tecton

Feature Stores for Real-Time Machine Learning. Build or Buy?

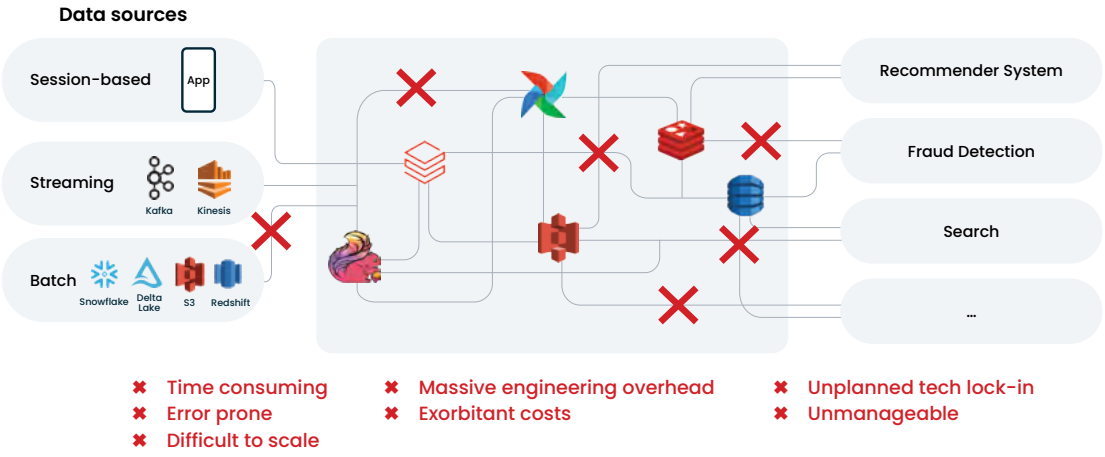
TABLE OF CONTENTS

Introduction

Building a feature store is challenging and exciting work, and until recently, running machine learning in production has merely been a pipe dream for most companies big and small. And with consumer habits changing and leaning toward wanting everything from shopping recommendations to food delivery estimates in real time, we've seen a huge rise in machine-learning platform teams that are responsible for building or buying the tools needed to enable engineers to efficiently build production machine learning (ML) systems.

However, most teams tasked with building an ML platform aren't aware of all the challenges they will be facing, including engineering overhead, difficulty scaling, and—the biggest one of all—managing the data pipelines needed to get ML models into production. (You can read more about why real-time data pipelines for machine learning are so challenging here.)

Building a solution yourself is hard. Maintaining it is harder.



If your team is considering building a feature store and isn't sure where to start, then this whitepaper is for you. We'll share some of the key challenges and considerations you'll need to make while designing your solution. We'll also share when it may make more sense to buy a solution and the benefits of doing so.

Why Do You Need a Feature Store?

But first things first: What's the reason you're looking to build a feature store? In our experience, most teams come across one of the following challenges that led them down this path:

1. You need a way to serve features to your models in real time for inference at high scale and low latency.
2. You want to automate batch, streaming, and real-time data pipelines so that your team can focus on building models quickly and reliably instead of wasting time building bespoke pipelines. Not only that—building data pipelines for real-time ML is notoriously difficult.
3. You don't want to maintain separate data pipelines for training and serving, and want to reduce training / serving skew for served models.
4. You want your data scientists to be able to easily share and re-use features across teams and models.

Note also that your team's reason for wanting a feature store will largely influence the system you design. In our experience, reason No. 1 is the key challenge that drives most teams to build a feature store right now. But is a feature store enough for your needs or do you really need a feature platform?

In this section, we'll take a quick look at the difference between a feature store and a feature platform. If you're already familiar with the difference between the two, skip to the next section "Building a Feature Store or Feature Platform: Getting Started."

What is a feature store?

A feature store serves two purposes: To store and serve features consistently across offline training and online inference environments.

When features are not stored consistently across both environments, the features a model is trained on might have subtle differences from the features it uses for online inference. This phenomenon is called train / serve skew, and it can derail a model's performance in silent and catastrophic ways that are extremely difficult to debug. A feature store solves this problem by having consistent data in both environments.

For offline training, feature stores need to contain months' or years' worth of data. This is stored in data warehouses or data lakes like Databricks, BigQuery, Snowflake, or Redshift. These data sources are optimized for low-cost batch retrieval.

