STITCH FIX

How do you evolve your data infrastructure?

Neelesh Srinivas Salian

Strata Data Conference, London
May 1, 2019

Stitch Fix

Personalized styling service serving Men, Women, and Kids

Founded in 2011, Led by CEO & Founder, Katrina Lake

Employ more than 6,000 nationwide (USA)

Algorithms + Humans

This talk

- Algorithms Team philosophy
- Generations of Infrastructure and the lessons learnt
- Story of the Evolution of our Readers/ Writers Tools
- Questions

Algorithms Team Philosophy

Culture of Data Science

1. First, you have to position data science as its own entity.
2. Next, you need to equip the data scientists with all the technical resources they need to be autonomous.
3. Finally, you need a culture that will support a steady process of learning and experimentation.

Curiosity-Driven Data Science by Eric Colson
Havard Business Review

Generations of Infrastructure

Generation 0

Key points of Generation 0

Think of data science before any platform. Ad-hoc tooling exists everywhere.

1. Data stored in a format within a form of storage
2. A Client to access the data
3. No other explicit products

Learning from Generation 0

Problems of Generation 0

1. This is a new team/company, things are not defined.
2. The business changes, you did what is needed for the users and business at the time.
3. You are slowly understanding the lack of the infrastructure and the pains it causes.

0

1

What happens between 0 and 1

1. Company changes - physically and culturally
2. Business expands and grows
3. Users increase

Generation 1

Key points of Generation 1

The team formulates decisions about what to build to bring up a platform for the users.

1. A platform is built with common resources that Data Scientists can share, although many specific capabilities are still built by Data Scientists themselves in an ad hoc fashion.
2. The common resources are presented as engineering artifacts to be learned and mapped onto the Data Scientists' work patterns.

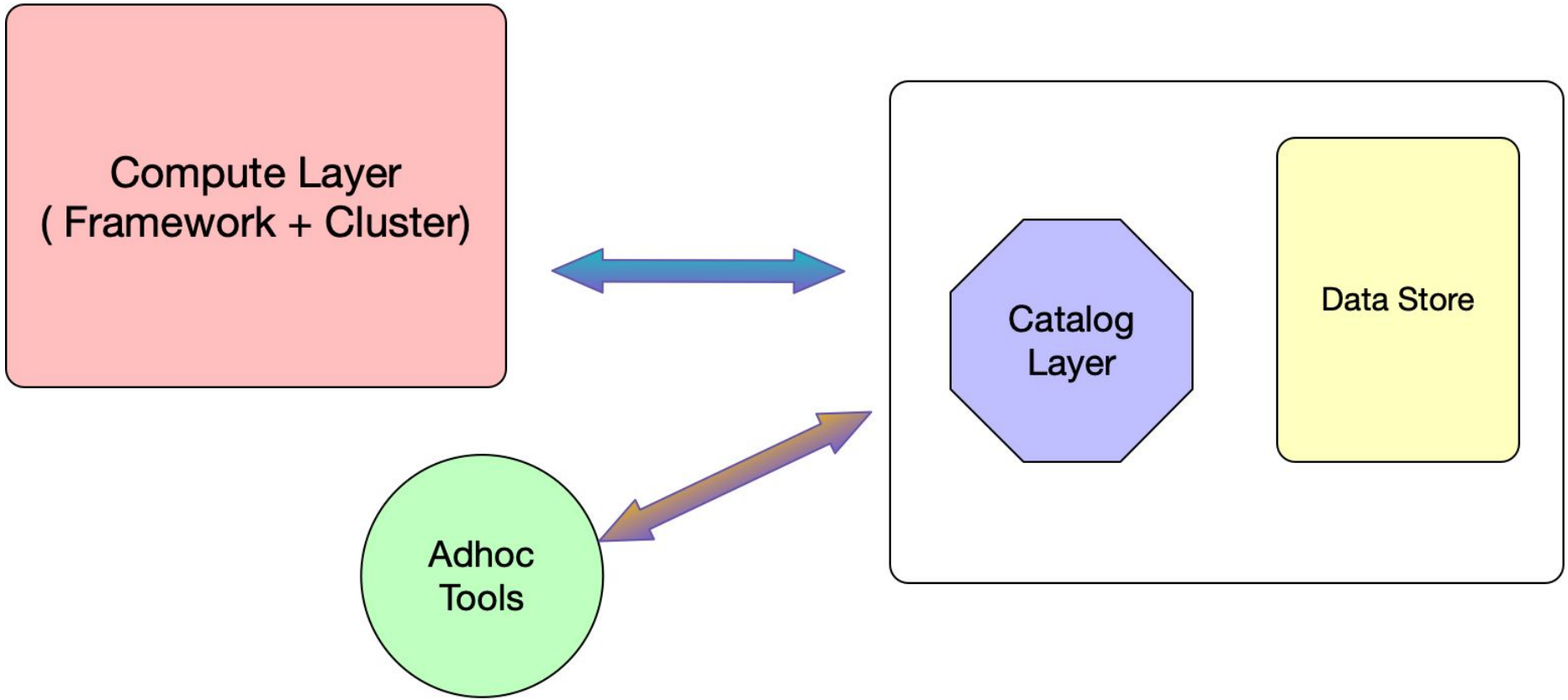


Figure 1 : Generation 1 of the Data Platform

Learning from Generation 1

Problems of Generation 1

1. Rushed into building tools
2. A reasonable time wasn’t spent in prototyping
3. The data model is better but hard to maintain.

