Database Migration Service â€” Case Study #1

Author: Tianxing Liu

Claps: 30

Date: May 8

Itâ€™s common for multiple engineering teams to share the same database. However, over time, having a shared database may introduce performance and maintenance challenges. Imagine coordinating across different teams to perform a database version upgrade, different teams fighting for a single database migration number during development, or investigating a mysteriously hot database reader/writer instance.

When these challenges arise, itâ€™s time to split the shared database. This has happened many times at Klaviyo. Our standard playbook is to modify code to double write to the old and new database during the migration. However, in two recent projects, we tried AWS Database Migration Service (DMS), seeing if we could develop a new playbook that required less engineering.

In this blog post, I describe a case study where DMS worked as intended. We used it to split up an Aurora MySQL database that was supporting multiple data science services. The tables in this case study had millions of rows and order 10 writes per second. Later, we also successfully migrated tables with hundreds of millions of rows and order 100 writes per second.

One of my colleagues is currently leading a project to use DMS to migrate a service built on Aurora Postgres where the tables have billions of rows with order 10K writes per second. Weâ€™ve hit a number of challenges with DMS on that project that weâ€™re working through with AWS. Once we complete the project, weâ€™ll publish a separate blog post on lessons learned.

The challenges

There are challenges when a single database is shared across teams and services:

- **Single point of failure** â€” If the database fails, all services that depend on it fail. As more services and workloads are added, the risk of failure increases, as does the blast radius of the failure.
- **Unpredictable performance profile â€”** When there is a spike in read / write throughput, itâ€™s difficult to track down which services are responsible. If multiple services have spikes at the same time, itâ€™s tricky to disentangle cause and effect and hard to avoid pushing the limits of the databaseâ€™s resources such as CPU and memory.
- **Multiple ownership** â€” If everyone owns it, then no one owns it. Imagine there is a database critical alert that wakes you up in the middle of the night, but you donâ€™t have any context to fix the issue because the culprit belongs to a service owned by another team.
- **Maintenance coordination** â€” Databases need to be maintained. Sometimes maintenance actions (e.g. restart, upsize, upgrade) cause brief interruptions in ongoing connections lasting a few seconds, and sometimes minutes of downtime are required. With a shared

database, maintainers need to coordinate with all tenants of the database to understand the risks and impact of the maintenance actions.

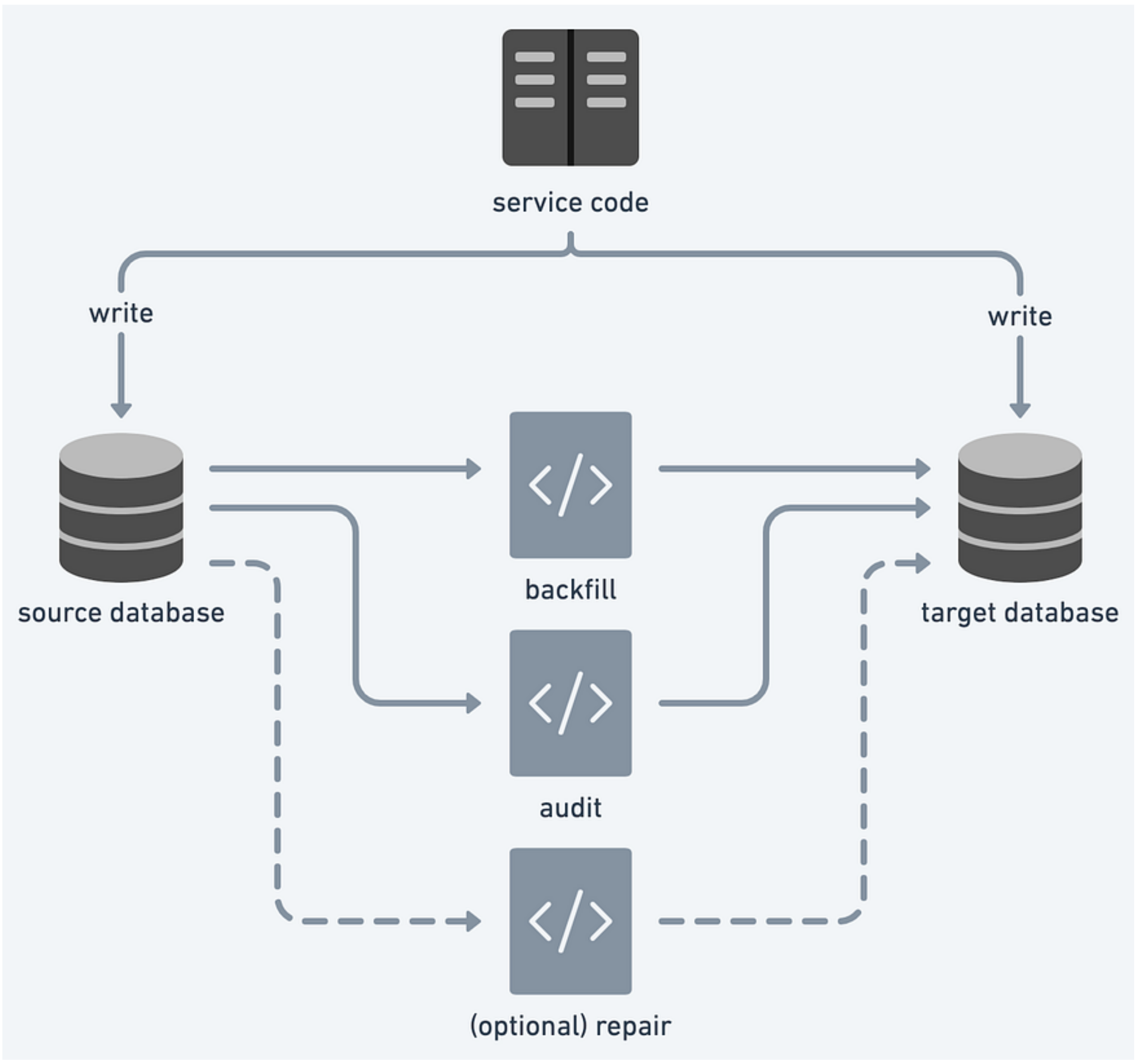
The database split project

When our data science organization formed in 2018, it stood up a single database cluster to support its initial production service. Over the years, the data science team, and later various subteams, developed new services, often choosing to add tables to that same database.

Last year, we decided it was time to split the database. To minimize engineering work, we assembled a tiger team and chose a particular service to migrate first. The goal was to migrate that service and document a standard migration strategy that other tenants of the shared database could follow. I led the tiger team. We decided to explore DMS as an alternative to doing things the old way.

