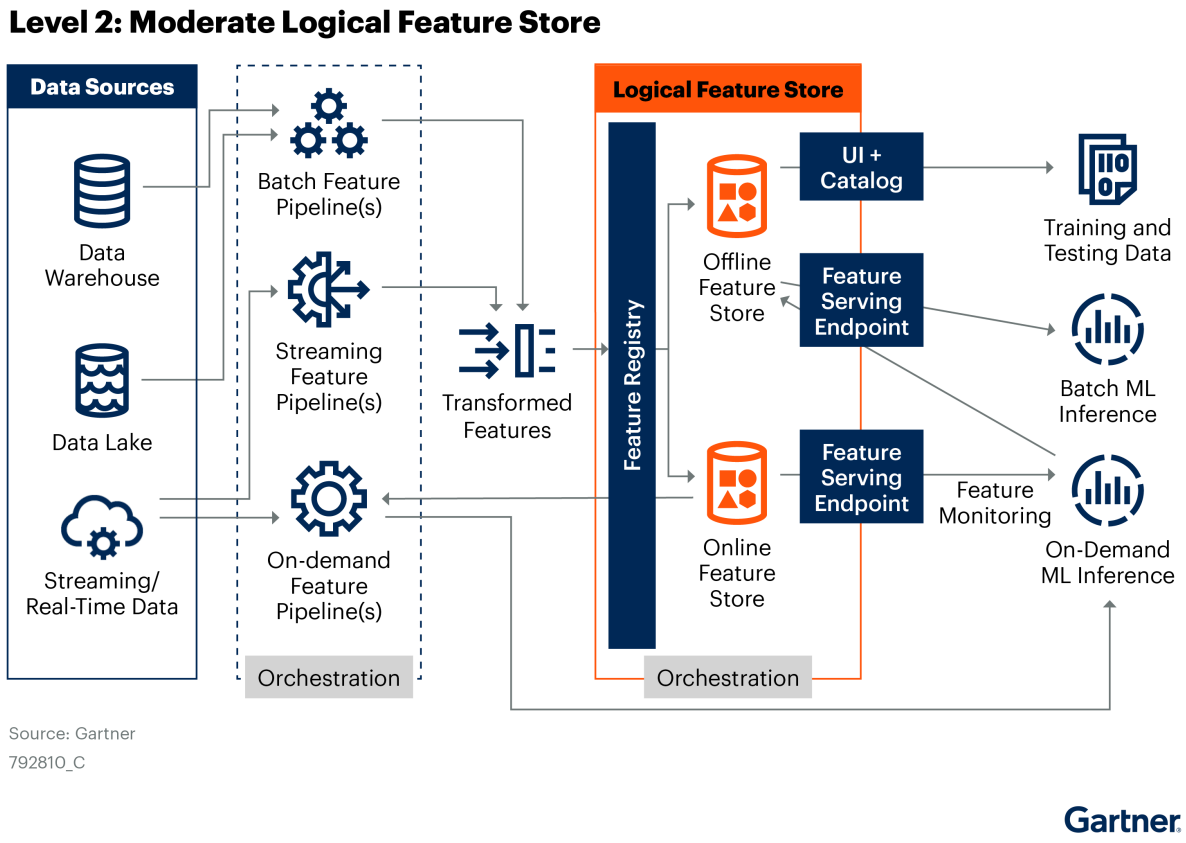
Online feature stores that facilitate low-latency retrieval of feature values or precomputed predictions can be implemented with key-value stores like Amazon DynamoDB, Redis or Cassandra because they have great tail latencies and are optimized for very high write loads. The selection of which technology to use will depend on factors like budget, in-house skills and resources for managing and optimizing the store. For example, managed services like DynamoDB are simple to use but can be very expensive to run, compared to Redis, which is cheaper but requires manual optimization. Ideally, the online feature store should only retain the latest feature values, while offline feature stores contain months' or years' worth of relevant historical data. Limiting storage within the online feature store to the latest data and the most recent model scorings provides an up-to-date view of the world while keeping storage to a minimum to conserve performance.

The split offline/online feature store design has evolved out of necessity to accommodate different ML needs (one providing scalability and the other low latency), but there are disadvantages to this approach. There are increased overheads with managing and maintaining two database technologies. First, certain feature values will be duplicated across the two stores, which increases storage costs. Second, information needs to be reconciled between the two sources to prevent data leakage and ensure data consistency across the two stores. Therefore, data science teams without on-demand scoring use cases should avoid introducing this unnecessary complexity into their architecture.

For this set of use cases, the data science team needs an application with a user interface (UI) that contains at least the following capabilities in order to have a minimally viable logical feature store, as illustrated in Figure 5:

- Feature registry
- Searchable feature catalog
- APIs for serving features (both for offline and online)
- Write-back permissions to the online feature store
- Identity access management (IAM) or integration with the existing IAM platform of the organization

Figure 5: Level 2: Moderate Logical Feature Store



Within this implementation, cataloging capabilities are no longer deferred to an enterprise data catalog but are contained within a catalog owned and managed by data science teams. When adding new features to the feature store, data scientists must have a mechanism for registering feature definitions and transformation code to the feature registry through the UI. At this level of complexity, a manually managed GitHub account, for example, is not sufficient for feature versioning. The feature registry may be backed by Git or equivalent versioning capabilities, but this integration should be orchestrated and not pushed in an ad hoc way.

The program should pull relevant metadata from the feature definition code to populate certain fields within the catalog, and data scientists can further curate these definitions and add metadata. To support this, data scientists ideally need to follow a standard feature definition convention. Metadata in the catalog should include feature freshness, feature transformation cadence and whether the feature is available in both online and offline feature stores, or only offline. Feature freshness refers to the last time the feature was updated, whereas feature transformation cadence indicates the feature update schedule. For example, feature cadence for batch features may be monthly, daily, hourly, every five minutes, and so on.

The key thing about the implementation of the logical feature store in this case is that it is not designed to facilitate the creation, orchestration and management of data transformation pipelines. This requires a much more sophisticated application and greater integrations with existing data management owned infrastructure. It would also require a software development kit (SDK) for data scientists to work in. This is discussed further in the Level 3: Comprehensive Logical Feature Store section.

In practice, features may be created directly in the offline feature store by data scientists using in-DB analytics capabilities, as described in the Level 1: Basic Logical Feature Store section. However, in many cases, this may not be possible. For example:

- In-DB analytics capabilities may not be present or allowed
- Relevant data may be distributed across multiple DB systems
- The ML use case may require streaming data or other raw data not already in a relational DB, such as raw sources contained within a data lake.

Here, data scientists must work with data engineers to build and orchestrate data transformation pipelines to persist transformed features to the offline feature store and keep them up to date.

This sounds like the scenario described in the Level 0: No Logical Feature Store section. The difference is that data pipelines are no longer being created on a per-use-case basis and are not directly serving models in production, as shown in Figure 5. Instead, computed features are persisted to the offline store, making them reusable across ML projects. When new data becomes available, it should be appended to the relevant tables in the offline feature store. For this, data engineers will need to establish and manage data pipeline orchestration with something like Apache Airflow or Dagster. For guidance on orchestrating data pipelines, consult Building Data Orchestration and Workflows. The catalog connects data available in the physical tables of the offline feature store with feature definitions, transformation code and other metadata, as described above.

To use these features in production for on-demand serving use cases and make them reusable across these ML workloads, they must also be available in the online feature store. However, to reduce costs and preserve optimal performance of the DB for low-latency retrieval, only features necessary for on-demand serving use cases should be persisted to this store. Similarly, only the latest value of each feature should be stored. To achieve this, data scientists must define which features must be in the online feature store, and pipelines feeding new feature values into the online store must replace the out-of-date values over time.