1

2

What happens between 1 and 2

Much Longer than 0 → 1

1. Attain maturity in terms of users and use
2. The business is more stable and the use cases are defined.
3. This allows the ideas to become clearer as the problems are known. This paves the way for solutions.
4. Larger blocks of the platform can now be designed and implemented.

Generation 2

Key points of Generation 2

1. Platform reaches nearly complete coverage of shared resources for the needs of data scientists.
2. Modern tools and framework. Initial versions are iterated upon.
3. One-off ad hoc infrastructure is only rarely built
4. This is a much more stable platform with better abstractions than Generation 1.
5. The platform is enough to be self-sufficient and expand upon.

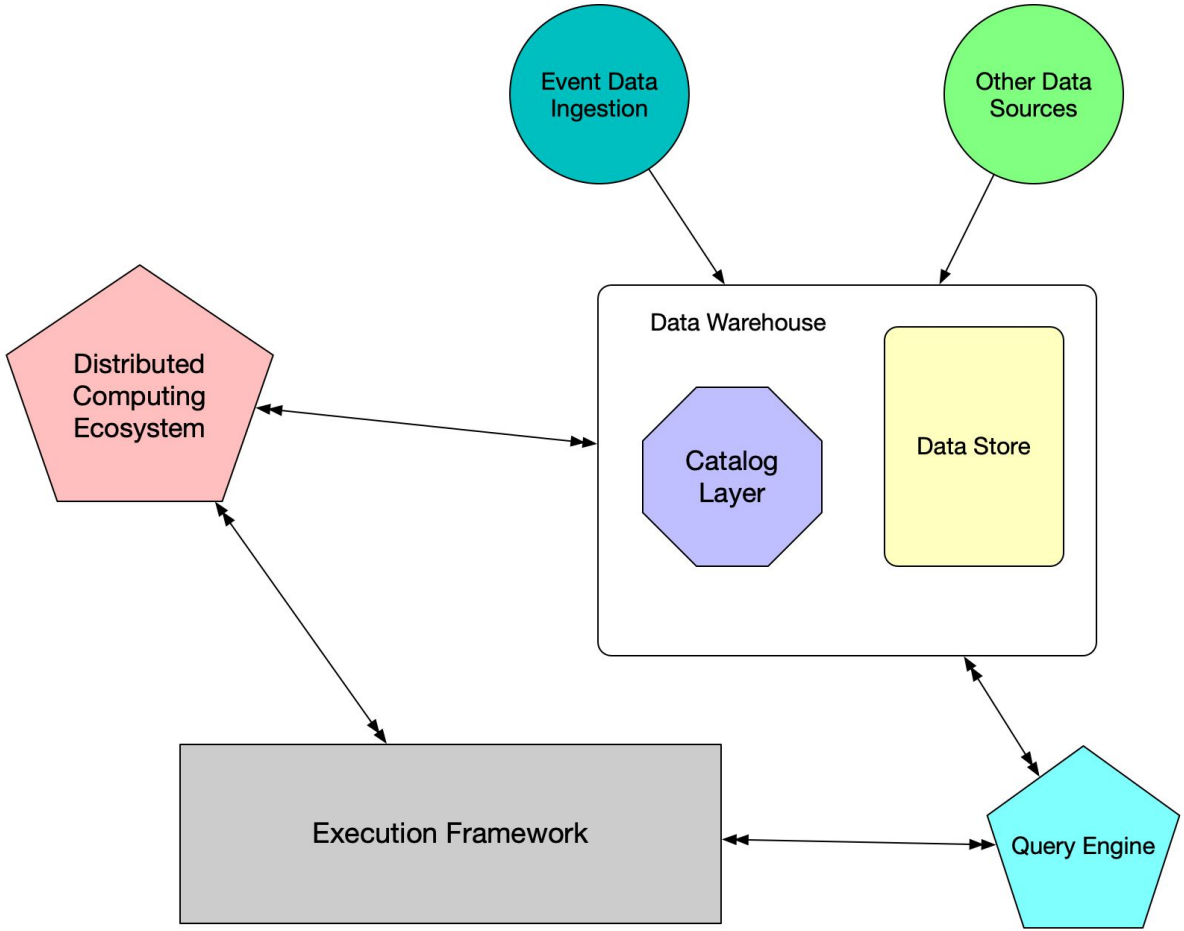


Figure 2 : Generation 2 of the Data Platform

Learning from Generation 2

Problems of Generation 2

1. Redundancy exists. More tools having similar methods/ functions.
2. Still might not have all the requirements, room for improvement.
3. Migration from the old generation is hard.
4. Things are not curated well. Need more guardrails. But this is ok, since the platform can be expanded upon.