Considerations

As mentioned above, most engineering teams at Klaviyo conduct splits using a double write pattern. This requires a sequence of steps to manually backfill data, implement code to simultaneously make updates to both the source and target database tables, make sure the tables are identical, and once confirmed, deprecate the source tables. Done right, thereâ€™s no database downtime.



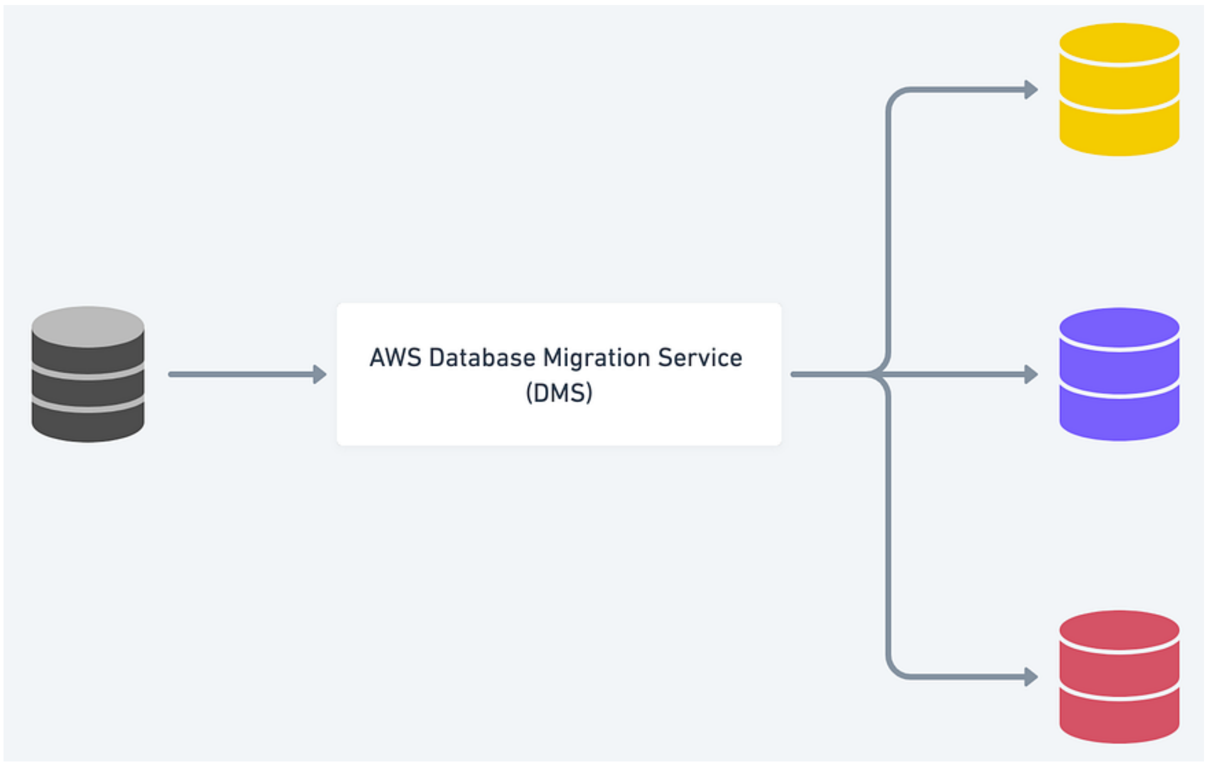
Double-write migration

However, it takes a lot of engineering. The team doing the migration needs close context around the query patterns and to identify all entry points to the tables to be replicated. The approach also requires a high degree of manual intervention. The team needs to manually manage which data to backfill, which data to replicate using double-write, and how to resolve issues resulting from inconsistencies between the source and the target tables. Lastly, during execution, the team needs to make multiple code changes and deployments to control starting and ending the replication.

We wanted to avoid spending so much engineering time on the split. We looked for a more universal approach that teams with tables on the shared database could easily follow. We also wanted to minimize required code changes to further derisk the project.

This is where DMS seemed attractive.

What is DMS?



DMS can even migrate from one DB engine to another

DMS is an AWS-managed tool introduced back in 2016 that allows migrations from one source database to multiple target databases. It supports on-prem and cloud-managed databases and schema conversion tools across different database engines. It handles backfilling data (*full load* in DMS terminology) and capturing ongoing changes (*ongoing replication* in DMS terminology) by running its data migration logic on an EC2 instance (*replication instance and task* in DMS terminology). Under the hood, it looks at the binary logs to know what database transactions have occurred, and uses them to replicate the ongoing changes, which means that no custom coding is needed to double write from the service.

Features

The following DMS features seemed particularly attractive for our project.