For online inference, applications need to have ultra-fast access to the freshest data possible. To enable low-latency lookups, this data is kept in an online store like DynamoDB, Redis, or Cassandra. Only the latest feature values for each entity are kept in the online store.

To support simple feature management, feature stores provide data abstractions that make it easy to deploy and serve features across environments.

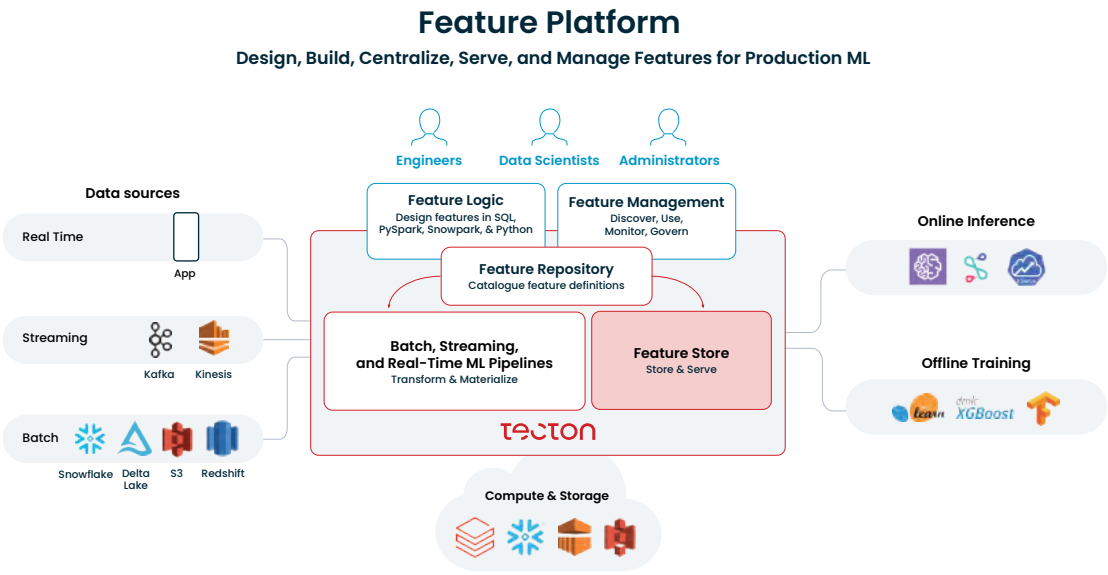
To support simple feature management, feature stores provide data abstractions that make it easy to deploy and serve features across environments. For example, they make it easy to store data consistently across offline and online storage, and present simple APIs to serve the data across both the development environment (for training on historical values) and the production environment (for inference with fresh feature values).

Feature stores bring economies of scale to ML organizations by enabling collaboration. When a feature is registered in a feature store, it becomes available for immediate reuse by other models across the organization. This reduces duplication of data engineering efforts and allows new ML projects to bootstrap with a library of curated production-ready features.

What is a feature platform?

As organizations matured in their use of operational ML, many often discovered a multitude of use cases that required real-time features built using streaming or real-time data sources—often with data freshness requirements in the order of milliseconds. The feature platform emerged from the need for a solution more robust than a feature store to support real-time ML.

To put it simply, **a feature platform is a system that manages the complete cycle of ML features**. A feature platform includes a feature store, but extends it with additional capabilities to automate pipelines, monitor features, and bring DevOps engineering best practices to the process of developing features.



A modern feature platform flexibly connects to an organization's existing data infrastructure and:

- Orchestrates data pipelines that transform raw data into feature values
- Stores and manages the feature data itself
- Serves feature data consistently for training and inference purposes
- Monitors operational service levels and data quality

A feature platform aims to solve the full set of data management problems encountered when building and operationalizing real-time machine learning applications by making it easy to:

- Build new features without extensive engineering support
- Automate feature computation, backfills, and logging
- Discover, share, and re-use feature pipelines across teams
- Develop new features collaboratively using DevOps engineering best practices
- Track feature versions, lineage, and metadata
- Achieve consistency between training and serving data
- Monitor the health of feature pipelines in production

Note: Feature platforms do not replace existing infrastructure. Instead, they should plug into your existing storage and compute infrastructure by:

- Connecting to batch data sources such as data lakes and data warehouses and streaming sources like Kafka
- Using existing compute infrastructure, such as a data warehouse or Spark, and existing storage infrastructure, such as S3, DynamoDB, or Redis.