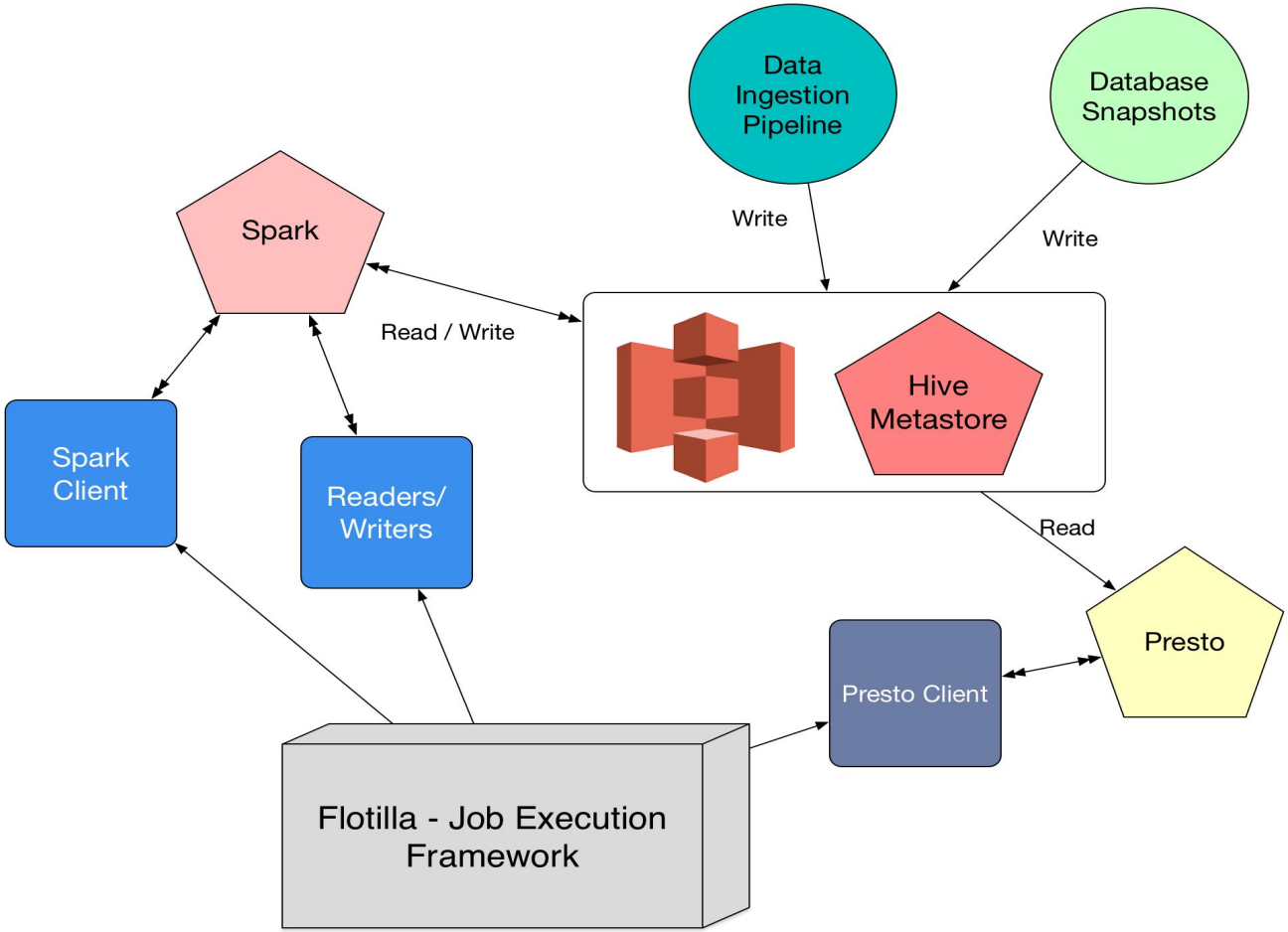


Figure 3 : Present day view of the Data Platform

Later Generations

Planning later generations

1. The focus is on designing for data science use cases rather than designing to expose technological capabilities.
2. The number of abstractions depends on the nature of the use cases.
3. Every aspect of the interface exposed to scientists is deliberately designed and crafted.
4. Migration from earlier generations should be deliberately designed, executed, and supported as much as the interface itself.
5. Executed slowly keeping in mind backwards compatibility.
6. The exposed interfaces should abstract enough to allow in situ replacement of backend technology for upgrades and capability evolution.

Let’s talk about an example of evolution..

Story of the Evolution of our Reader/Writers Tools

In Generation 1

What are Readers + Writers Tools?

Reader+ Writers

1. Born out of the need to use Python clients to read / write data for ETL
2. Pandas Dataframe was the default abstraction.
3. Implementation focused on adding the files to S3 and updating the Hive Metastore

Hive Metastore interface

1. Help read + update the Hive Metastore setup.
2. The Hive Metastore setup == MySQL Database + Thrift Layer + Rest Client
3. Became the only way to interact with the HiveMetastore

Why do we need Readers + Writers?

Use Cases are different from general spark usage or Ad hoc queries.

1. They help run large ETL, store results in one table which they then manipulate in Pandas
2. Help getting data in/out of one table in the warehouse in various row centric formats (JSON object per row, etc.)