As a result, the process of publishing features to the online feature store and keeping them up-to-date must be carefully thought through. If this can only be defined at the initial pipeline building stage, data scientists lack the flexibility of later pushing features from the offline store to the online store without requiring data engineers to alter data pipelines. One solution to this is to persist features to the offline feature store first, and then have a separate pipeline and orchestration that pushes the necessary features and updates from the offline to the online feature store. This process should be carefully monitored to ensure consistency across stores.

It may be possible to provide an abstraction layer via the feature registry to enable data scientists to trigger and configure the materialization of features in the online feature store themselves. This depends on the level of sophistication of the application supporting the logical feature store. However, this would require data engineers to build the underlying data pipeline mechanism that can be triggered by a call from the program. It would also require software developers to build out the necessary integrations and application functionality.

Special consideration should also be paid to use cases where on-demand feature calculations are needed as part of the model scoring process. This use case reflects *true* real-time ML and is extremely difficult and computationally expensive to do. It is also very difficult to optimize all elements of the pipeline to achieve the extreme low latency that these use cases often demand. Response times are prolonged because certain stages of the pipeline must be done in sequence.

As a result, when people talk about real-time ML, they are often referring to low-latency retrieval (as low as 10 milliseconds (ms) to 15 ms of prescored predictions that can be stored in a DB optimized for low-latency. These predictions are themselves updated in batch. Here, there is a trade-off between prediction accuracy given the freshness of the features and optimizing for storage and cost. Although it might be possible to continually update these stored predictions as often as every five seconds, this is not advisable because the costs of maintaining this solution would be astronomical.

In addition, certain features can only be known at model trigger time — like a client's current location. Or they can only be partially computed ahead of time and require an on-demand or "last mile" calculation — like a client's current distance from home (a combination of current location and home location retrieved from the online feature store). When on-demand calculations are truly necessary, they must be carried out by a separate processing framework, like Apache Flink. These features may require retrieval of feature values from the online store to complete the calculation. In most cases, the feature vector requested by the ML model will consist of a combination of precalculated features from the online feature store and these on-demand features.

The problem with this approach is that this external pipeline must be developed and maintained separately from the feature store. This increases overheads and runs the risk of inconsistency in feature values and data loss. One critical step to mitigate the potential for data loss and inconsistency in this implementation is to persist a copy of the externally calculated feature values to the offline feature stores for historical reference as part of this pipeline. This is also illustrated in Figure 5. These on-demand features should also be defined within the feature registry. However, the metadata in the catalog should reflect that these are not materialized in the online feature store. Gartner recommends that the business perform a cost-benefit analysis to determine whether they have sufficient justification for such ML use cases.

As the creation and orchestration of data pipelines here largely falls under the purview of data engineers, they will be responsible for the quality of the data entering the logical feature store. Data scientists are subject matter experts on the features they have created. Therefore, data scientists should assist data engineers in defining relevant and actionable data quality profiles and rules. This ensures that the necessary checks can be integrated into the data pipelines to catch poor-quality data before it enters the logical feature store. Quality assessment and remediation steps may be embedded manually within the extract, transform, load (ETL) code itself, or handled by an external data quality tool, such as those provided by Talend, IBM, Informatica, CluedIn and OvalEdge. More information on implementing data quality practices and relevant vendors in this space can be found in Data Quality Fundamentals for Data and Analytics Technical Professionals.

In addition to evaluating the quality of data entering the logical feature store, it is also necessary to monitor feature output over time to identify things like data drift. Monitoring data drift is essential to prevent training-serving skew from impacting the performance of models in production. In this implementation of the logical feature store, this analysis is done periodically by data scientists, rather than automatically using some application. This requires data scientists to capture characteristics of feature sets used to train their models and compare these to data serving models in production on a regular basis. The analysis may be conducted in a program like Grafana. If data scientists detect meaningful differences in the samples, this should prompt them to retrain and tune their model on the new data. A more in-depth review of monitoring ML in production is available in Launch an Effective Machine Learning Monitoring System.

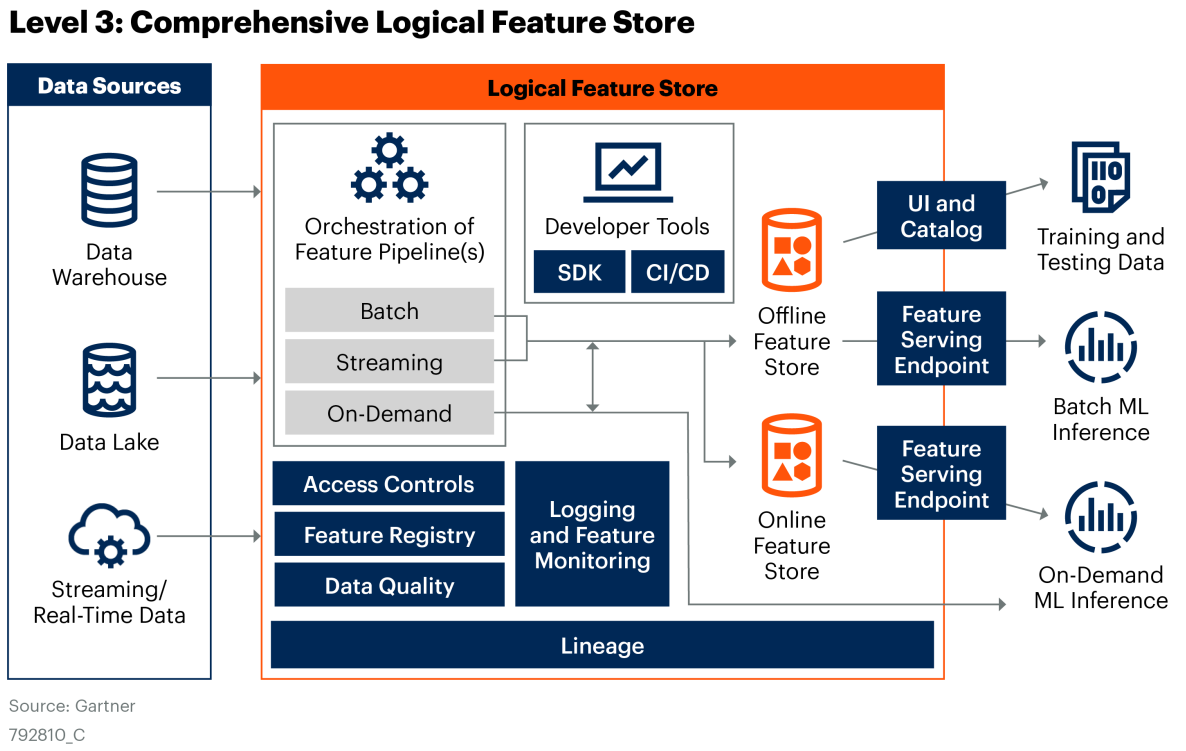
Level 3: Comprehensive Logical Feature Store

As organizations scale their ML portfolio to hundreds and thousands of models in production, at a certain point they will benefit from data scientists creating their own data transformation pipelines. This allows data scientists to work independently, without relying on, or burdening, data engineers. Level 3 extends the capabilities described in Level 2: Moderate Logical Feature Store to include defining and orchestrating the pipelines necessary to transform raw data into features, persist them into the online and offline stores, and keep them up to date over time.

The application described above should be expanded to give data scientists an SDK where they can conduct feature engineering following DevOps engineering best practices. For example, they may conduct exploratory feature engineering in a development workspace environment before promoting finalized features to production and setting the feature update cadence. This must be supported with the following capabilities, as illustrated in Figure 6:

- Feature versioning
- Data lineage
- Automatic time travel capabilities
- CI/CD capabilities for managing the deployment of features

Figure 6: Level 3: Comprehensive Logical Feature Store



The application supporting the logical feature store should support connecting to the necessary batch data sources within the organization, such as their data lakes and data warehouses, and streaming sources. On the back end, it may also leverage existing compute infrastructure, such as in-DB calculations or Spark, and existing storage infrastructure for the materialization of feature datasets, like S3, DynamoDB or Redis. In order to materialize features, data scientists need to be granted write-back permissions to the underlying storage layer.