A feature platform is a system that manages the complete cycle of ML features

Building a Feature Store or Feature Platform: Getting Started

There are a lot of key requirements and other considerations to take into account when building a feature store. It will take a lot of data engineering expertise and engineering hours, but some teams prefer to build their own solution to avoid vendor lock-in and so they can have more control of their roadmap.

To help you get started, here are some questions you should ask to gather the information you need before you start building a feature store.

Question	Requirements
What **batch data sources** do you need to read raw data from?	Snowflake, Redshift, S3, GCS, HDFS, etc.
What **real-time data sources** do you need to read raw data from?	Kinesis, Kafka, Pub/Sub, RDS, etc.
What **platforms** does your data team use to write feature transformation logic	Snowflake, Redshift, Spark (EMR / Databricks), local Python, DBT
How **fresh** do your features need to be for the use cases you are supporting?	<100 ms, <1 second, <1 minute, <1 hour, <1 day, etc.
What are the **serving latency constraints** of the use cases you are supporting?	P(99) / Media -<10 ms, <100 ms, <1 s, batch, etc.
What is the **expected serving load** for the use cases you are supporting?	XXX QPS
How many **users** do you plan to support?	<10, <100, 100+
Will your feature store handle **sensitive or regulated** data assets?	Yes / No
Will your team need to provide an **on-call engineer** to support the feature store?	Yes / No
What is your **uptime requirement**?	99%, 99.9%, 99.99%

Additionally, before you start building, you should also take into consideration:

× Hidden challenges

Most teams start building a feature store without realizing the full scope what they need to build. Some of the most common mandatory components that are overlooked include monitoring, serving infrastructure, orchestration, and compliance.

× Future proofing vs. premature optimization

There are going to be a ton of places where you will need to decide if you want to architect a flexible (but complicated) solution or build something simpler that just works for now. Our advice is to follow the 80/20 rule—make sure what you're building will solve for one or two key use cases at the start, then expand to other use cases that can easily fit in with a little bit of work.

× Where to draw the line

After reading this, you may be tempted to build a lot of components—some teams have even developed their own database technologies to support the online store! However, we recommend managed online stores like DynamoDB, Redis, and pretty much any other cloud-native service that makes sense for your needs.

× How much to invest in training and onboarding

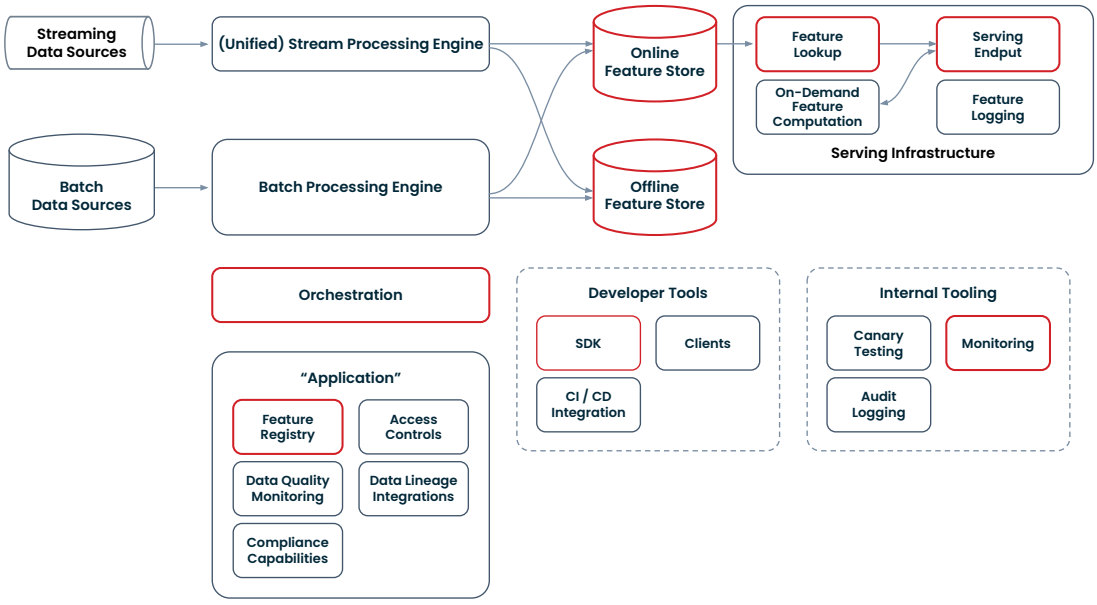
Every great enterprise tool comes with great documentation and training. Providing that same quality of documentation and training for the tools you build will pay dividends later.