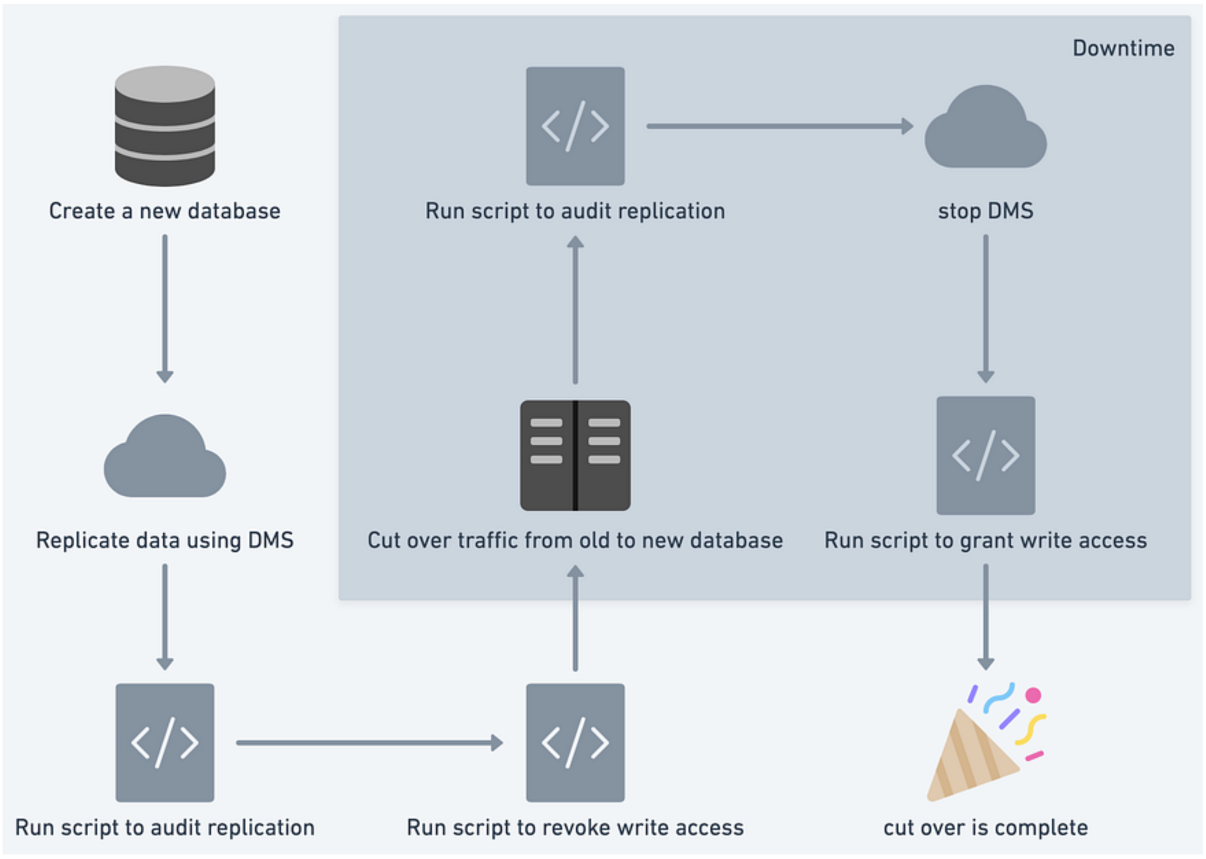
- Backfilling existing data
- Replicating ongoing changes
- Selecting a subset of tables to replicate
- Changing the table definition during replication
- Row-by-row data validation

Comparison

Our traditional "double write" approach	DMS
Code changes required	Code changes not required
Engineers manage backfill	DMS handles backfill

Using DMS

Hereâ€™s how we used DMS to split the data science database.


Downtime
Create a new database
Replicate data using DMS
Run script to audit replication
Run script to revoke write access
Cut over traffic from old to new database
Run script to audit replication
stop DMS
Run script to grant write access
cut over is complete

High level steps we took to complete the cutover

Create a new database

Our source database was Aurora MySQL, so we created our new target database using the same setup. Then, we used a database migration tool (overloaded term unfortunately, here it means a tool like alembic) to generate the relevant tables in the target database.

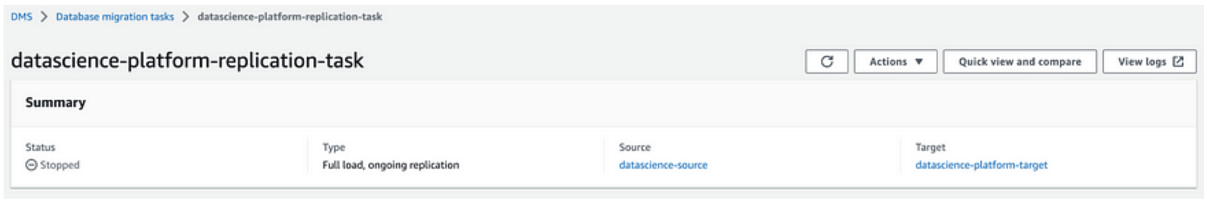
Replicate data using DMS

We set up and configured a bunch of DMS resources:

- Replication instance

- Replication task
- Source endpoint
- Target endpoint
- Table selection rules
- Table transformation rules

At a high level, our configuration told DMS to connect to the source and target databases as admin users, select the tables of interest, replicate their data to the target tables, and validate.


DMS > Database migration tasks > datascience-platform-replication-task
Actions
Quick view and compare
View logs

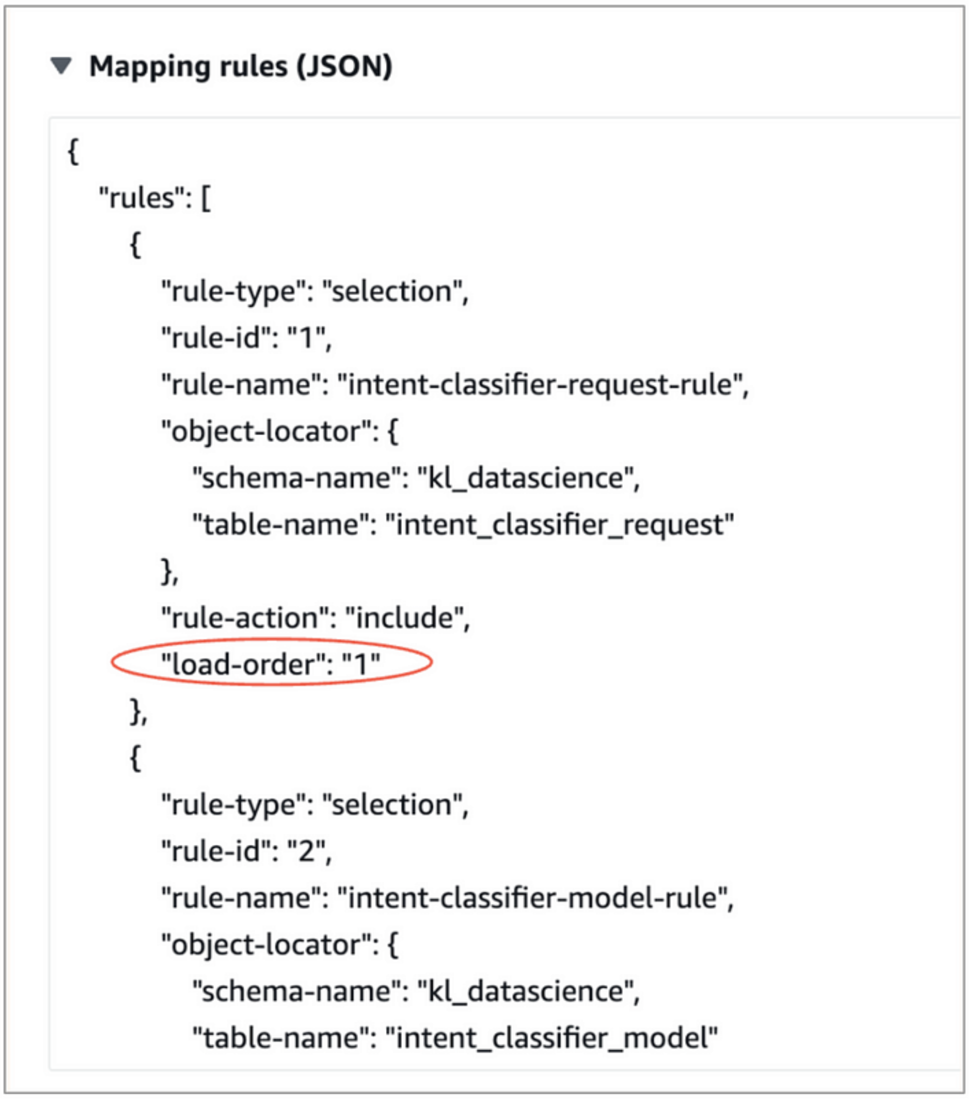
datascience-platform-replication-task

Summary

Status	Type	Source	Target
Stopped	Full load, ongoing replication	datascience-source	datascience-platform-target