When data scientists push features into production, they commit the feature definitions to the feature registry. The registry should extract relevant metadata to populate the catalog, as before. However, in this implementation, the interface should also allow data scientists to directly trigger the materialization of features into the offline feature store and set a backfill period (time travel capabilities). In addition to creating a historical feature set, they should also be able to set a feature update cadence (e.g., weekly, hourly, every 5 minutes) and be able to specify which features should also be materialized to the online feature store. The application has to have internal orchestration capabilities tied to this configuration that actually triggers the updates and manages dependencies (through the use of Directed Acyclic Graphs [DAGs]).

Data scientists will also need to be able to define features in the registry that have to be calculated on-demand (including last-mile calculations) and create the required transformation pipelines through the application supporting the logical feature store. However, these features will not be materialized to either the online or offline feature stores because they cannot be precomputed. When such a feature is requested, there will be instances where a last-mile calculation has a dependency on a feature in the online feature store, requiring retrieval of certain feature values in order to complete its calculation. On-demand feature calculations need to be served directly to the ML model in production that requested the feature vector for scoring, in addition to other features from the online feature store. There must also be a mechanism for writing back the on-demand feature value to the offline feature store for historical reference. This supports audit and retraining use cases.

Feature lineage not only needs to display the journey of the data from the source to the feature repository; it must go beyond this to include what features, and what version of these features, are serving which models in production. Lineage can also indicate how fresh or stale a feature is based on feature transformation cadence and when it was last updated. For example, if new data for a particular feature is transformed in batch every seven days and added to the repository, knowing whether the data was last updated one or six days ago could make a big difference to a data scientist seeking to include it within a training dataset. The data scientist may want to trigger a manual refresh before pulling the dataset.

Feature monitoring aims to safeguard the quality, consistency and reliability of data for ML by providing a view of the health of the data. Feature data can be validated on ingest based on user-defined schemas or other structural criteria and adherence to certain data quality rules related to missing data, outliers and more. Automated data monitoring identifies data drift and other data quality issues quickly to enable timely remediation, which mitigates issues' effects on model performance. This is achieved by automatically calculating and storing metrics on the features at baseline (during model development), such as descriptive statistics and distribution information (skewness and kurtosis). Data scientists should have the ability to view feature health within the logical feature store UI. Feature health metrics displayed within the UI can include those about incoming data (such as completeness and accuracy) and outgoing data (how current feature data compares to baseline properties). Logical feature store developers may also include capabilities for data scientists to configure alerts on these metrics. Regardless of whether alerts are automated or not, it is critical that there is an established practice or workflow where the appropriate stakeholders (for example, the individual who developed the model and/or the feature) are notified of flagged issues so they can investigate and remediate them.

In addition, feature management architectures must be maintained and optimized over time. This is achieved by monitoring operational metrics related to feature storage (e.g., availability, capacity, memory utilization, and staleness), feature serving (e.g., query performance and latency, loads, and error rates) and data processing (e.g., job success rate, throughput, processing lag and rate). There also must be at least one dedicated owner of the logical feature store who will be responsible for managing usage, governance and costs. This is described further in the Reducing Impact on Data Management section. Figure 6 shows the end-to-end feature management capabilities contained within this comprehensive logical feature store implementation.

Developing a comprehensive logical feature store solution requires considerable time and data engineering and software engineering time. Despite this, some teams prefer to build their own solution to have more control and avoid vendor lock-in. Most teams start building a feature management solution without realizing the full scope of what they need to build. Often, teams overlook crucial capabilities such as monitoring, serving infrastructure and orchestration.

When planning for a comprehensive logical feature store, it is key to balance future proofing against developing a strategy that is too complex and difficult to manage. There is a risk that the solution may be tailored to support only known ML use cases,and when future use cases emerge, this is no longer suitable (e.g., use cases that have a more demanding lookup latency). This can require organizations to modify or design their logical feature store to accommodate the new use cases.

In many cases, large enterprises will combine several of the approaches described in this section, rather than implementing one enterprise-level logical feature store solution. This is because the cost and complexity of building an enterprisewide logical feature store may outweigh the potential benefits for some organizations. For example, organizations can have departments or teams with more advanced data science capabilities, skills and requirements that are doing a lot of production ML, and other areas that are still experimenting and conducting proofs of concept (POCs). In such cases, organizations may implement the appropriate solution for the use case, such as a basic implementation in one department and a more comprehensive solution in another.

This would allow data scientists in more advanced teams to create and manage their own data pipelines. In other areas they would still rely on data engineers. It can also be necessary to separate physical implementations by geographical region or by security boundary, distributing the storage of feature data. This can happen when organizations must adhere to certain data residency rules or when they have governance policies that prohibit access to certain data assets by role, line of business, and so on. In such cases, the head of data science or other strategic-level decision maker must decide whether to also separate the feature registries or maintain one, unified catalog of features.

Finally, a technical professional — such as a senior data scientist — must be assigned ownership of the feature repository within the logical feature store system and be responsible for its management. This person should regulate how new features are added to the repository to ensure that users (mostly other data scientists) include all relevant metadata. This will help to prevent the repository from turning into a swamp. This will involve training users in best practices and, ideally, implementing a system where certain metadata fields are mandatory, which prevents new features from being pushed to the repository without satisfying a list of requirements.

Logical Feature Store vs. Feature Store: Build vs. Buy

As defined earlier, the logical feature store is a feature management solution, providing the capabilities described in Figure 2. It's a flexible term because it allows organizations to implement a solution proportionate to their level of maturity, volume and complexity of ML use cases, and other specific ML requirements. The previous section, Implementing the Logical Feature Store, describes how organizations can achieve this across four levels and provides sample architectures for each implementation.

The practice of implementing a feature management solution proportionate to the organization's ML requirements is the logical feature store approach.

Large organizations such as Uber, Twitter, Spotify, DoorDash and Gojek (with Google) have developed their own feature management systems in-house, and there are several vendors offering solutions in this space. [4] The industry recognizes these solutions as "feature stores" or "feature platforms." Due to the complexity of creating and orchestrating the various logical feature store implementations described above, many organizations will lack the skills and resources to build it from scratch themselves. For this, many want to leverage an off-the-shelf product to simplify their solution.

The distinction between logical feature store and the feature store/platform is made by Gartner because the industry generally conflates the solution, feature management, with the product, feature store. It is important to separate the practice of feature management from the feature store product space for the following reasons:

1. There is considerable variability within the vendor landscape as to what constitutes a feature store or feature platform, and what capabilities it should have. This causes confusion and sets the wrong expectations.

2. Many organizations with basic ML use cases need a feature management solution, but buying a feature store product will be overkill because it provides capabilities they do not need.

3. Organizations can have specific requirements that a feature store product cannot meet, such as particular high-throughput, low-latency serving requirements.

4. Some organizations want to avoid vendor lock-in and choose to construct their ML platforms out of a combination of open-source, best-of-breed and custom components.

Organizations should take a step back from the idea of feature-store-as-a-product. Instead, they should start by determining the requirements of their ML portfolio and the capabilities needed to effectively manage features for their use cases. The practice of implementing a feature management solution proportionate to these requirements is the logical feature store approach. A certain degree of future proofing is advised for the solution. However, it is also important to strike a balance between solution comprehensiveness and over-engineering, which creates unnecessary complexity and increases management overheads.