× How to plan for maintenance

We've worked with dozens of teams that have built feature stores, and all of them have had at least 3 full-time engineers dedicated to maintaining the project. In other words, any software this complex will require a significant amount of ongoing engineering support.

The figure below is a template for the components of a feature platform.
The red items are essential to build a minimum viable product.

The Components of a Feature Platform



And all this is just the beginning! Check out Appendix A for a deep dive into the key dependencies and considerations for each mandatory component when building a feature store.

Should I Build or Buy a Feature Store / Platform?

You may be asking yourself whether it makes sense to build a feature store after reading everything that goes into it—it's not an easy project by any means.

There are a number of specific challenges that should be considered when building an in-house solution, including:

- Needing to backtest to ensure offline and online data consistency
- Difficulty adding new features and re-using existing ones
- Integrating stream and real-time aggregations
- Ensuring the solution meets all government and security requirements
- Providing the required engineering expertise and time
- Risk of high employee turnover, leading to IP loss
- Maintaining and scaling the solution to fit changing business needs

In contrast, buying a feature store or a feature platform will enable ML teams to manage their workflows, at scale, while minimizing costs and engineering overhead required to build and maintain bespoke and error-prone pipelines. Buying a solution enables engineers to focus on building quality features to help drive competitive advantage for the business. They can calso deploy ML models to production in days rather than months, which directly impacts the bottom line.

There are many feature store solutions in the market, and it can be difficult to differentiate between them. Ideally, we recommend that whatever solution your team chooses provides:

- One central source of truth for ML features—and enables easy re-use of features
- Native support for batch, streaming, and real-time data
- Faster development cycles for new streaming and real-time features
- Reliability, resilience, and scale
- Consistency in offline and online data
- Fully managed service, with SLAs and enterprise support

Build vs. Buy: One Company's Experience

Tide, a UK business banking provider for small and medium enterprises, thrives on making data-driven decisions to help its customers save time and money. To address real-time use cases and to scale their operational ML efforts across the board, Tide began developing an internal feature store.

However, during the process, they not only encountered hurdles incorporating both batch and streaming data into their feature store, but they also quickly realized that **building their basic feature store would take another 6-9 months and require 3 additional full-time employees just to maintain it.** Additionally, although their in-house solution meant no vendor lock-in and gave them complete control of the roadmap, it also presented the following challenges:

× Adding features was hard

Adding features from new topics was difficult and required significant engineering overhead. There was no support for stream aggregations or adding features from Snowflake.

× Changes to production were unpredictable

Expected changes from training did not hold up in production environments. Approval rates were lower than anticipated during training, and the team had to do feature engineering twice for backtesting.

× Slow iteration times

Adding new features required extensive engineering support, and over 50% of engineering time was spent on creating feature pipelines.

After considering the financial and time costs of building their own solution, Tide decided to evaluate buy options and landed on Tecton. With support from Tecton's engineering team, **Tide was able to roll out Tecton's fully integrated feature platform into production in just 6 weeks and saw the following benefits:**

✓ Easy re-use of high-quality features

Tide's data science and engineering teams heavily re-use features to power different credit risk and fraud detection models. Tide is also now able to fine-tune the features for any use case in a fraction of the time it would take them to build new features from scratch.

✓ **Ease of backtesting**

Tide can now estimate the holistic impact of changes on production environments before deploying those changes into production. Teams can regularly run experiments to measure how changes to rules or ML impact overall system results, approval rates, or historical risk realizations—and the pre-deployment estimations closely match production performance.

✓ **Improved deployment time**

Tide uses Tecton to re-use pre-built feature types, greatly reducing workloads previously involved when adding features. Tide reduced its overall delivery time from model conception to deployment in production by 50% while greatly improving model accuracy.

Conclusion

The key takeaway is that whether your teams choose to build or buy will heavily depend on what your organization's needs are. For instance, how quickly do you need to deploy the solution? Do you have the engineering support and expertise required to build and maintain an in-house feature store? If you choose to build, do you have the budget needed for a project that may run months before a working model can be deployed? Can it handle streaming data? Do you want a feature store or a feature platform?

Whether your team chooses to build or buy a feature store or feature platform, a vital step is to spend time collecting the requirements you want in the solution.