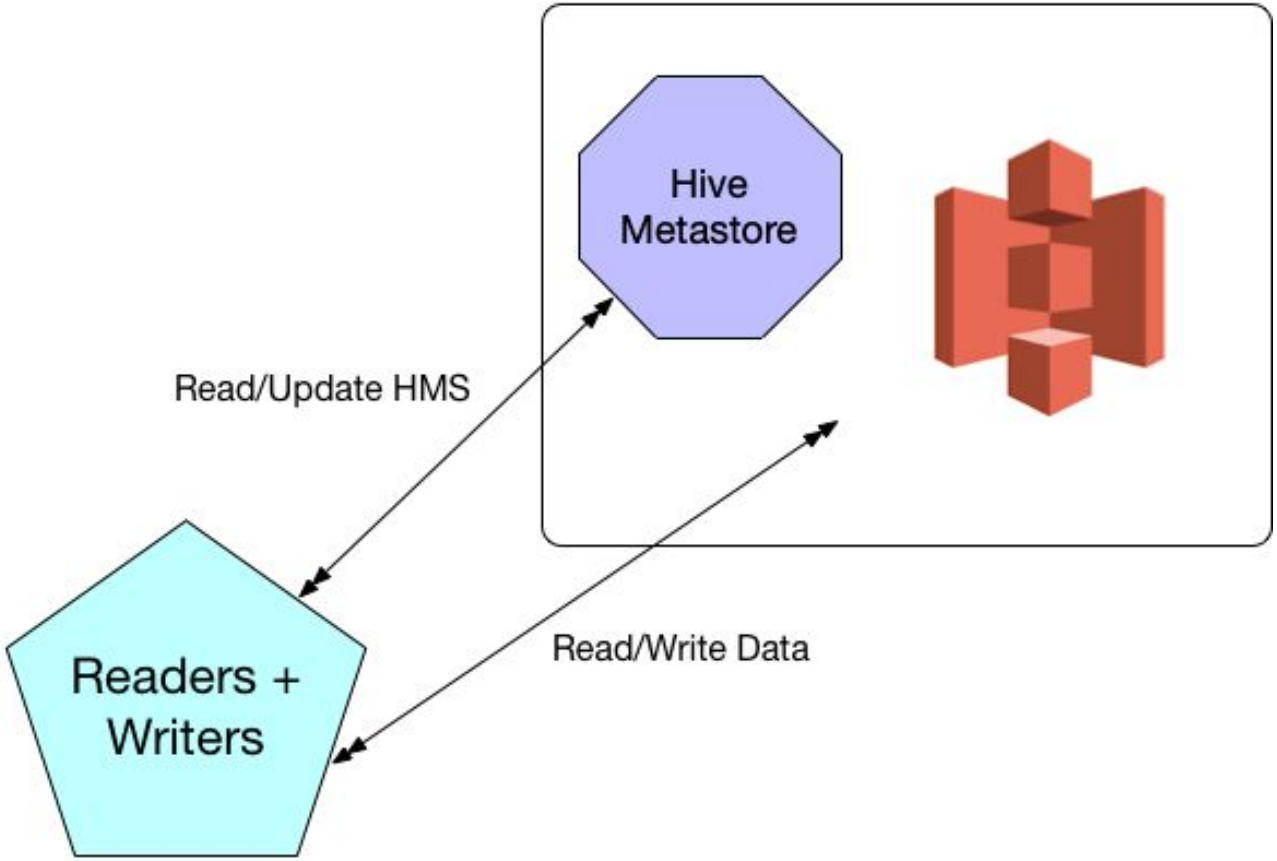


Figure 4 : Former Readers/Writers Infrastructure

Moving to Generation 2

Going from Generation 1 to 2

Readers + Writers

1. There was room for efficiency in the Readers + Writers since the implementation relied on pure python operations.
2. Pandas was the only data format to be used.
3. No validation for Pandas dataframe to match Hive types.

Going from Generation 1 to 2

Hive Metastore Interface

1. The hive implementation was inadequate (or inefficient) for some calls
2. The interface was not geared to add clear metadata and the metadata representation needed cleansing.

Phases to get to Generation 2..

Planning

Planning

1. Discussed the shortcomings of the current system and listed the new changes.
    1. Solicited feedback from Data Scientists
    2. Came up with a list of issues + ideas
2. Changing both Readers + Writers tools and the Hive Metastore needed coordination.
3. The first goal for both the tools was basic feature set + stability.

