AWS Whitepaper

Migrating Your Databases to Amazon Aurora

Migrating Your Databases to Amazon Aurora: AWS Whitepaper

Table of Contents

Migrating Your Databases to Amazon Aurora

Publication date: **July 28, 2021** (*Document history*)

Abstract

Amazon Aurora is a MySQL and PostgreSQL-compatible, enterprise grade relational database engine. Amazon Aurora is a cloud-native database that overcomes many of the limitations of traditional relational database engines. The goal of this whitepaper is to highlight best practices of migrating your existing databases to Amazon Aurora. It presents migration considerations and the step-by-step process of migrating open- source and commercial databases to Amazon Aurora with minimum disruption to the applications.

Are you Well-Architected?

The AWS Well-Architected Framework helps you understand the pros and cons of the decisions you make when building systems in the cloud. The six pillars of the Framework allow you to learn architectural best practices for designing and operating reliable, secure, efficient, cost-effective, and sustainable systems. Using the AWS Well-Architected Tool, available at no charge in the AWS Management Console, you can review your workloads against these best practices by answering a set of questions for each pillar.

For more expert guidance and best practices for your cloud architecture—reference architecture deployments, diagrams, and whitepapers—refer to the AWS Architecture Center.

Introduction

For decades, traditional relational databases have been the primary choice for data storage and persistence. These database systems continue to rely on monolithic architectures and were not designed to take advantage of cloud infrastructure. These monolithic architectures present many challenges, particularly in areas such as cost, flexibility, and availability. In order to address these challenges, AWS redesigned relational database for the cloud infrastructure and introduced Amazon Aurora.

Amazon Aurora is a MySQL and PostgreSQL-compatible relational database engine that combines the speed, availability, and security of high-end commercial databases with the simplicity and cost-effectiveness of open-source databases. Aurora provides up to five times better performance than

MySQL, three times better performance than PostgreSQL and comparable performance of high-end commercial databases. Amazon Aurora is priced at 1/10th the cost of commercial engines.

Amazon Aurora is available through the Amazon Relational Database Service (Amazon RDS) platform. Like other Amazon RDS databases, Aurora is a fully managed database service. With the Amazon RDS platform, most database management tasks such as hardware provisioning, software patching, setup, configuration, monitoring, and backup are completely automated.

Amazon Aurora is built for mission-critical workloads and is highly available by default. An Aurora database cluster spans multiple Availability Zones in a Region, providing out-of-the-box durability and fault tolerance to your data across physical data centers. An Availability Zone is composed of one or more highly available data centers operated by Amazon. Availability Zones are isolated from each other and are connected through low-latency links. Each segment of your database volume is replicated six times across these Availability Zones.

Amazon Aurora enables dynamic resizing for database storage space. Aurora cluster volumes automatically grow as the amount of data in your database increases with no performance or availability impact—so there is no need for estimating and provisioning large amount of database storage ahead of time. The storage space allocated to your Amazon Aurora database cluster will automatically increase up to a maximum size of 128 tebibytes (TiB) and will automatically decrease when data is deleted.

Aurora's automated backup capability supports point-in-time recovery of your data, enabling you to restore your database to any second during your retention period, up to the last five minutes. Automated backups are stored in Amazon Simple Storage Service (Amazon S3), which is designed for 99.999999999% durability. Amazon Aurora backups are automatic, incremental, and continuous and have no impact on database performance.

For applications that need read-only replicas, you can create up to 15 Aurora Replicas per Aurora database with very low replica lag. These replicas share the same underlying storage as the source instance, lowering costs and avoiding the need to perform writes at the replica nodes. Optionally, Aurora Global Database can be used for high read throughputs across six Regions up to 90 read replicas.

Amazon Aurora is highly secure and allows you to encrypt your databases using keys that you create and control through AWS Key Management Service (AWS KMS). On a database instance running with Amazon Aurora encryption, data stored at rest in the underlying storage is encrypted, as are the automated backups, snapshots, and replicas in the same cluster. Amazon Aurora uses SSL (AES-256) to secure data in transit.

For a complete list of Aurora features, refer to the Amazon Aurora product page. Given the rich feature set and cost effectiveness of Amazon Aurora, it is increasingly viewed as the go-to database for mission-critical applications.