At Tecton, we've helped many teams on their journey to real-time ML. If you'd like a demo or have questions about whether a feature store or feature platform is better for your organization, contact us today. Or if you would like to try out Tecton for yourself, sign up for a free trial.

Appendix A

How to Build a Feature Store

Gathering Requirements

Here are the key requirements you should collect:

Question	Requirements
What **batch data sources** do you need to read raw data from?	Snowflake, Redshift, S3, GCS, HDFS, etc.
What **real-time data sources** do you need to read raw data from?	Kinesis, Kafka, Pub/Sub, RDS, etc.
What **platforms** does your data team use to write feature transformation logic	Snowflake, Redshift, Spark (EMR / Databricks), local Python, DBT
How **fresh** do your features need to be for the use cases you are supporting?	<100 ms, <1 second, <1 minute, <1 hour, <1 day, etc.
What are the **serving latency constraints** of the use cases you are supporting?	P(99) / Media -<10 ms, <100 ms, <1 s, batch, etc.
What is the **expected serving load** for the use cases you are supporting?	XXX QPS
How many **users** do you plan to support?	<10, <100, 100+
Will your feature store handle **sensitive or regulated** data assets?	Yes / No
Will your team need to provide an **on-call engineer** to support the feature store?	Yes / No
What is your **uptime requirement**?	99%, 99.9%, 99.99%

Additionally, there are a lot of optional enhancements you will likely get asked about that you'll need to decide to build or hold off on, such as:

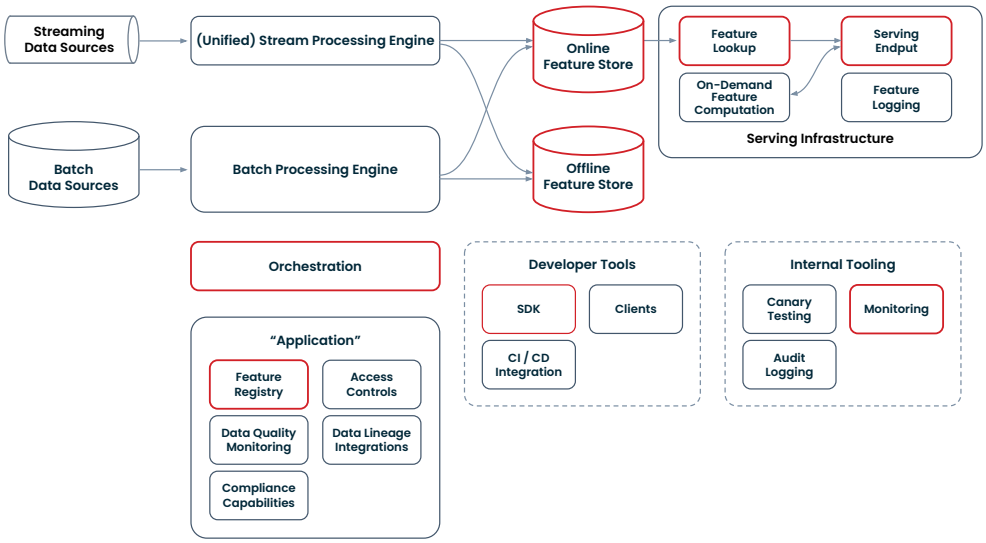
- Feature versioning
- Data lineage (including integrations with data lineage systems)
- Data Quality Monitoring
- Automatic time travel capabilities
- CI/CD capabilities for managing the deployment of features
- Cost visibility into ML data pipelines

And finally there is a whole suite of tooling that you may need to support and maintain for your feature store, including operational monitoring and canary capabilities.

And with all projects, it is vital to be forward thinking—the most common story we hear from teams who have built internal feature stores is that their solution worked great for use cases No. 1 and No. 2, but then use case No. 3 came along and had different requirements (e.g., a more demanding p(99) lookup latency).

Understanding the Components

Here's a template for the components of a feature store. The red items are essential for most MVPs—without a solution for each of these components, driving adoption will likely be pretty difficult.



We'll cover the key considerations for each of the most important components in the next section.

The Build

We would need to write a whole book to do a full technical deep dive on each of the components described above, so instead we'll focus on the key dependencies and key considerations when designing the components you need.