Screenshot from DMS console. Source endpoint was the shared database, target database was a new database owned by an individual engineering team

We also defined rules so that DMS knew how to load the tables.

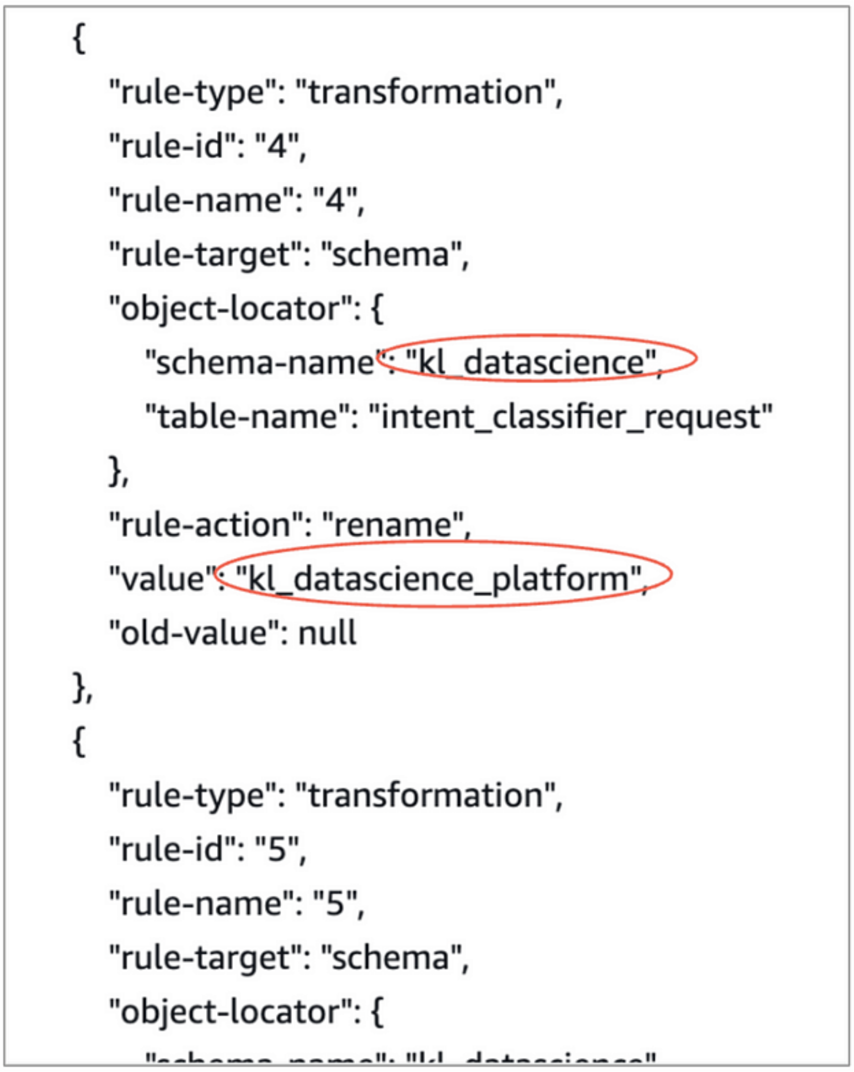


Mapping rules (JSON)

```
{
  "rules": [
    {
      "rule-type": "selection",
      "rule-id": "1",
      "rule-name": "intent-classifier-request-rule",
      "object-locator": {
        "schema-name": "kl_datascience",
        "table-name": "intent_classifier_request"
      },
      "rule-action": "include",
      "load-order": "1"
    },
    {
      "rule-type": "selection",
      "rule-id": "2",
      "rule-name": "intent-classifier-model-rule",
      "object-locator": {
        "schema-name": "kl_datascience",
        "table-name": "intent_classifier_model"
```

Example of selection rule with load order

One interesting thing here was that the tables we wanted to replicate had foreign key relationships. There are a few ways to configure DMS to prevent foreign key integrity issues. In our case, we defined the load order of tables.



```
{
   "rule-type": "transformation",
   "rule-id": "4",
   "rule-name": "4",
   "rule-target": "schema",
   "object-locator": {
      "schema-name": "kl_datascience",
      "table-name": "intent_classifier_request"
   },
   "rule-action": "rename",
   "value": "kl_datascience_platform",
   "old-value": null
},
{
   "rule-type": "transformation",
   "rule-id": "5",
   "rule-name": "5",
   "rule-target": "schema",
   "object-locator": {
      "schema-name": "kl_datascience"
```

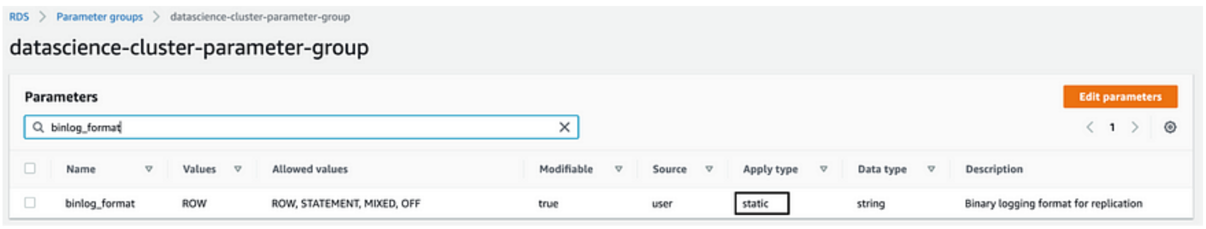
Example of transformation rule

Another rule we defined said how to transform the schema name. The source table had the schema â€œkl_datascienceâ€� and the target table had the schema â€œkl_datascience_platform.â€� This rule told DMS how to replicate the data into the correct tables.