Organizations considering an off-the-shelf feature store product should be aware that this is a developing space and not without its limitations, with no standard product design or set of capabilities. For example, the term “feature store” has been applied to solutions ranging from feature storage only to more end-to-end managed solutions. With existing feature store or feature platform solutions, some vendors focus on solution completeness. Others focus on optimizing specific capabilities. The market is further complicated by a certain degree of overlap with the capabilities of other data management, DSML and MLOps tooling.

The main variants in feature store products currently offered by vendors are described in Table 1, in order of increasing feature management comprehensiveness. These levels also reflect increasing complexity and, therefore, the difficulty associated with building equivalent functionality from scratch in-house.

Table 1: Range of Capabilities Found Across Feature Store Vendor Solutions

(Enlarged table in Appendix)

Vendor Solutions Offering	Description	Benefits
Storage	The feature store product contains only a dedicated repository of curated features from which training and test datasets can be pulled for ML model development.	■ Decreases time spent on individualized feature engineering efforts. ■ Standardizes feature definitions. ■ Promotes collaboration and the sharing of features.
Storage + Online Serving	In addition to storing features offline, some feature store products serve features to ML models in production for online scoring use cases. Some provide service-level agreements (SLAs) for online feature serving to meet clients' low-latency requirements. Care must be taken when selecting a product because some vendors have better SLAs than others.	■ Provides a consistent view of features in development and production, preventing training-serving skew. ■ Supports on-demand ML scoring use cases.
Feature Transformations + Storage + Online Serving	Few solutions have the capacity to orchestrate the transformation of raw data into features and ingest them directly into the online and offline feature stores. Those that do can manage various update cadences and both batch and streaming data. However, there are still limitations in their ability to do on-demand feature transformations.	■ With feature transformations contained within the feature store solution, rather than outsourced to other ETL pipelines, versioned transformation code and related metadata can be captured to record the lineage of features. ■ Updating features according to their appropriate cadence ensures an up-to-date world view.
Feature Transformations + Storage + Online Serving + Monitoring	Certain feature store implementations calculate and store descriptive statistics alongside the features they contain. They use these benchmark metrics to detect things like poor data quality and data drift, to monitor features serving models in production, and to flag when these deviate significantly from baseline statistics.	■ Automated detection and the ability to diagnose issues facilitates model maintenance. ML model performance degrades over time. Much of this can be attributed to issues with data, such as changing real-world context, adversarial attacks and the ingestion of poor-quality data not caught by preceding data pipelines.

Source: Gartner (July 2023)

If selecting a vendor solution to serve as part of the organization's logical feature store, ML teams should start by determining their feature management needs based on their ML use cases. Next, they must carefully discern the capabilities being offered by their shortlisted platforms and conduct POCs to determine suitability. It is important to note, too, that requirements can evolve over time as ML use cases grow in volume and complexity. Therefore, organizations may go through more than one iteration of build vs. buy for their logical feature store. For example, an organization at Level 2: Moderate Logical Feature Store may select and implement an open-source feature store product. As needs change over time, it may build additional capabilities in-house to support this implementation or integrate with other open-source tools, like a monitoring tool. At a certain point, this solution may become too complex to manage and maintain, and the organization may consider transitioning to an enterprise feature store product. Vendors in the space, along with alternative solutions, are discussed in the Evaluating the Product Landscape section.

Current off-the-shelf offerings come with their own set of challenges. For instance, the implementation of open-source solutions can be complex and require a lot of skill. With enterprise solutions, many vendors are heavily involved during the initial setup phase because these solutions can require a degree of customization to fit the enterprise's infrastructure. As vendors work with clients to establish and implement their feature stores, they continue to make improvements to their technology, including expanding their list of native data source connectors. As feature store products mature, they should invest in becoming easier and more flexible to deploy.

Another factor that concerns potential buyers of feature store products is their ability to support the latency, availability and throughput requirements of their particular use cases. Benchmarking studies have been conducted to demonstrate the online serving latency and availability of certain feature stores under varying degrees of throughput. A list of such studies can be found in Note 2. For example, one study reported a p50 latency of 14 ms for features served from their online feature store for a fixed throughput, and 20 ms for a single on-demand feature calculation. [5] p50 is standard terminology in such benchmarking studies, meaning that in 50% of cases, features were served with this speed or less. The p99 latencies reported were 55ms and 70 ms for online feature store serving and on-demand feature calculations, respectively (i.e., 99% of cases were equal to, or less than, this latency). Overall, they achieved ~99.9999% availability on average.

Latency, availability and throughput requirements are equally important when determining the suitability of a certain product for the organization's ML use cases. Data scientists should scrutinize the available benchmarking studies to determine if the experimental conditions map on to their own ML requirements. In addition, organizations should request POCs from shortlisted vendors to assess whether their specific requirements can be supported by the platform. If their requirements cannot be met, and if the expected value of the ML solution justifies the cost and effort involved, this may prompt the ML team to build a more performant solution in-house.

Strengths

- **Accelerates ML model development by making features reusable across ML workloads:** Data scientists can query logical feature stores to discover features, pulling subsets of data to create training and test datasets for model development. Without this, data scientists typically spend vast amounts of time engineering the features they need for ML model development. These efforts result in siloed work, making sharing and governing features almost impossible. Because there can be considerable overlap in features used by ML models, this practice is inefficient and leads to duplication of work. Logical feature stores aim to break down these silos, promote collaboration between data scientists and enable reuse of features across ML models. Increased access to data curated for ML removes the need for data scientists to find and prepare data, freeing up more time for ML model development and accelerating the process.

- **Reduces training-serving skew by enabling data scientists to write production-ready code:** With certain logical feature store implementations, individual data scientists can define and contribute features in their preferred programming languages (e.g., Python or R). This code is versioned, and metadata is captured to provide feature lineage. The traditional practice of moving models from development to production takes several months and involves re-creating feature engineering code in production languages (e.g., C++ or SQL). This is vulnerable to issues like training-serving skew caused by subtle differences in feature transformation code, which reduces model accuracy. With a logical feature store, you can shift the responsibility of creating production-ready feature transformation code from data engineers to data scientists. Once committed to the logical feature store, features are instantly made available across development and production environments.

- **Increases model accuracy by ensuring the reliability of features serving models in production:** A consistent view of features across these environments reduces the likelihood of training-serving skew and increases the reliability of the data serving ML models. A case study by Tecton found that its feature store platform improved the accuracy of online features from 95%-97% to 99.9%. [6] The ability to serve this data to ML models in production improved prediction accuracy of existing models by 2% to 20%. In addition, some feature store solutions include the ability to monitor data quality, distributions and other characteristics. Automating the detection of data drift and other such issues helps to prevent model degradation that results from changes in the underlying data serving models in production.

- **Supports the reproducibility of datasets used for model development:** When versioning, time stamps and time-travel capabilities are integrated into logical feature store solutions. This allows datasets to be recreated with point-in-time correctness, a key component of managing the continuous cycle of ML model debugging, retraining and improvements. This also facilitates model governance, audit and explainability.

- **Accelerates time to production:** A case study by Tecton found that it reduced time to deploy new models by more than 50%, from two to four months down to one month. [7] Tecton has also reported that its feature store, integrated with the Amazon SageMaker platform, can reduce time to production from six months to one to two days. [8] Similarly, Iguazio observed that its platform enabled a client to deploy ML models with 90% less code and six to 12 times faster than its previous approach. [9]

- **Derives real value from ML initiatives:** Feature store products have helped businesses derive real value from ML initiatives, such as recovering revenue by automating certain tasks, faster automated detection of fraud (from 40 minutes to 12 seconds) and reduced false-positive anomaly detection (99% more accurate than a rule-based engine). [10, 11] Additional benefits include reduced storage and compute costs. For instance, Hopsworks reported a 90% reduction in ML portfolio costs as a result of adopting its platform in one client. [12] These costs were associated with storing large volumes of data, as well as compute resources to process this data. A case study by Iguazio found that its AI platform solution, for which a feature store is an integral component, reduced operational expenses by 50%, required 16 times less storage and three to six times fewer compute nodes. [9]

These examples are provided for illustrative purposes and are based on publicly available case study data from feature store vendors. Links to case studies are provided in the Evidence section.