Overall considerations

Here are a few overall best practices to consider throughout the process of building a feature store:

- Even with your own build, you don't need to manage all of the infrastructure. Look for opportunities to use managed services for critical infrastructure. Tools like DynamoDB can be a lifesaver compared to managing your own infrastructure.
- When you don't use a managed service, try to find a well-established open-source project—there's no need to reinvent the wheel when there are communities supporting some of the key building blocks.
- There is a frequent tension in this space between future proofing and premature optimization. In the data space, tools evolve quickly. If you don't future proof you could end up with an obsolete feature store in less than 12 months—but you also can't engineer for every possibility

Feature registry

Key dependencies

- Data processing engine, offline feature store, batch data sources, orchestration, CI/CD integration, access controls

Key Considerations

- We recommend building on top of Feast as an underlying feature registry. Feast is a widely used open source feature store, with a mature registry component that should serve as a solid foundation for your product.

- The feature registry is in many ways the core of a feature store. To be confident in how features are created and registered, you need to have a full understanding of the tools your team will use to store and process data.
- The feature registry one of the key places to consider future proofing—we've talked to many teams that have had to completely reinvent their registry because of a change in their data platform (like adopting Snowflake, for example).
- If you plan to do real-time ML, your features will directly impact your production application, so you will want a way to integrate features into the CI/CD pipelines used elsewhere in your team to roll out production software. This likely means features should be defined as declarative code assets.
- There is an inherent tension between access controls and feature discoverability and re-use. You'll either want to design a solution custom engineered to how your team wants to strike that balance, or you'll need to build a system flexible enough to handle multiple possible scenarios.

Data processing engine (batch and/or streaming)

Key dependencies

- Feature freshness requirements, batch data sources, streaming data sources, offline feature store, online feature store, size of data

Key Considerations

- Once again, you'll need to decide how much to future proof your compute engine. For instance, if you use Spark today but transition to Snowflake next year, will you need to rebuild your processing framework? At the very least, you should build a plan for how your framework will generalize to new technologies.
- Users will have different requests for how they want to define feature transformation logic; data scientists will want to write Pandas code, analysts will expect to write SQL, ML engineers might expect to use Spark in Scala. You'll need to decide between flexibility (which will take a huge engineering effort to implement) and prescriptiveness (which will reduce adoption of your feature store).
- We recommend erring on the side of prescriptiveness—we've seen over and over again that people will adapt to the tools that work as long as you provide a reliable path to

production. Going the prescriptiveness route means you'll hear complaints, but that's better than missing deadlines.

- If you need to process streaming data, you'll need to figure out how to do backfills. At Tecton we've implemented unified streaming / batch pipelines to ensure the same logic is used to process streaming data and historical data. You'll likely need to do the same, otherwise you'll open yourself up training / serving skew issues. We chose to use Spark for our unified processing, but Flink and ksqlDB are other popular choices used by many teams.
- If you have large enough scale, you may be able to cut out backfilling altogether by leveraging feature logging. Many of the largest companies in the world (Google and Twitter, for example) rely almost exclusively on logging. If you're lucky enough to operate at that scale, then backfilling is probably not worth your time.
- In general when processing streaming data, processing can fail in a lot of places or costs can explode in common but counter-intuitive scenarios. This is normal, but it's good to be aware of some common issues. One of the more common examples is aggregations—and one possible solution to that is a tiling approach.
- Your data engine will be one of the primary infrastructure costs in your solution. To help reduce costs, you may want to build support for Spot Instances, which will save you up to 50% in processing costs in many cases.

Orchestration

Key dependencies

- Data processing engine, feature registry

Key considerations

- There are several orchestration solutions available (like Airflow, Dagster), but most analysts and data scientists aren't comfortable with these tools. You may need to figure out an abstraction layer (via the feature registry) that makes these tools easier to work with for the narrow scope of feature pipelines.

- Alternatively, you can build your own orchestrator to ensure it is as simple as possible for your users to interact with. This is obviously much more expensive and will take a good amount of engineering effort.

Offline Feature Store

Key dependencies

- Batch data sources, orchestration engine, SDK, compliance capabilities

Key considerations

- How you architect the data in your offline store will have a significant impact on the performance of your SDK for generating training data. You can even consider not having an offline store at all, and instead only store the transformation logic needed to compute features. Doing so will result in slower training data generation, but if you infrequently re-use a feature, it could be cost efficient.
- You'll likely want to match the storage location of your batch data sources — if you've settled on a central data strategy, your feature store should store data in that same central location.
- The same caveats as data sources apply here — if your company migrates to the cloud, you should make sure that your offline store code will be easily adapted to the new solution.