Binary Logs

DMS requires the source database to have binary logs turned on. AWS has this documentation on how to enable it for Aurora MySQL. There are several ramifications to turning on binary logs. You should be aware of them before proceeding.

Drops ongoing connections â€” Turning on binary logs requires a reboot on the writer instance because the binary log parameter is static. This will forcefully drop any ongoing database connections for a brief moment (usually a few seconds, but this may vary), causing an influx of service errors. This is an interruption that the users may notice.



RDS > Parameter groups > datascience-cluster-parameter-group

datascience-cluster-parameter-group

Parameters

binlog_format

Name	Values	Allowed values	Modifiable	Source	Apply type	Data type	Description
binlog_format	ROW	ROW, STATEMENT, MIXED, OFF	true	user	static	string	Binary logging format for replication

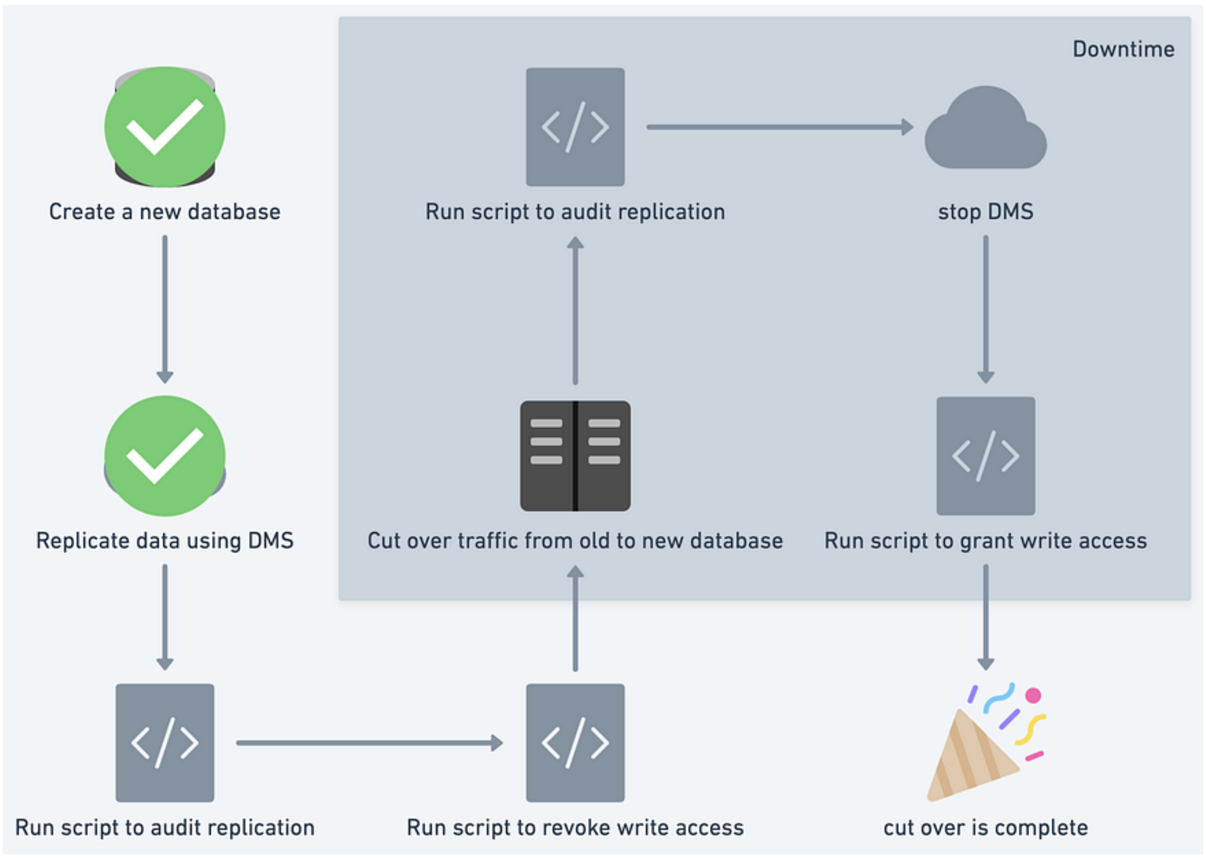
Source database parameter group

More data to store â€" Binary logs require more storage because they keep track of every database transaction. This is reflected in a Cloudwatch metric called Volume Bytes Used. Although this shouldnâ€™t be an issue for most of the cases, itâ€™s something to keep an eye on.

May break certain database tools â€" At Klaviyo, we use an open source tool pt-online-schema-change for updating the database schema without interrupting ongoing production traffic. This tool is conservative, and by default it will start failing when it detects binary logs are enabled. It does this because there are risks with modifying table schema with replication filters. In our case, since we didnâ€™t rely on replication filters, we could safely work around it by explicitly bypassing this check condition `--check-replication-filters` for this tool. However, this may not be safe in other use cases, so proceed with caution.

Starting the replication

After everything was set up, we pressed â€œstartâ€� on the replication task and let it do its magic. It completed full load first, and then we let it continue to process the ongoing replications. Now, we had completed two steps in the cutover process.


Downtime
Create a new database
Run script to audit replication
stop DMS
Replicate data using DMS
Cut over traffic from old to new database
Run script to grant write access
Run script to audit replication
Run script to revoke write access
cut over is complete

Run script to audit replication

We created a custom script that connected to our source and target databases, compared row counts, and, based on flags, performed other (slower) sanity checks such as comparing the actual rows. We ran this script repeatedly to monitor replication status and ensure the target database was keeping up as expected.