Amazon Aurora Serverless v2 is the new version of Aurora Serverless, an on- demand, automatic scaling configuration of Amazon Aurora that automatically starts up, shuts down, and scales capacity up or down based on your application's needs. It scales instantly from hundreds to hundreds-of-thousands of transactions in a fraction of a second. As it scales, it adjusts capacity in fine-grained increments to provide just the right amount of database resources that the application needs. There is no database capacity for you to manage, you pay only for the capacity your application consumes, and you can save up to 90% of your database cost compared to the cost of provisioning capacity for peak.

Aurora Serverless v2 is a simple and cost-effective option for any customer who cannot easily allocate capacity because they have variable and infrequent workloads or have a large number of databases. If you can predict your application’s requirements and prefer the cost certainty of fixed-size instances, then you may want to continue using fixed-size instances.

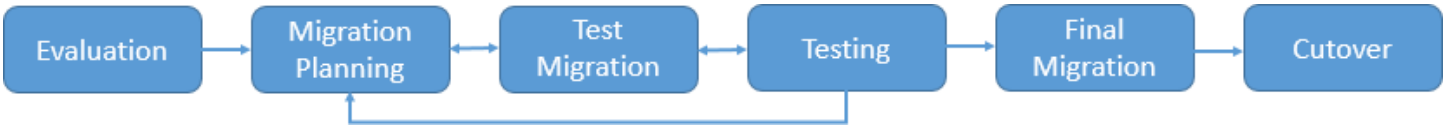
Amazon Aurora capabilities discussed in this whitepaper apply to both MySQL and PostgreSQL database engines, unless otherwise specified. However, the migration practices discussed in this paper are specific to Aurora MySQL database engine. For more information about Aurora best practices specific to PostgreSQL database engine, refer to Working with Amazon Aurora PostgreSQL in the Amazon Aurora user guide.

Database migration considerations

A database represents a critical component in the architecture of most applications. Migrating the database to a new platform is a significant event in an application's lifecycle and may have an impact on application functionality, performance, and reliability. You should take a few important considerations into account before embarking on your first migration project to Amazon Aurora.

Migration phases

Because database migrations tend to be complex, we advocate taking a phased, iterative approach.



Migration phases

Application considerations

Evaluate Aurora features

Although most applications can be architected to work with many relational database engines, you should make sure that your application works with Amazon Aurora.

Amazon Aurora is designed to be wire-compatible with MySQL 5.6 and 5.7. Therefore, most of the code, applications, drivers, and tools that are used today with MySQL databases can be used with Aurora with little or no change.

However, certain MySQL features, like the MyISAM storage engine, are not available with Amazon Aurora. Also, due to the managed nature of the Aurora service, SSH access to database nodes is restricted, which may affect your ability to install third-party tools or plugins on the database host.

Performance considerations

Database performance is a key consideration when migrating a database to a new platform. Therefore, many successful database migration projects start with performance evaluations of the new database platform. Although the *Amazon Aurora Performance Assessment* whitepaper gives you a decent idea of overall database performance, these benchmarks do not emulate the data

access patterns of your applications. For more useful results, test the database performance for time-sensitive workloads by running your queries (or subset of your queries) on the new platform directly.

Consider these strategies:

- If your current database is MySQL, migrate to Amazon Aurora with downtime and performance test your database with a test or staging version of your application or by replaying the production workload.
- If you are on a non-MySQL-compliant engine, you can selectively copy the busiest tables to Amazon Aurora and test your queries for those tables. This gives you a good starting point. Of course, testing after complete data migration will provide a full picture of real-world performance of your application on the new platform.

Amazon Aurora delivers comparable performance with commercial engines and significant improvement over MySQL performance. It does this by tightly integrating the database engine with an SSD-based virtualized storage layer designed for database workloads. This reduces writes to the storage system, minimizes lock contention, and eliminates delays created by database process threads.

Our tests with SysBench on r5.16xlarge instances show that Amazon Aurora delivers close to 800,000 reads per second and 200,000 writes per second, five times higher than MySQL running the same benchmark on the same hardware.

One area where Amazon Aurora significantly improves upon traditional MySQL is highly concurrent workloads. In order to maximize your workload's throughput on Amazon Aurora, we recommend architecting your applications to drive a large number of concurrent queries.

Sharding and read replica considerations

If your current database is sharded across multiple nodes, you may have an opportunity to combine these shards into a single Aurora database during migration. A single Amazon Aurora instance can scale up to 128 TB, supports thousands of tables, and supports a significantly higher number of reads and writes than a standard MySQL database.