Weaknesses

- **Different definitions of the "feature store" product and its capabilities:** As discussed above, the feature store, as a product, has been defined differently by industry leaders (vendors and organizations who have built their own solutions, like DoorDash and Uber), creating confusion. Feature store product capabilities are yet to be standardized. Some focus on solution completeness and others on optimizing specific capabilities.

- **Enterprises need help implementing logical feature stores:** Logical feature stores require the enterprise to have certain dedicated infrastructures in place and the resources to manage it (e.g., their own Apache Spark cluster for feature transformations and a Redis instance for low-latency serving). Depending on the chosen logical feature store implementation, the organization will need certain skills to orchestrate data pipelines, integrate aspects of the solution, set up monitoring and so on. In some cases, the feature store vendors themselves will be heavily involved in the initial setup and integrating the solution into the enterprise's existing technology stack.

- **Two separate database management technologies are needed to provide storage scalability and low-latency retrieval:** Database management systems have limitations on speed of data transformation and real-time ML serving. Vendor and in-house-built solutions have attempted to overcome this by providing two stores supported by separate database technologies, one optimized for scalability (offline feature store) and the other for low-latency retrieval (online feature store). Despite this, performance for complex real-time transformation and serving use cases may still not meet desired SLAs. This approach also runs the risk of data inconsistency across stores and increases management overheads.

- **Feature lineage is incomplete:** The self-contained catalog within the logical feature store will only include lineage information from the point data is extracted from a source (data warehouse, data lake and so on), transformed and persisted into the store. It will not include lineage pertaining to its journey prior to this — such as how it was originally collected and the ETL it underwent before being ingested into a data warehouse. Likewise, an enterprise data catalog will not contain metadata and lineage related to feature transformations or how features are used by ML models in production. End-to-end lineage could be achieved by connecting an enterprise data catalog tool to the feature store to copy its metadata. However, the cost (time, effort and enterprise data catalog resources) may outweigh the benefit of providing such an end-to-end lineage of features.

- **Feature stores need to be governed and managed:** There is a concern that, without proper management, a feature store could become just another analytical silo within the business. Enterprises that implement a feature store will face governance and maintenance challenges, as with any technology employed by the business. Therefore, the designated feature store owner (such as a data architect, senior data scientist or similar role within the business) must establish and enforce policies and best practices for adding new features, pulling data for model development and connecting data to ML models in production. The feature store owner should promote a collaborative environment where individual data scientists contribute features to the store, providing all of the relevant metadata to make them understandable to others. This individual needs to monitor the feature store capacity and retire features that are not used by models after a defined period of time. This person also needs to ensure consistency across online and offline feature stores.
- **Impact on data management technology, cost and storage:** Because logical feature stores recruit existing data management technologies within the organization, there will be a resulting cost for storage, compute and serving of features that will impact the data management function. Without close collaboration between data scientists and data management professionals (such as data architects, data engineers and database administrators) during the design, implementation and running of the logical feature store, there is considerable risk that the logical feature store will incur unexpected and steep costs.

Guidance

Technical professionals seeking to increase the efficiency of ML model development within their organizations, maintain the performance of ML models in production and enable things like feature and ML model audit must implement an effective feature management system.

> **The point here is not "to feature store or not to feature store" — it is the implementation of a feature management solution (1) that is feasible to the business and (2) that satisfies the needs of the business's particular ML portfolio — this is the "logical" feature store approach.**

Evaluating the Product Landscape

Very large enterprises like Uber and DoorDash have been successful in creating their own custom logical feature store implementations. [4] However, these organizations already had a great deal of ML maturity, available resources and expertise. The skill, time and cost of building an infrastructure that would be as feature complete as one provided by a feature store vendor would be considerable. The many component parts of the solutions described in Implementing the Logical Feature Store would need to be architected, integrated, optimized, orchestrated and managed.

Technical professionals such as data engineers, software developers and data scientists will face numerous challenges that have already been encountered and overcome by feature store developers. Therefore, while it may be possible to build a custom feature management system, it is not advisable for small or midsize enterprises with fewer resources and less experienced teams. It is likely that the costs of this approach will outweigh the benefits of achieving a fully customized and self-hosted solution. Some estimates suggest that it would take three dedicated full-time employees between six to nine months to build an internal logical feature store. [7] And this is only to get it up and running — it still has to be maintained long-term and evolve with changing use cases and requirements. D&A teams should interpret this estimate with caution as it assumes the presence of certain skills and resources within the organization. Because of the complexity, time, resources and skills involved, many organizations choose to leverage an existing open-source or enterprise-grade solution rather than build their own.

Before considering the acquisition of any new product or tool, data science teams should complete the following steps:

1. Examine the current state of feature engineering and scale of ML operations within the organization.

2. Determine what degree of feature management is needed (i.e., how comprehensive the logical feature store solution needs to be).

3. Review current architecture and tooling to identify existing capabilities and gaps.

4. Decide to build vs. buy based on the information gathered above and the organization’s budget, available resources, skills and expertise.

If the organization ultimately decides to acquire a new tool, it is not limited to stand-alone feature store product vendors but can also consider certain MLOps or DSML platforms. MLOps and DSML platforms, in particular, offer a more end-to-end solution for ML, which includes data preparation and transformation, as well as model serving. However, not all of these tools allow for sharing data engineering efforts across ML workloads. They may also have limited capabilities with regard to data versioning and monitoring. An overview of MLOps approaches can be found in Demystifying XOps: DataOps, MLOps, ModelOps, AIOps and Platform Ops for AI. Table 2 contains a representative list of stand-alone feature store vendors and MLOps/DSML vendors with feature stores built into their products.

Table 2: Sample List of Feature Store Vendors

Stand-Alone Feature Store Vendors	DSML/MLOps Vendors With Built-in Feature Stores
■ Hopsworks by Logical Clocks ■ Tecton ■ Feast by Tecton ■ FeatureByte ■ Featureform	■ 4Paradigm: OpenMLDB* ■ Abacus.AI ■ Amazon SageMaker ■ C3 AI ■ Databricks ■ Dataiku ■ Google Vertex AI ■ H2O.ai ■ Iguazio^ ■ Qwak

*Product not listed on vendor's main website
^Be aware that this vendor was acquired in January 2023 by McKinsey. See: McKinsey Acquires Iguazio, A Leader in AI and Machine Learning Technology.

Source: Gartner (July 2023)

As discussed in Logical Feature Store vs. Feature Store: Build vs. Buy, there is considerable variability in the capabilities of feature management solutions available on the market today. It is beyond the scope of this document to assess and compare the feature store products listed in Table 2.

Gartner recommends that technical professionals evaluating products derive a list of technical requirements based on their ML use cases. They should then request POCs from potential prospects to determine whether these solutions can meet their needs. For example, the requirements may include:

- The list of batch data sources the solution needs to pull raw data from
- The list of real-time and streaming data sources the solution needs to pull raw data from
- Whether feature transformation pipelines should be internal or external to the platform and what platforms are required for writing feature transformation logic
- Feature transformation cadence requirements, such as daily, hourly, every minute or as low as every second or subsecond transformations
- p99 and p50 serving latency requirements, like less than 1 second, 100 ms, 50 ms, or less than 10 second serving latency
- Availability requirements, which may be 99.99% or as high as 99.9999% availability
- Expected load the platform must handle, including number of users, number of use cases and the expected serving load for each use case
- Security features if ML use cases involve sensitive or regulated data assets such as personal identifiable information (PII)
- Requirements for data quality and data drift monitoring and alerting capabilities

The organization's stance on best-of-breed technologies vs. native offerings will also factor into this decision. For example, the ease of which a cloud service provider's native feature store product can be integrated into an organization's existing ecosystem may outweigh the benefits of certain features a best-of-breed feature store product could offer.