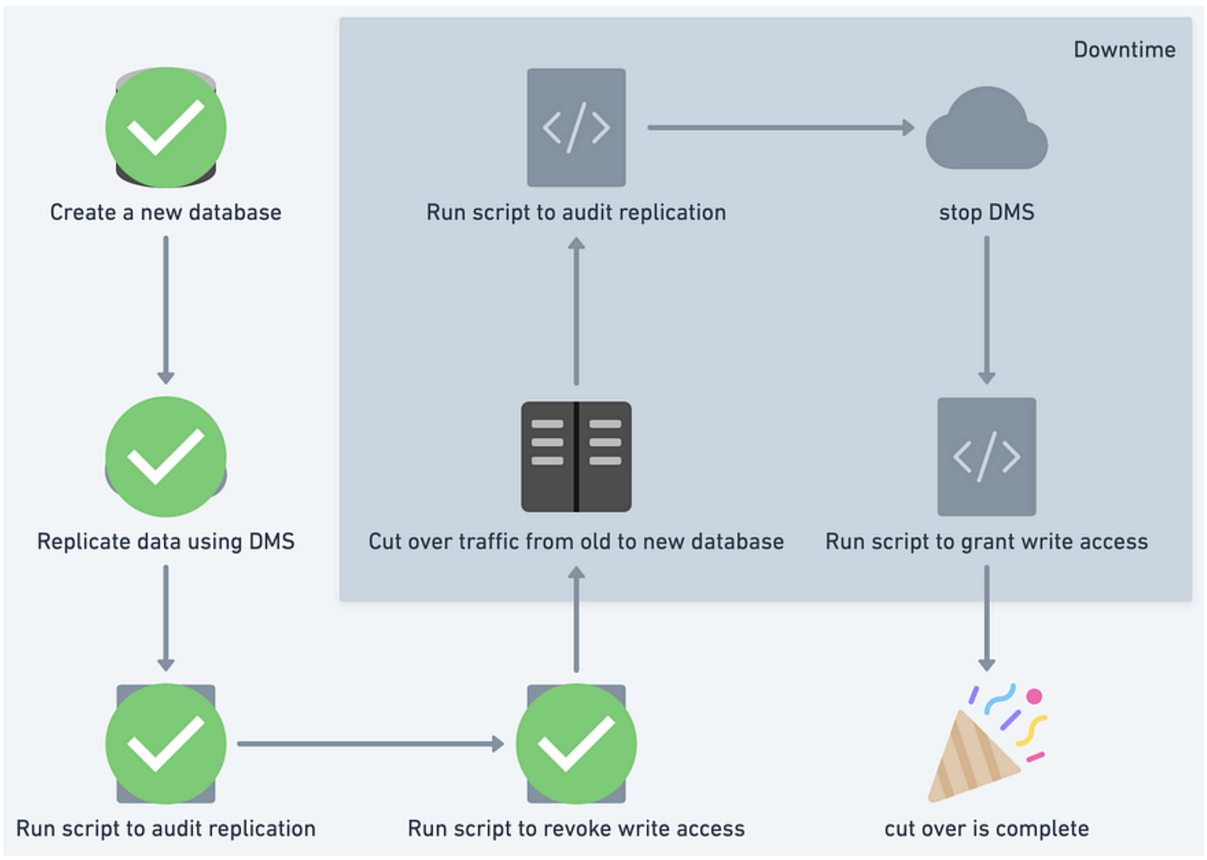
You may be wondering why we bothered implementing this script since DMS already offers logs and a UI console that displays row counts and validation status. We wanted visibility to the source of truth (the tables themselves). We noticed that the DMS UI sometimes had a delay before reflecting the up-to-date row count. We did in fact run into situations where if we were relying on the DMS UI alone we would have needlessly delayed proceeding with the cutover.

Run script to revoke write access

We implemented another script to revoke write access (insert, update, delete) on the target database from the SQL user that the service code uses. This was to prepare for the cutover and allowed us to control when the downtime began. Note that, by design, this did not block DMS replication which ran as a different database user.

Ready for cutover

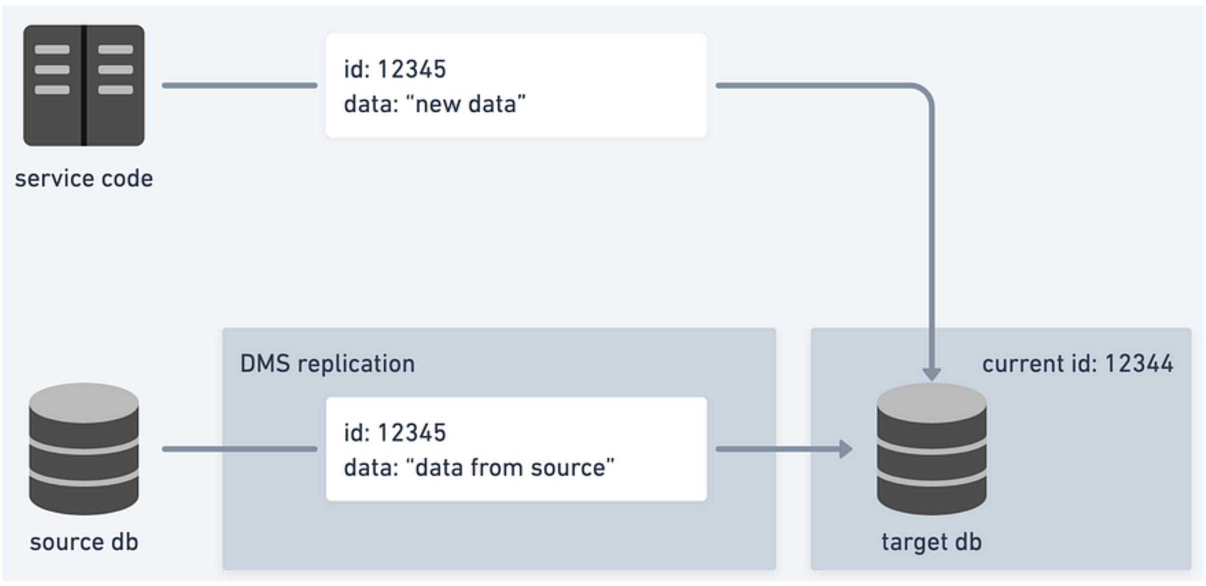
We had completed the preparation steps for the cutover. The next steps were to be done during a maintenance window with the stakeholders informed about the downtime.


Downtime
Create a new database
Run script to audit replication
stop DMS
Replicate data using DMS
Cut over traffic from old to new database
Run script to grant write access
Run script to audit replication
Run script to revoke write access
cut over is complete

However, before we proceed, letâ€™s discuss why we needed downtime.