If your application is read/write heavy, consider using Aurora read replicas for offloading read-only workload from the primary database node. Doing this can improve concurrency of your primary database for writes and will improve overall read and write performance. Using read replicas can

also lower your costs in a Multi-AZ configuration since you may be able to use smaller instances for your primary instance while adding failover capabilities in your database cluster. Aurora read replicas offer near-zero replication lag and you can create up to 15 read replicas.

Reliability considerations

An important consideration with databases is high availability and disaster recovery. Determine the recovery time objective (RTO) and recovery point objective (RPO) requirements of your application. With Amazon Aurora, you can significantly improve both these factors.

Amazon Aurora reduces database restart times to less than 60 seconds in most database crash scenarios. Aurora also moves the buffer cache out of the database process and makes it available immediately at restart time. In rare scenarios of hardware and Availability Zone failures, recovery is automatically handled by the database platform.

Aurora is designed to provide you zero RPO recovery within an AWS Region, which is a major improvement over on-premises database systems. Aurora maintains six copies of your data across three Availability Zones and automatically attempts to recover your database in a healthy AZ with no data loss. In the unlikely event that your data is unavailable within Amazon Aurora storage, you can restore from a DB snapshot or perform a point-in-time restore operation to a new instance.

For cross-Region DR, Amazon Aurora also offers a global database feature, designed for globally distributed transactions applications, allowing a single Amazon Aurora database to span multiple AWS Regions. Aurora uses storage-based replication to replicate your data to other Regions with typical latency of less than one second and without impacting database performance. This enables fast local reads with low latency in each Region, and provides disaster recovery from Region-wide outages. You can promote the secondary AWS Region for read-write workloads in case of an outage or disaster in less than one minute.

You also have the option to create an Aurora Read Replica of an Aurora MySQL DB cluster in a different AWS Region, by using MySQL binary log (binlog) replication.

Each cluster can have up to five Read Replicas created this way, each in a different Region.

Cost and licensing considerations

Owning and running databases come with associated costs. Before planning a database migration, an analysis of the total cost of ownership (TCO) of the new database platform is imperative. Migration to a new database platform should ideally lower the total cost of ownership

while providing your applications with similar or better features. If you are running an open-source database engine (MySQL, Postgres), your costs are largely related to hardware, server management, and database management activities. However, if you are running a commercial database engine (Oracle, SQL Server, DB2, and so on), a significant portion of your cost is database licensing.

Since Aurora is available at one-tenth of the cost of commercial engines, many applications moving to Aurora are able to significantly reduce their TCO. Even if you are running on an open-source engine like MySQL or Postgres, with Aurora's high performance and dual purpose read replicas, you can realize meaningful savings by moving to Amazon Aurora. Refer to the Amazon Aurora Pricing page for more information.

Other migration considerations

Once you have considered application suitability, performance, TCO, and reliability factors, you should think about what it would take to migrate to the new platform.

Estimate code change effort

It is important to estimate the amount of code and schema changes that you need to perform while migrating your database to Amazon Aurora. When migrating from MySQL-compatible databases, negligible code changes are required. However, when migrating from non-MySQL engines, you may be required to make schema and code changes. The AWS Schema Conversion Tool can help to estimate that effort (refer to the the section called "Schema migration using the AWS Schema Conversion Tool" section in this document).

Application availability during migration

You have options of migrating to Amazon Aurora by taking a predictable downtime approach with your application or by taking a near-zero downtime approach. The approach you choose depends on the size of your database and the availability requirements of your applications. Whatever the case, it's a good idea to consider the impact of the migration process on your application and business before starting with a database migration. The next few sections explain both approaches in detail.

Modify connection string during migration

You need a way to point the applications to your new database. One option is to modify the connection strings for all of the applications. Another common option is to use DNS. In this case,

you don't use the actual host name of your database instance in your connection string. Instead, consider creating a canonical name (CNAME) record that points to the host name of your database instance. Doing this allows you to change the endpoint to which your application points in a single location rather than tracking and modifying multiple connection string settings. If you choose to use this pattern, be sure to pay close attention to the time to live (TTL) setting for your CNAME record. If this value is set too high, then the host name pointed to by this CNAME might be cached longer than desired. If this value is set too low, additional overhead might be placed on your client applications by having to resolve this CNAME repeatedly. Though use cases differ, a TTL of five seconds is usually a good place to start.