Beyond feature store products, whether stand-alone or baked into an MLOps or DSML platform, organizations can consider alternatives like feature engineering platforms, feature engineering packages or libraries, dataset versioning tools, feature definition layer tools, or even semantic layer tools. These help to achieve some of the goals of feature reusability, orchestrating feature transformation pipelines and so on. They are not a replacement for the full breadth of logical feature store capabilities described in this document. Examples of these are listed in Table 3.

Table 3: Alternative Feature Engineering Ecosystem Products

(Enlarged table in Appendix)

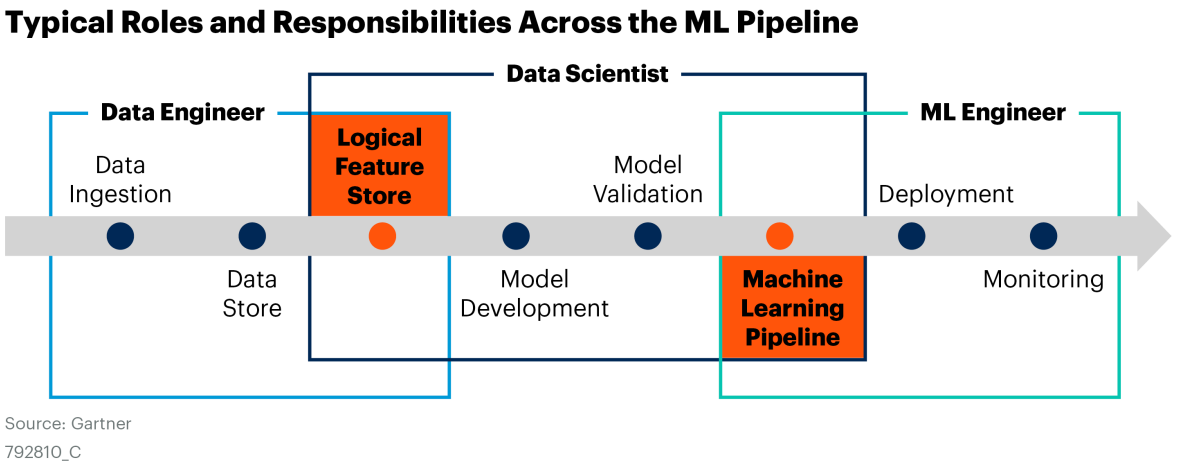
Product Type	Sample Vendors
Feature engineering	■ Enrich Intelligence Platform by Scribble Data ■ Kaskada^ ■ fal.ai (also known as "Features and Labels") ■ dotData ■ Mobilewalla
Feature engineering packages and libraries	■ tsfresh ■ Featuretools by Alteryx ■ autofeat ■ Anovos by Mobilewalla ■ Feature-engine
Feature dataset versioning	■ Data Version Control (DVC) by Iterative
Semantic layer*	■ AtScale
^Be aware that this vendor was acquired in January 2023 by DataStax. See: DataStax Acquires Machine Learning Company Kaskada to Unlock Real-Time AI. *For more information on semantic layers, consult Demystifying Semantic Layers for Self-Service Analytics.	

Source: Gartner (July 2023)

Reducing Impact on Data Management

Off-the-shelf comprehensive feature stores, MLOps or DSML platforms can typically be implemented without the need to involve the organization's data engineers, DBAs or other data management professionals. In these cases, the vendors are more involved in assisting data science teams to provision the appropriate resources and integrating them into an existing architecture. However, where open-source or custom solutions are deployed, data management technical professionals (such as data engineers, data architects and DBAs) are required to establish and maintain the logical feature store. Figure 7 demonstrates this relationship by illustrating how responsibilities are divided and shared by roles across the ML development pipeline. Data scientists should work closely with data engineers, ML engineers and other relevant roles to define, deploy and manage the appropriate logical feature store solution for the organization.

Figure 7: Typical Roles and Responsibilities Across the ML Pipeline



In addition to their role in building certain logical feature store implementations, data management technical professionals will be concerned with the impact they will have on data management infrastructure. For example, using existing DB technologies for the materialization of features will have associated storage costs. Serving of features from these DBs and leveraging in-DB analytics also has resulting compute costs on the DB side, and further compute costs will be associated with running external data pipelines, if applicable for the particular implementation. Data management technical professionals and the owner of the logical feature store solution should provide guidance to users to minimize incurring unnecessary costs. This guidance should equip people with an understanding of the impact of certain decisions, allowing them to base their usage on their specific requirements, as outlined in Table 4.

Table 4: The Impact of Logical Feature Stores on Data Management Infrastructure

(Enlarged table in Appendix)

Requirement	Sample Technologies	Ways to Optimize Costs
Scalable Storage of Historical Feature Sets	**Data warehouses:** Snowflake, BigQuery, Apache Hive, Parquet **Data lakes:** Amazon S3, Delta Lake	**Backfill period:** Users should avoid excessive backfilling of features. For example, if data is only required for a three-month period, users should not materialize a year's worth of data for that feature. **Number of features:** Users can further optimize costs by persisting only features for known ML use cases, rather than materializing many features based on their potential future utility. Definitions and transformation code can still be added to the feature registry without materializing the features, which allows these features to be easily generated at a later date, if needed.
Low-Latency Feature Serving	**Key-value stores:** Amazon DynamoDB, Redis **Distributed storage:** Apache Cassandra (NoSQL), CockroachDB (SQL)	**Number of features:** Users should avoid making a copy of all features from the offline feature store (DB for scalable storage) available in the online feature store (DB for low-latency serving). Instead, they should only persist features to the online feature store that are required for known real-time ML scoring use cases.
Compute to Transform Raw Data into Features	**In-DB analytics:** Snowflake, BigQuery ETL engines: Databricks, Amazon EMR **Streaming/on-demand engines:** Apache Flink, Apache Spark/PySpark SQL	**Feature transformation cadence:** Features updated more often will incur a greater cost than those updated infrequently. Therefore, users should think carefully about their requirements for feature freshness. For example, does this feature really need to be updated every 30 seconds or would every 2 minutes work just as well? **Streaming and on-demand pipelines:** These incur additional cost, so users should justify their need for such pipelines based on the requirements of their ML use case, only creating them when necessary.