Design

Design

Readers + Writers

1. Dedicated Server + Client
2. Parity with interface of older tools
3. Clear semantics for methods
4. Make sure the Hive Metastore setup is compatible

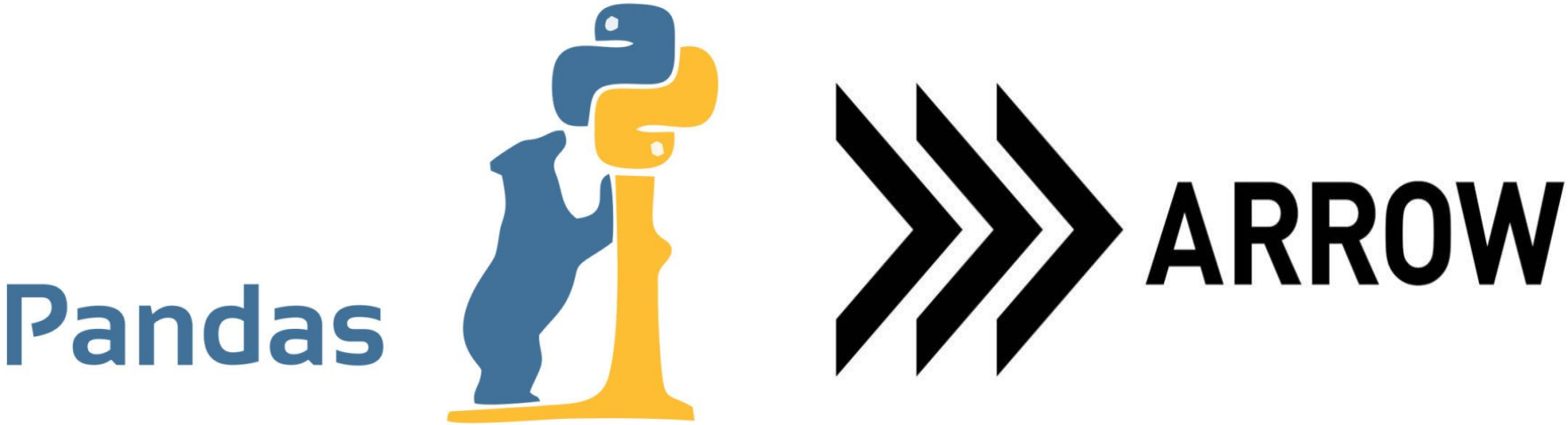
Design

Hive Metastore Interface

1. Splitting up the Rest API + Thrift Layer
2. Dedicated Server + Client
3. Spec the methods visible to the Data Scientists
4. Improve the representations of data from the Hive Metastore - making it consumable easily.
5. Validation and standardization of Hive table data.

Implementation

Readers and Writers Implementation


Pandas
ARROW

Why Arrow?

You could load a CSV file to a table but it needed quoting options, to specify how nulls should be handled, and distinguishing null strings from blank strings.

1. Arrow has a much better interchange format than CSV, which avoids the above issues.
2. Tight integration with Pandas but also has a general API allowing us to handle the other read/write cases
3. It is becoming more widely used
4. The Arrow/Parquet interaction -- key enabling step for the whole process

Arrow gives us the interchange format, but what should be the backend?

1. Thought about having our own server built on Hadoop Libraries but we had existing infrastructure that served Spark.
2. Spark was general purpose and gave querying power as well.
3. The choice of Presto limited us to reads only hence we went with Spark for reads + writes.