Planning your database migration process

The previous section discussed some of the key considerations to take into account while migrating databases to Amazon Aurora. Once you have determined that Aurora is the right fit for your application, the next step is to decide on a preliminary migration approach and create a database migration plan.

Homogeneous migration

If your source database is a MySQL 5.6 or 5.7 compliant database (MySQL, MariaDB, Percona, and so on.), then migration to Aurora is quite straightforward.

Homogeneous migration with downtime

If your application can accommodate a predictable length of downtime during off-peak hours, migration with the downtime is the simplest option and is a highly recommended approach. Most database migration projects fall into this category as most applications already have a well-defined maintenance window. You have the following options to migrate your database with downtime.

- **RDS snapshot migration** — If your source database is running on Amazon RDS MySQL 5.6 or 5.7, you can simply migrate a snapshot of that database to Amazon Aurora. For migrations with downtime, you either have to stop your application or stop writing to the database while snapshot and migration is in progress. The time to migrate primarily depends upon the size of the database and can be determined ahead of the production migration by running a test migration. Snapshot migration option is explained in the *RDS snapshot migration* section of this document.
- **Migration using native MySQL tools** — You may use native MySQL tools to migrate your data and schema to Aurora. This is a great option when you need more control over the database migration process, you are more comfortable using native MySQL tools, and other migration methods are not performing as well for your use case. You can create a dump of your data using the mysqldump utility, and then import that data into an existing Amazon Aurora MySQL DB cluster. For more information, refer to Migrating from MySQL to Amazon Aurora by using mysqldump. You can copy the full and incremental backup files from your database to an Amazon S3 bucket, and then restore an Amazon Aurora MySQL DB cluster from those files. This option can be considerably faster than migrating data using mysqldump. For more information, refer to Migrating data from MySQL by using an Amazon S3 bucket.

- **Migration using AWS Database Migration Service (AWS DMS)** — One-time migration using AWS DMS is another tool for moving your source database to Amazon Aurora. Before you can use AWS DMS to move the data, you need to copy the database schema from source to target using native MySQL tools. For the step-by-step process, refer to the *Migrating data* section of this document. Using AWS DMS is a great option when you don't have experience using native MySQL tools.

Homogeneous migration with near-zero downtime

In some scenarios you might want to migrate your database to Aurora with minimal downtime. Here are two examples:

- When your database is relatively large and the migration time using downtime options is longer than your application maintenance window.
- When you want to run source and target databases in parallel for testing purposes.

In such cases, you can replicate changes from your source MySQL database to Aurora in real time using replication. You have a couple of options to choose from:

- **Near-zero downtime migration using MySQL binlog replication** — Amazon Aurora supports traditional MySQL binlog replication. If you are running MySQL database, chances are that you are already familiar with classic binlog replication setup. If that's the case, and you want more control over the migration process, one-time database load using native tools coupled with binlog replication gives you a familiar migration path to Aurora.
- **Near-zero downtime migration using AWS Database Migration Service (AWS DMS)** — In addition to supporting one-time migration, AWS DMS also supports real-time data replication using change data capture (CDC) from source to target. AWS DMS takes care of the complexities related to initial data copy, setting up replication instances, and monitoring replication. After the initial database migration is complete, the target database remains synchronized with the source for as long as you choose. If you are not familiar with binlog replication, AWS DMS is the next best option for homogenous, near-zero downtime migrations to Amazon Aurora. Refer to the the section called "Introduction and general approach to AWS DMS" section of this document.
- **Near-zero downtime migration using Aurora Read Replica** — If your source database is running on Amazon RDS MySQL 5.6 or 5.7, you can migrate from a MySQL DB instance to an Aurora MySQL DB cluster by creating an Aurora read replica of your source MySQL DB instance. When the replica lag between the MySQL DB instance and the Aurora Read Replica is zero, you can

direct your client applications to the Aurora read replica. This migration option is explained in the *Migration using Aurora Read Replica* section of this document.

Heterogeneous migration

If you are looking to migrate a non-MySQL-compliant database (Oracle, SQL Server, PostgresSQL, and so on) to Amazon Aurora, several options can help you accomplish this migration quickly and easily.