Online Feature Store

Key dependencies

- Latency constraints, expected serving load, expected total data volume, cost sensitivity

Key considerations

- The defacto standard for online feature stores is some sort of NoSQL database like Redis, DynamoDB, or Cassandra. These systems typically have excellent tail latencies and are optimized for very high write loads, making them an ideal choice for a high-volume application.

- Which solution you choose will depend on the constraints of your use cases. Managed services like DynamoDB are simple to use but can be extremely expensive for high QPS. Redis is extremely performant, but will require some manual management.
- Make sure you scope out more than a single use case when choosing an online store—as more demanding use cases surface, one of the most common stories we hear is having to migrate online stores.
- In particular, look out for ranking and recommendations. These use cases are often much more demanding on the online store because they need to retrieve features for thousands of candidates for each prediction, leading to requirements that are much more demanding than other use cases in your application.

Serving Infrastructure

Key dependencies

- Online feature store, latency constraints, expected serving load, expected total data volume, cost sensitivity

Key considerations

- Having clients directly consume from the online store can make it challenging to deploy upgrades because it tightly couples your feature consumers to your infrastructure. We recommend building an endpoint that exposes real-time feature data centrally for your ML applications.
- You will likely do some last-minute computation at feature-serving time, like joining together multiple features to form a feature vector or computing request-time features. These capabilities can be very challenging to implement efficiently so be sure to budget enough engineering time to build a scalable, efficient execution engine on your serving infrastructure.

Access Controls

Key dependencies

- Batch data sources, stream processing engine, feature registry, offline feature store, online feature store, feature serving infrastructure

Key considerations

- Access controls for feature stores are tricky and can get in the way of realizing your goals of being able to freely share features across your organization. Depending on your use cases, everything from raw data to feature definitions to feature data could be considered sensitive and subject to access controls.

Compliance Capabilities (e.g., GDPR)

Key dependencies

- Offline feature store, online feature store

Key considerations

- The biggest consideration here is how you plan to architect your offline feature store. If you plan to hold sensitive and regulated data in the feature store, we recommend using a format like Delta Lake or another large-scale offline storage technology that supports efficient single-row deletions. We also recommending figuring this out upfront to avoid an expensive migration later.

SDK

Key dependencies

- Offline feature store, data scientist preferences

Key considerations

- Data scientists often prefer to work in iPython notebooks—we recommend designing a simple mechanism for them to prototype features and retrieve data from a standard Python environment. Depending on where your data is stored, this can be very challenging and may require you to kick off asynchronous jobs to process large volumes of data

Monitoring

Key dependencies

- Online feature store, serving infrastructure, computation engine

Key considerations

- You'll need to outfit nearly every piece of the critical inference path with monitoring capabilities to successfully support the SLAs you need to provide. To be able to scale as quickly as your team, this monitoring needs to be deployed automatically, so we recommend designing tooling that automatically plugs into monitoring. Some common failures to catch are:
    - Job failures when computing features
    - Features becoming stale (either from stale jobs or issues with upstream data)
    - Serving infrastructure underprovisioning
    - High serving latencies

Canary Testing

Key dependencies

- Computation engine

Key considerations

- Part of the promise of a feature store is consistency—feature logic should remain steady through time. If you plan to update or even just upgrade your computation engine (which is likely to happen as you improve the framework), we recommend having a mechanism to test that the features computed are the same before and after the changes you make. This is a challenging piece to get right, so make sure you budget necessary engineering time for it.

Key Takeaways

A quick recap of the most important takeaways:

- **Future proofing vs. premature optimization.** There are going to be a lot of places where you'll need to decide if you want to architect a flexible (but complicated) solution, or build something that just works (for now). Our recommendation is to follow the 80/20 rule: Make sure what you're building will solve for one or two key use cases to start, then expand to other use cases that fit well enough.
- **Hidden challenges.** Most teams start building a feature store without realizing the full scope of what they need to build. Some of the most common mandatory components that are overlooked include monitoring, serving infrastructure, orchestration, and compliance.
- **Draw the line.** You will be tempted to build a lot of components—some teams have gone as far as developing their own database technologies to support the online store! However, when in doubt, use pre-built components—we recommend Feast, managed online stores like DynamoDB and Redis Enterprise, and pretty much every cloud native service that makes sense.