Existing Infrastructure for Spark

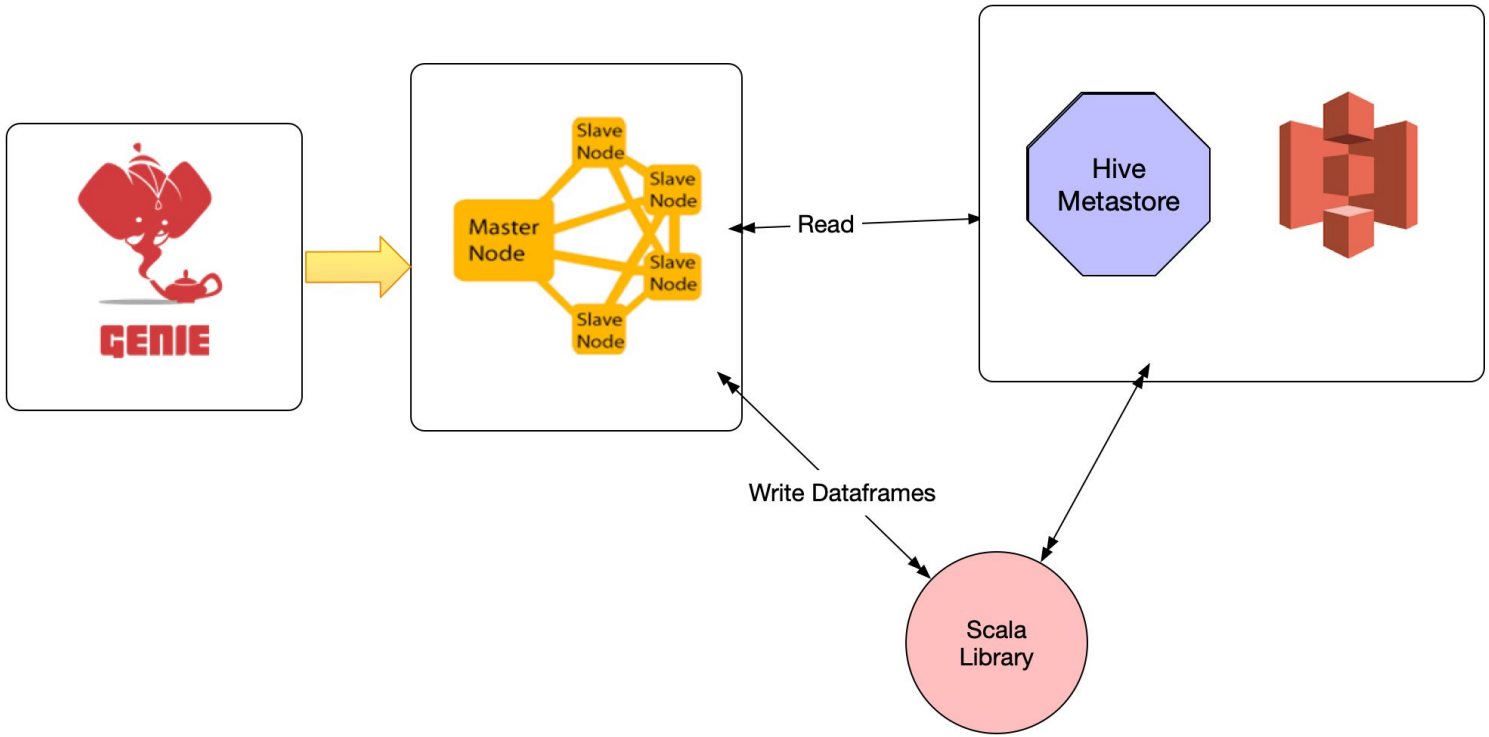


Figure 5 : Existing Infrastructure for Spark

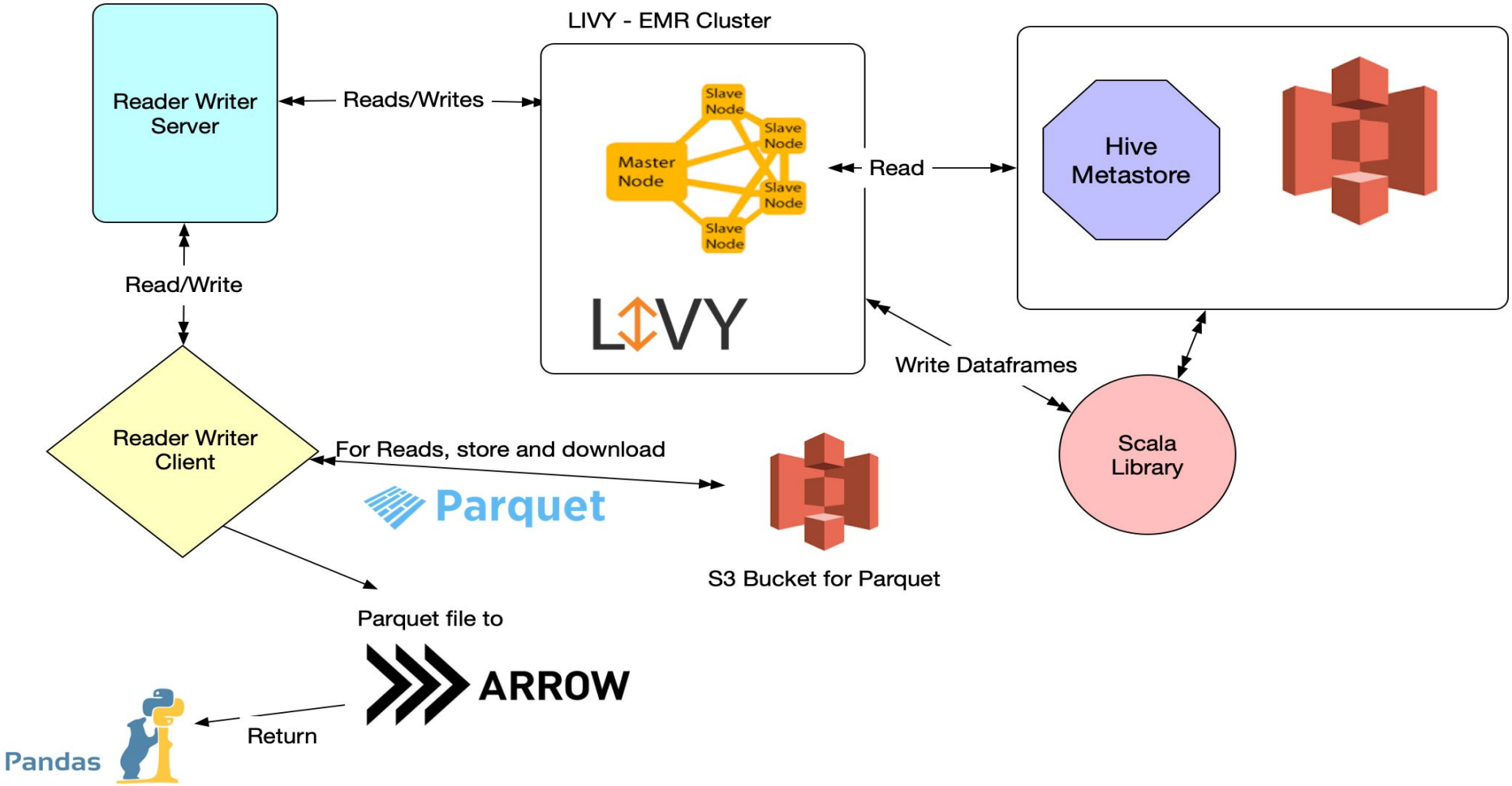


Figure 6 : Readers Writers Service

Benefits of Livy and the Reader Writer Server

Livy

1. Keeps warm Spark sessions that are easily reusable
2. Acts as a job server to support the reader/writer service
3. Uses our Spark libraries for writing.