DMS with downtime

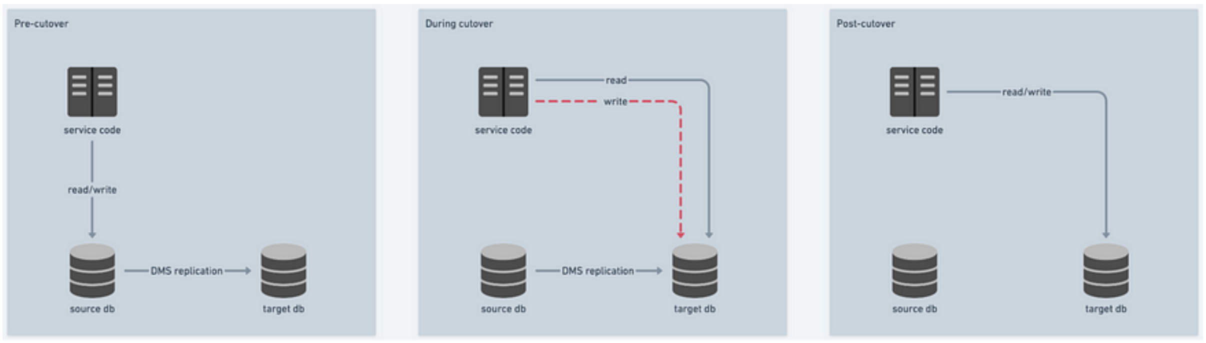
There is sometimes a tradeoff between data integrity and service availability with database maintenance. Hereâ€™s an example. Suppose you have a table that has an auto-incrementing primary key. The last assigned key was 12345 in the source database, but there was a replication delay to the target so the targetâ€™s last assigned key was 12344. Before row 12345 was replicated, a new request came into the service code and inserted data into the target. Now, replicating row 12345 will fail due to key conflicts.



Example of database insertion conflict

To resolve this, we could either redesign our system so that we no longer depend on the insertion order of the records (for example, many teams at Klaviyo use ULID), or tolerate some degree of service write downtime to allow the replication to catch up before allowing new writes.

The acceptability of downtime varies by service. Some services are read-heavy, so having a write downtime has minimal impact; others require a more careful decision between investing more engineering time and tolerating write downtime. An example of a middle ground is to tolerate data delay where the failed writes during the downtime are replayed at a later time (by pausing asynchronous workers, pausing cron jobs, or capturing/replaying failure logs). We let the owners of each service decide how to approach this. For the first service we migrated, we decided that up to an hour of write delay due to downtime was acceptable.



During cutover, service code was not able to write to the target database until replication was fully caught up.

Begin the cutover

Cutover required merging and deploying a pull request so that the service code pointed to the target database instead of the source. Since we previously revoked the serviceâ€™s write access on the target database, deploying this PR marked the start of downtime. (Although in the first service we migrated, we simply turned off calls to the service that required writing on the caller side, so there were no attempts to write to the database during the downtime period.)

Run script to audit replication (again)

During the downtime, we continued using the audit replication script to monitor the replication status. We waited until the source and target database tables had reached an identical state (i.e. DMS replication had fully caught up). This was almost instant for our tables with <10 writes per second.

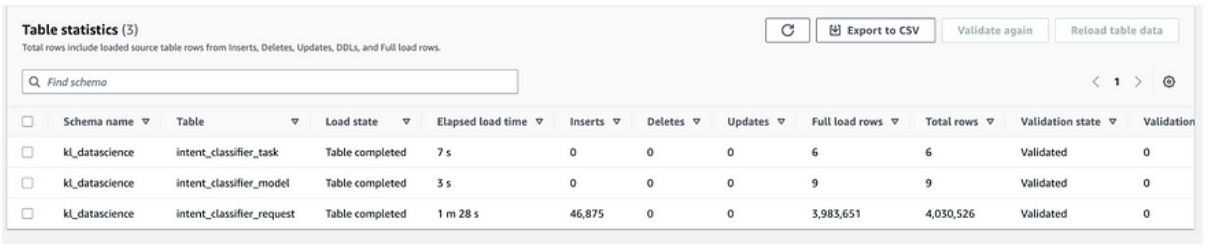


Table statistics (3)

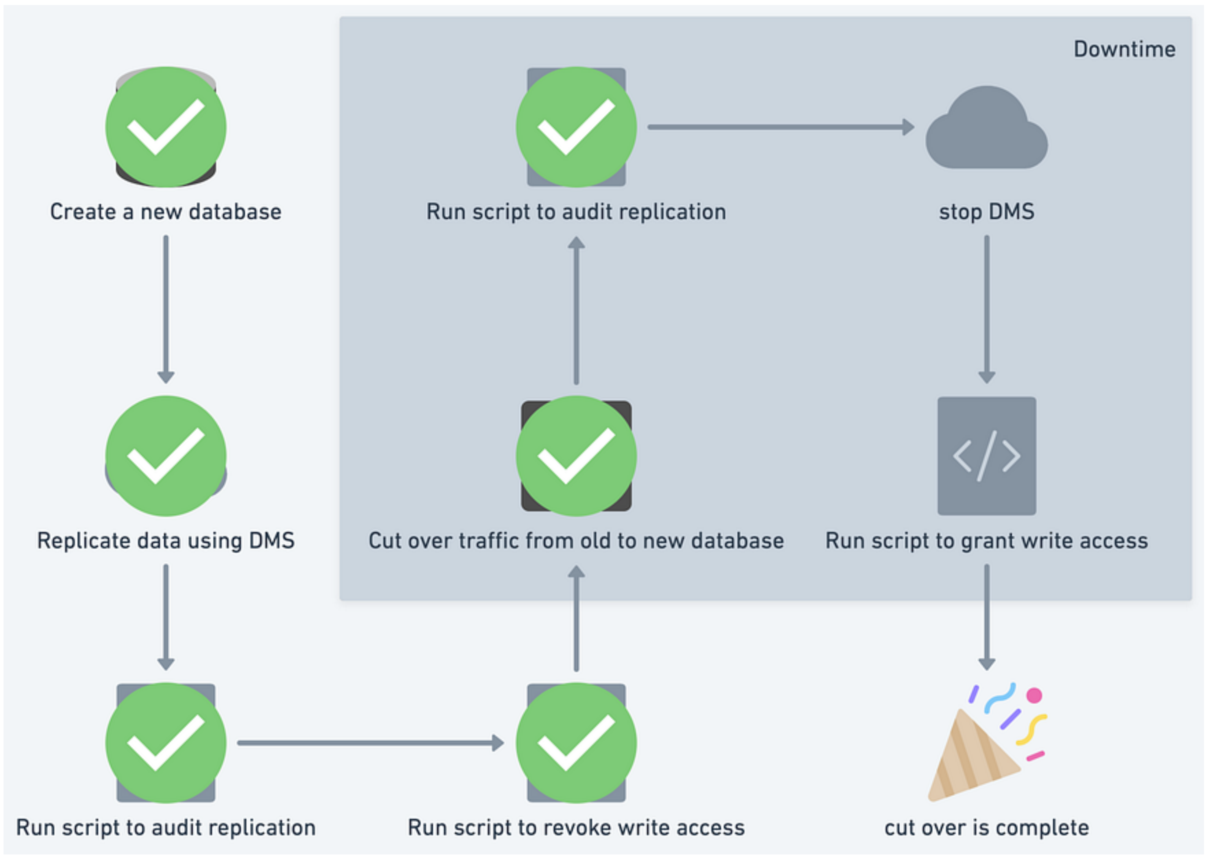
Total rows include loaded source table rows from Inserts, Deletes, Updates, DDLs, and Full load rows.