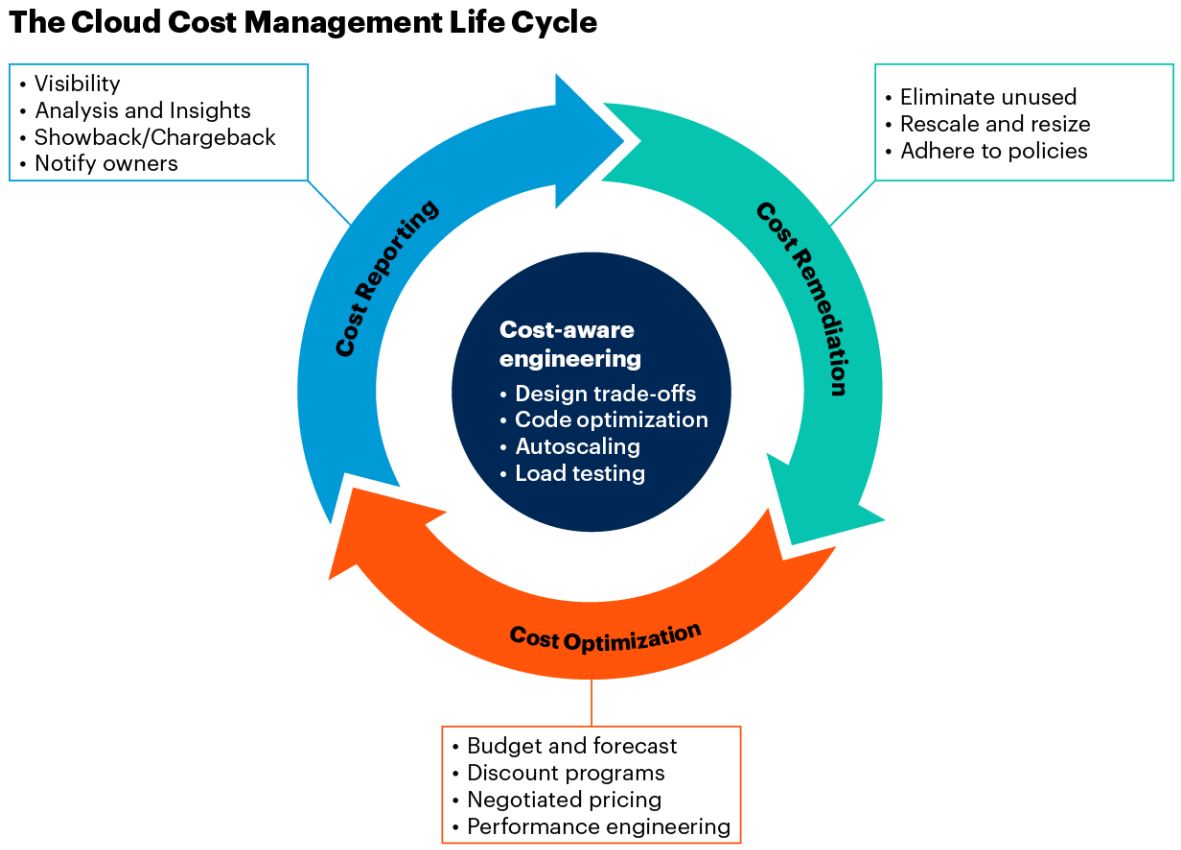
Source: Gartner (July 2023)

Certain feature store vendors encourage leveraging the organization's existing DB technology when it comes to the offline feature store, but use their own managed DB for the online feature store. This gives them more control over feature serving latency and performance, allowing them to meet their SLAs. Tecton, for example, manages instances of DynamoDB for its customers by default. However, it also allows customers to use and manage their own Redis instance if they prefer.

Regardless of whether the organization is leveraging an online store managed by a vendor or hosting and managing it themselves, it should be aware that there can be unexpectedly high costs associated with use of certain DBs, such as key value stores. [13] This is because key value stores are typically billed by usage, and there are costs associated with both reads and writes to the store. Lacking a full understanding of how key value store APIs work can lead to inefficient use of resources. In addition, increasing the number of models in production and accommodating a growing volume of pulls for on-demand ML scoring will have a direct impact on key value store costs. One way to manage costs is to switch from pay-per-use billing to a provisioned instance. The DB will have fixed storage and compute, which means that it has to be manually scaled. However, this allows data management professionals to have more control and predict monthly costs.

Data management technical professionals should incorporate appropriate tagging to the use of resources to better track who is utilizing what and adjust their strategies accordingly. Resource tagging is done automatically by some feature store vendors. Figure 8 illustrates common practices data and infrastructure teams employ to manage costs in cloud environments. A more in-depth analysis of data management, engineering and infrastructure cost optimization is beyond the scope of this document. However, technical professionals who want to learn more about this topic can get started with Gartner's Essential Skills for Cloud-Native Application Architects and Client Question Video: How Do I Make Engineers Responsible for Their Cloud Spending?.

Figure 8: The Cloud Cost Management Life Cycle



Source: Gartner
755398_C

Gartner

Evidence

[1] The State of AI and Machine Learning Report, Appen. Direct link to report: State of AI 2022 and Machine Learning Report: Managing Data for the AI Lifecycle

[2] Data + AI Summit Keynote Wednesday, Databricks, 30 June 2023 (YouTube video).

[3] P-21023 2021 Gartner AI in Organizations Survey. 2021 Gartner AI in Organizations Survey: This survey was conducted to understand the keys to successful AI implementations and the barriers to the operationalization of AI. The research was conducted online from October through December 2021 among 699 respondents from organizations in the U.S., Germany and the U.K. Organizations were required to have developed AI or intended to deploy AI within the next three years. Respondents were required to be part of the organization's corporate leadership or report into corporate leadership roles, and have a high level of involvement with at least one AI initiative. The survey was developed collaboratively by a team of Gartner analysts and Gartner's Research Data, Analytics and Tools team. Disclaimer: Results of this survey do not represent global findings or the market as a whole, but reflect the sentiments of the respondents and companies surveyed.

[4] Feature Store for ML, Feature Store. A collection of blogs, webinars, case studies and other resources on feature stores.

[5] Serving 100,000 Feature Vectors per Second With Tecton and DynamoDB, Tecton.

[6] Atlassian Dramatically Improves Machine Learning Model Accuracy and Deployment Times, Tecton.

[7] Tide Safely Processes Transactions Using Real-Time Fraud Detection, Tecton.

[8] Amazon SageMaker and Tecton: How to Choose the Right Feature Store on AWS, Tecton.

[9] NetApp Case Study: Real-Time Predictive Maintenance and Advanced Analytics, Iguazio.

[10] Payoneer Using Real-Time AI for Fraud Prevention, Iguazio.

[11] Finance and Banking: Semi-Supervised Deep Learning to Identify Fraudulent Transactions, Logical Clocks.

[12] Healthcare and Pharmaceuticals: Secure and Low Cost Platform to Manage Large and Sensitive Data, Logical Clocks.

[13] DynamoDB Feature Stores, At What Cost?, Feature Store Comparison.

Note 1: Point-in-Time Correctness

Logical feature stores should use an entity-based data model where each feature value is associated with an entity (e.g., a user) and a time stamp (typically the date and time of commit to the feature store post-transformation). Including a time stamp on each feature row provides point-in-time-correct data. This allows datasets pulled from the logical feature store for ML model development to be restricted to certain time periods, and it enables users to later recreate the datasets used for training and testing. Logical feature stores help to prevent data leakage, where timepoints that the model is attempting to predict are inadvertently included in the training data used to create the model. If this occurs, the model will have an overly optimistic estimate of accuracy that the model cannot achieve when deployed into production. Time-stamping features means that the most recent data for each entity can be identified, and models can be updated based on an up-to-date world view. Identifying the latest model scorings is also an advantage, as certain ML use cases require that these are retrieved with very low latency for business or customer insights, decision making or predictions.

Certain feature store vendors also support “time travel,” which is the ability to backfill historical data for new features added to the store or to query data as it was at a specific point in time/time-interval. This is possible when the feature store pulls data from source systems that retain commit date and time metadata as part of their lineage, like Apache Hudi. With time travel, the source system commit time stamp is persisted to the feature store along with the transformed feature data for the date range specified. Time travel allows data scientists to:

- Create datasets for training that represent a particular historical point in time with data not already in the feature store
- Modify an existing dataset by adding a new feature and backfilling its values to cover the same time frame as the rest of the data
- Recreate datasets used for model development if these are not kept in permanent storage

The latter case may occur where the deliberate decision is made to save on storage costs within the repository. It may not be feasible to keep years' worth of historical data in the feature store long term, for example. Rather, the feature store can reproduce datasets used for training and testing on-demand, using versioned transformation code and time travel to specify the relevant time frame. Time travel is useful when making changes to a dataset, proving the ability to roll back a bad commit of data, comparing statistics for different versions of features and measuring how features change over time. Therefore, time stamps and time travel capabilities are necessary to address specific questions with ML for model audit and for model retraining.