Reader/ Writer Server

1. Simple API for the Client
2. Tracking and Caching of Livy Sessions
3. Cache other job metadata to reduce load on Livy

Hive Metastore Interface Implementation

Implementation Details

1. The plan was carved out to have decoupling of the Rest API and the Thrift Server itself.
    a. The REST Api was modeled to look like the old client interface
    b. The Thrift server was deployed as a service to talk to the Hive Metastore MySQL DB
2. The new interface would have the following pieces
    a. A Python Client with methods allowing to do things like create_table, get_partitions
    b. A server handling those methods + REST calls from the ecosystem.
        i. This server holds the interface to the Thrift code.

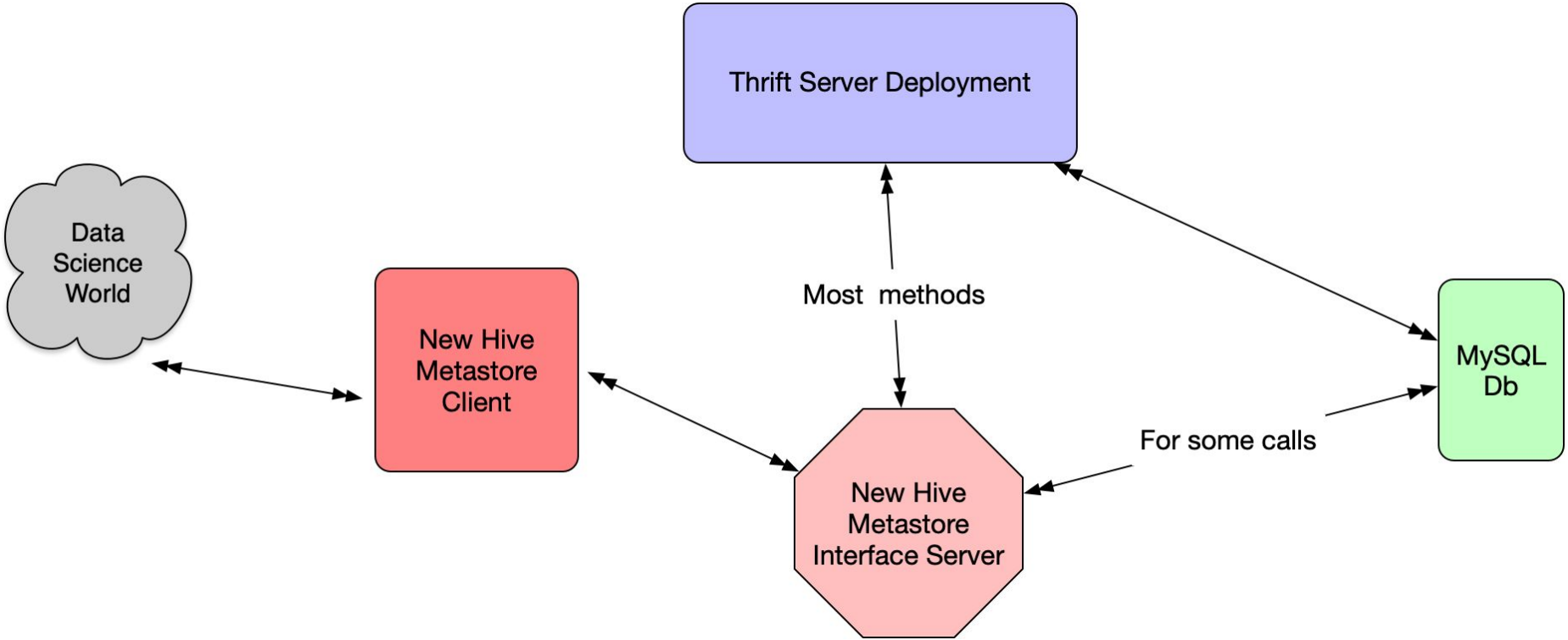


Figure 6 : Improved Hive Metastore Interface

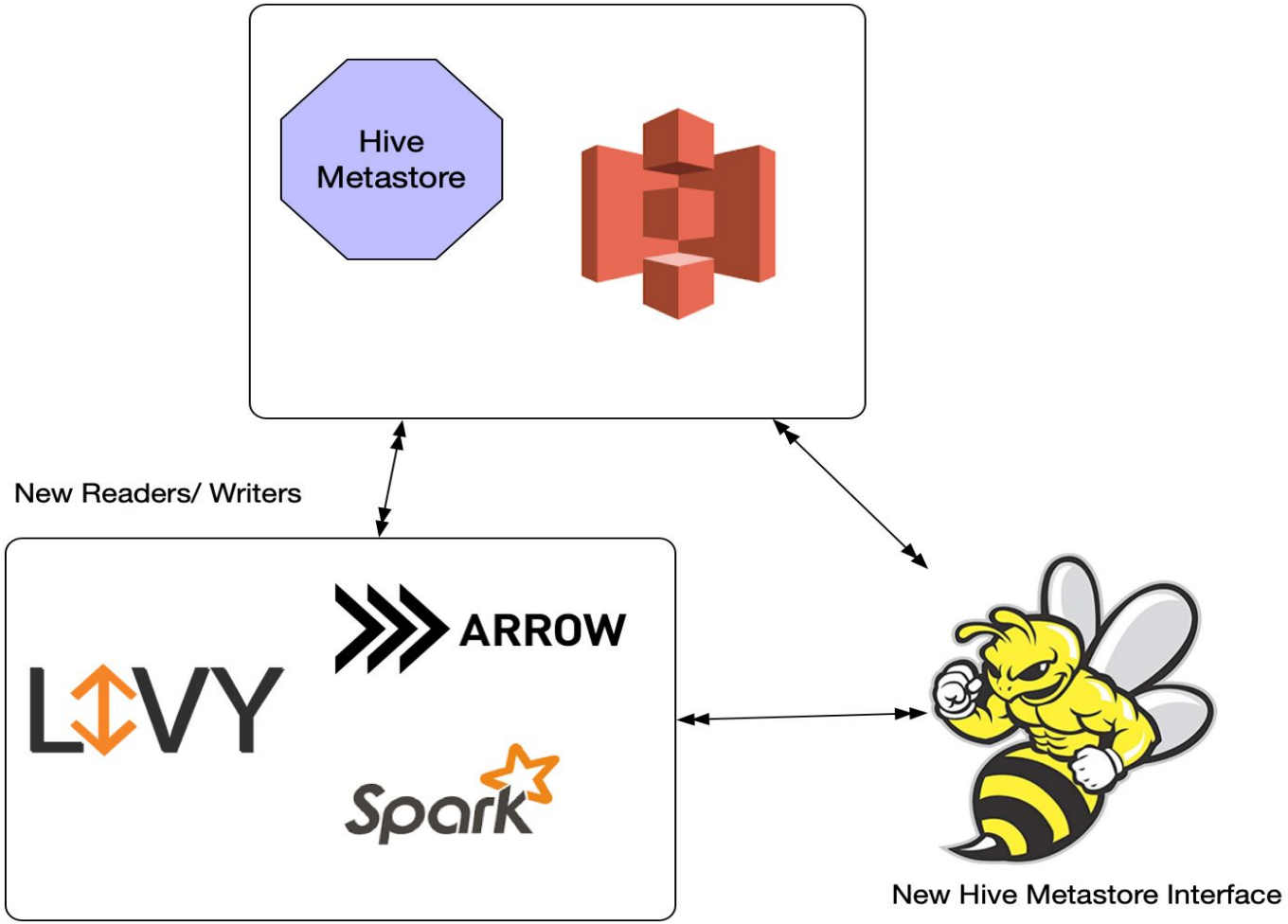


Figure 7 : Improved Readers+Writers Tools

Testing

Testing - **Readers + Writers**

1. Tested the pieces separately
    a. Livy setup was tested on its own
    b. The Reader Writer setup was unit tested
        i. Testing for data types – pandas to hive and vice versa

2. Once Livy was set up
    a. Integration tests
    b. Beta release within the sub-team to test it out

Testing - Hive Metastore Interface

1. Unit test + Integration tests
    a. This included a local client + server module
    b. Different tests for types and data structures from Thrift
2. High Impact testing
    a. The data quality improvement project was the production test for this layer
    b. Moving data from one bucket to the other while maintaining the current world