Schema name	Table	Load state	Elapsed load time	Inserts	Deletes	Updates	Full load rows	Total rows	Validation state	Validation
kl_datascience	intent_classifier_task	Table completed	7 s	0	0	0	6	6	Validated	0
kl_datascience	intent_classifier_model	Table completed	3 s	0	0	0	9	9	Validated	0
kl_datascience	intent_classifier_request	Table completed	1 m 28 s	46,875	0	0	3,983,651	4,030,526	Validated	0

DMS console at start of downtime

Replication complete

After we confirmed that the source and target tables had the exact same row count, we considered the replication complete.


Downtime
Create a new database
Run script to audit replication
stop DMS
Replicate data using DMS
Cut over traffic from old to new database
Run script to grant write access
Run script to audit replication
Run script to revoke write access
cut over is complete

Stop DMS

We used the AWS UI to stop the DMS replication task. DMS had done its job!

Run script to grant write access

This was the step that ended the downtime. The script granted database write permission to the service code user.

For the simplicity of our playbook, our script revoked/granted write access on the database level, like this:

```
REVOKE INSERT, UPDATE, DELETE ON kl_datascience_platform.* FROM `datascien

FLUSH PRIVILEGES;
```

According to the MySQL documentation, database level privileges may not take effect until the client has refreshed its connection. We had to force a new deployment where the running containers drained the existing connections and spun up new connections to the database. Alternatively, teams can avoid this connection refresh step by manipulating the privileges on a table level:

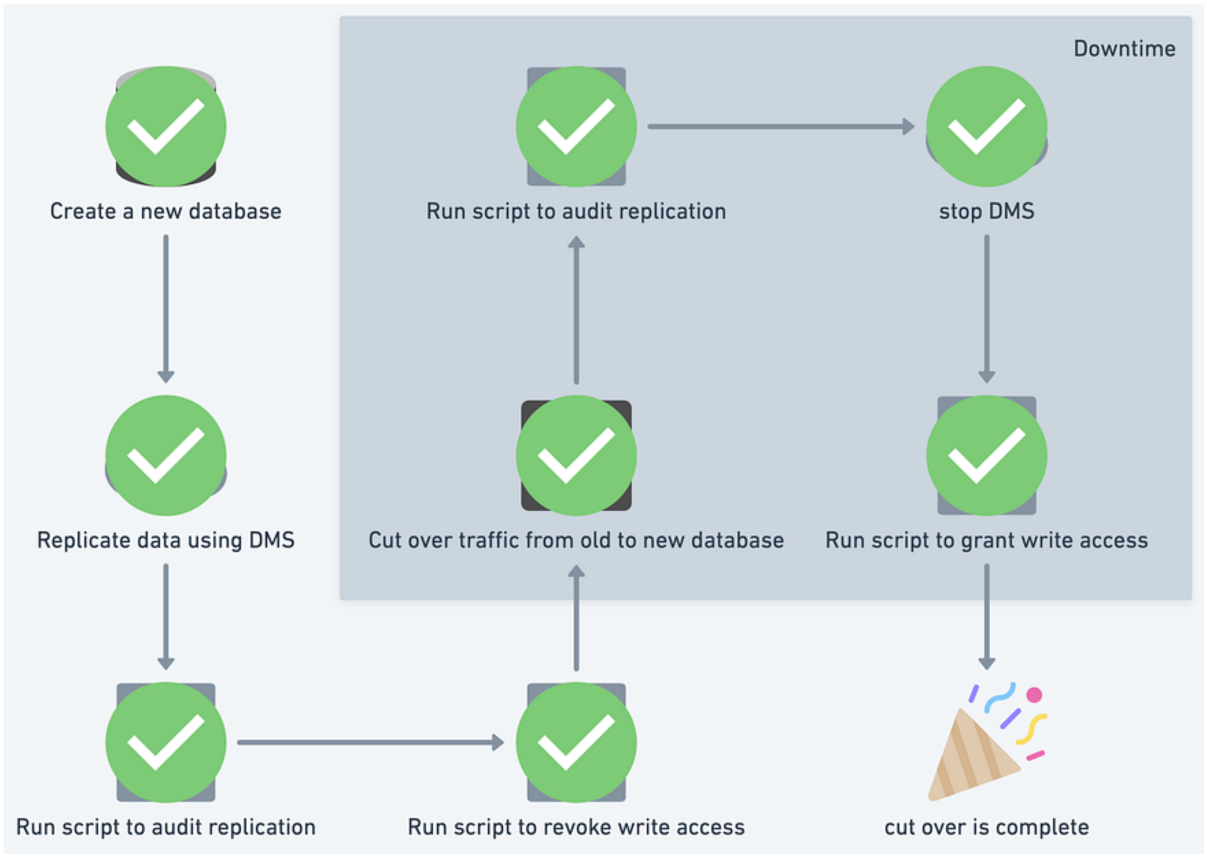
```
REVOKE INSERT, UPDATE, DELETE ON kl_datascience_platform.`intent_classifie

REVOKE INSERT, UPDATE, DELETE ON kl_datascience_platform.`intent_classifie

REVOKE INSERT, UPDATE, DELETE ON kl_datascience_platform.`intent_classifie
```

```
FLUSH PRIVILEGES;
```

Cutover complete


Downtime
Create a new database
Run script to audit replication
stop DMS
Replicate data using DMS
Cut over traffic from old to new database
Run script to grant write access
Run script to audit replication
Run script to revoke write access
cut over is complete

Total downtime

For our first service, we incurred about 40 minutes of downtime. The 40 minutes were not caused by DMS replication to catch up â€" that took a few seconds. Most of the downtime was turning on and off callers, deploying cutover code, checking the outputs of our audit script, and forcing client deployments. Using the same approach, but with more practice, we were confident we could reduce the downtime to 15 minutes, and indeed, other teams following the playbook and splitting tables off from the database were able to achieve that.

Conclusion

We succeeded in splitting up our database! For certain types of database split projects, using DMS and the playbook we developed will save considerable engineering time compared with our traditional approach. Look out for a future blog post with learnings from using DMS to migrate tables with billions of rows.