Note 2: Benchmarking Studies for Online Feature Stores

The following is a sample list of known benchmarking studies for feature store implementations:

- AI/ML Needs a Key-Value Store, and Redis Is Not Up to It, Hopsworks.
- Sysbench Evaluation of RonDB, RonDB.
- Performance Test for the Python-Based Feast Feature Server: Comparison Between DataStax Astra DB (Based on Apache Cassandra), Google Datastore & Amazon DynamoDB, Feast.
- Serving 100,000 feature vectors per second with Tecton and DynamoDB, Tecton.

Independent commentaries or analyses:

- Building a Gigascale ML Feature Store with Redis, Binary Serialization, String Hashing, and Compression, DoorDash.
- DynamoDB Feature Stores, At What Cost?, Feature Store Comparison.
- A Comparison of Data Stores for the Online Feature Store Component: A Comparison Between NDB and Aerospike, DiVA. Thesis by Alexander Volminger (2021) for a degree in computer science and engineering from the KTH Royal Institute of Technology, Sweden. NDB = network database cluster, sometimes called MySQL Cluster.

Document Revision History

Feature Stores for Machine Learning (Part 1): The Promise of Feature Stores - 29 April 2021

Recommended by the Author

Some documents may not be available as part of your current Gartner subscription.

Incorporate, Test, Deploy and Maintain Machine Learning Models in Production Applications

Launch an Effective Machine Learning Monitoring System

A Guidance Framework for Deploying Data and Analytics in the Cloud

Solution Comparison for Cloud Data Science and Machine Learning Platforms

Design IoT Stream Analytics From Edge to Platform

Essential Patterns for Event-Driven and Streaming Architectures

Use Gartner's MLOps Framework to Operationalize Machine Learning Projects

Market Guide for DSML Engineering Platforms

Market Guide for Multipersona Data Science and Machine Learning Platforms

Demystifying XOps: DataOps, MLOps, ModelOps, AIOps and Platform Ops for AI

Create a Data Strategy to Ensure Success in Machine Learning Initiatives

Building a Framework for Managing Effective Machine Learning Workloads

A Guidance Framework for Operationalizing Machine Learning

Implementing an Enterprise Open-Source Machine Learning Stack

Getting Started With Machine Learning Monitoring in Production

Table 1: Range of Capabilities Found Across Feature Store Vendor Solutions

Vendor Solutions Offering	Description	Benefits
Storage	The feature store product contains only a dedicated repository of curated features from which training and test datasets can be pulled for ML model development.	■ Decreases time spent on individualized feature engineering efforts. ■ Standardizes feature definitions. ■ Promotes collaboration and the sharing of features.
Storage + Online Serving	In addition to storing features offline, some feature store products serve features to ML models in production for online scoring use cases. Some provide service-level agreements (SLAs) for online feature serving to meet clients' low-latency requirements. Care must be taken when selecting a product because some vendors have better SLAs than others.	■ Provides a consistent view of features in development and production, preventing training-serving skew. ■ Supports on-demand ML scoring use cases.
Feature Transformations + Storage + Online Serving	Few solutions have the capacity to orchestrate the transformation of raw data into features and ingest them directly into the online and offline feature stores. Those that do can manage various update cadences and both batch and streaming data. However, there are still limitations in their ability to do on-demand feature transformations.	■ With feature transformations contained within the feature store solution, rather than outsourced to other ETL pipelines, versioned transformation code and related metadata can be captured to record the lineage of features.

		■ Updating features according to their appropriate cadence ensures an up-to-date world view.
Feature Transformations + Storage + Online Serving + Monitoring	Certain feature store implementations calculate and store descriptive statistics alongside the features they contain. They use these benchmark metrics to detect things like poor data quality and data drift, to monitor features serving models in production, and to flag when these deviate significantly from baseline statistics.	■ Automated detection and the ability to diagnose issues facilitates model maintenance. ML model performance degrades over time. Much of this can be attributed to issues with data, such as changing real-world context, adversarial attacks and the ingestion of poor-quality data not caught by preceding data pipelines.

Source: Gartner (July 2023)

Table 2: Sample List of Feature Store Vendors

Stand-Alone Feature Store Vendors	DSML/MLOps Vendors With Built-in Feature Stores
■ Hopsworks by Logical Clocks ■ Tecton ■ Feast by Tecton ■ FeatureByte ■ Featureform	■ 4Paradigm: OpenMLDB* ■ Abacus.AI ■ Amazon SageMaker ■ C3 AI ■ Databricks ■ Dataiku ■ Google Vertex AI ■ H2O.ai ■ Iguazio^ ■ Qwak

*Product not listed on vendor's main website
^Be aware that this vendor was acquired in January 2023 by McKinsey. See: McKinsey Acquires Iguazio, A Leader in AI and Machine Learning Technology.

Source: Gartner (July 2023)

Table 3: Alternative Feature Engineering Ecosystem Products

Product Type	Sample Vendors
Feature engineering	■ Enrich Intelligence Platform by Scribble Data ■ Kaskada^ ■ fal.ai (also known as “Features and Labels”) ■ dotData ■ Mobilewalla
Feature engineering packages and libraries	■ tsfresh ■ Featuretools by Alteryx ■ autofeat ■ Anovos by Mobilewalla ■ Feature-engine
Feature dataset versioning	■ Data Version Control (DVC) by Iterative
Semantic layer*	■ AtScale

^Be aware that this vendor was acquired in January 2023 by DataStax. See: DataStax Acquires Machine Learning Company Kaskada to Unlock Real-Time AI.
*For more information on semantic layers, consult Demystifying Semantic Layers for Self-Service Analytics.

Source: Gartner (July 2023)

Table 4: The Impact of Logical Feature Stores on Data Management Infrastructure

Requirement	Sample Technologies	Ways to Optimize Costs
Scalable Storage of Historical Feature Sets	**Data warehouses:** Snowflake, BigQuery, Apache Hive, Parquet **Data lakes:** Amazon S3, Delta Lake	**Backfill period:** Users should avoid excessive backfilling of features. For example, if data is only required for a three-month period, users should not materialize a year's worth of data for that feature. **Number of features:** Users can further optimize costs by persisting only features for known ML use cases, rather than materializing many features based on their potential future utility. Definitions and transformation code can still be added to the feature registry without materializing the features, which allows these features to be easily generated at a later date, if needed.
Low-Latency Feature Serving	**Key-value stores:** Amazon DynamoDB, Redis **Distributed storage:** Apache Cassandra (NoSQL), CockroachDB (SQL)	**Number of features:** Users should avoid making a copy of all features from the offline feature store (DB for scalable storage) available in the online feature store (DB for low-latency serving). Instead, they should only persist features to the online feature store that are required for known real-time ML scoring use cases.
Compute to Transform Raw Data into Features	**In-DB analytics:** Snowflake, BigQuery ETL engines: Databricks, Amazon EMR	**Feature transformation cadence:** Features updated more often will incur a greater cost than those updated infrequently. Therefore, users

	Streaming/on-demand engines: Apache Flink, Apache Spark/PySpark SQL	should think carefully about their requirements for feature freshness. For example, does this feature really need to be updated every 30 seconds or would every 2 minutes work just as well? **Streaming and on-demand pipelines:** These incur additional cost, so users should justify their need for such pipelines based on the requirements of their ML use case, only creating them when necessary.

Source: Gartner (July 2023)