3. Operations were built into the interface to allow adding metadata fields in the Hive tables

Rollout + Migration

Rollout + Migration Process

1. Changing code isn't easy
    a. We built a compatibility layer to ease the way into the new world
2. We ran both the versions of the old and the new setup
3. Allowed testing to happen for a while with Data Scientists wherein
    a. They got a chance to be familiar with the setup
    b. Ask questions + concerns
    c. Help migrate themselves to the new setup + help others.
4. We addressed the concerns and the issues that came up prior to releasing and making it standard.

Takeaways

Learnings from the evolution

1. Migration isn't easy
   a. Time consuming and needs planning
   b. Coordinated with teams to help them. Large team with many pipelines.
2. There were some optimizations we needed to do on the Hive Metastore Interface
   a. Surpassing the Thrift Layer and going to the MySQL DB for some calls.
   b. Improving filtering for partitions
3. Livy took some experimentation to get it right
   a. How often do we need sessions?
4. How do we iterate on these pieces further?
   a. In our experience over the year, we have added multiple improvements on both the pieces.

Summary

To Summarize..

1. At Stitch Fix, our approach has been to provide self sufficiency first and iterate on the services moving forward.
2. Every steps need to be measured and intentional.
3. Migration from one generation to the other can be painful but there are ways to improve.
4. Always **learn** from your previous generation mistakes.

Thank you!

https://multithreaded.stitchfix.com/