Schema migration

Schema migration from a non-MySQL-compliant database to Amazon Aurora can be achieved using the AWS Schema Conversion Tool. This tool is a desktop application that helps you convert your database schema from an Oracle, Microsoft SQL Server, or PostgreSQL database to an Amazon RDS MySQL DB instance or an Amazon Aurora DB cluster. In cases where the schema from your source database cannot be automatically and completely converted, the AWS Schema Conversion Tool provides guidance on how you can create the equivalent schema in your target Amazon RDS database. For details, refer to the *Migrating the database schema* section of this document.

Data migration

While supporting homogenous migrations with near-zero downtime, AWS Database Migration Service (AWS DMS) also supports continuous replication across heterogeneous databases and is a preferred option to move your source database to your target database, for both migrations with downtime and migrations with near-zero downtime. Once the migration has started, AWS DMS manages all the complexities of the migration process like data type transformation, compression, and parallel transfer (for faster data transfer) while ensuring that data changes to the source database that occur during the migration process are automatically replicated to the target.

Besides using AWS DMS, you can use various third-party tools like Qlik Replicate, Tungsten Replicator, Oracle Golden Gate, etc. to migrate your data to Amazon Aurora. Whatever tool you choose, take performance and licensing costs into consideration before finalizing your toolset for migration.

Migrating large databases to Amazon Aurora

Migration of large datasets presents unique challenges in every database migration project. Many successful large database migration projects use a combination of the following strategies:

- **Migration with continuous replication** — Large databases typically have extended downtime requirements while moving data from source to target. To reduce the downtime, you can first load baseline data from source to target and then enable replication (using MySQL native tools, AWS DMS, or third-party tools) for changes to catch up.
- **Copy static tables first** — If your database relies on large static tables with reference data, you may migrate these large tables to the target database before migrating your active dataset. You can use AWS DMS to copy tables selectively or export and import these tables manually.
- **Multiphase migration** — Migration of large database with thousands of tables can be broken down into multiple phases. For example, you may move a set of tables with no cross joins queries every weekend until the source database is fully migrated to the target database. Note that in order to achieve this, you need to make changes in your application to connect to two databases simultaneously while your dataset is on two distinct nodes. Although this is not a common migration pattern, this is an option nonetheless.
- **Database cleanup** — Many large databases contain data and tables that remain unused. In many cases, developers and DBAs keep backup copies of tables in the same database, or they just simply forget to drop unused tables. Whatever the reason, a database migration project provides an opportunity to clean up the existing database before the migration. If some tables are not being used, you might either drop them or archive them to another database. You might also delete old data from large tables or archive that data to flat files.

Partition and shard consolidation on Amazon Aurora

If you are running multiple shards or functional partitions of your database to achieve high performance, you have an opportunity to consolidate these partitions or shards on a single Aurora database. A single Amazon Aurora instance can scale up to 128 TB, supports thousands of tables, and supports a significantly higher number of reads and writes than a standard MySQL database. Consolidating these partitions on a single Aurora instance not only reduces the total cost of ownership and simplify database management, but it also significantly improves performance of cross-partition queries.

- **Functional partitions** — Functional partitioning means dedicating different nodes to different tasks. For example, in an ecommerce application, you might have one database node serving product catalog data, and another database node capturing and processing orders. As a result, these partitions usually have distinct, nonoverlapping schemas.
- **Consolidation strategy** — Migrate each functional partition as a distinct schema to your target Aurora instance. If your source database is MySQL compliant, use native MySQL tools to migrate the schema and then use AWS DMS to migrate the data, either one time or continuously using replication. If your source database is non-MySQL complaint, use AWS Schema Conversion Tool to migrate the schemas to Aurora and use AWS DMS for one-time load or continuous replication.
- **Data shards** — If you have the same schema with distinct sets of data across multiple nodes, you are leveraging database sharding. For example, a high-traffic blogging service may shard user activity and data across multiple database shards while keeping the same table schema.
- **Consolidation strategy** — Since all shards share the same database schema, you only need to create the target schema once. If you are using a MySQL- compliant database, use native tools to migrate the database schema to Aurora. If you are using a non-MySQL database, use AWS Schema Conversion Tool to migrate the database schema to Aurora. Once the database schema has been migrated, it is best to stop writes to the database shards and use native tools or an AWS DMS one-time data load to migrate an individual shard to Aurora. If writes to the application cannot be stopped for an extended period, you might still use AWS DMS with replication but only after proper planning and testing.

Migration options at a glance

Table 1 — Migration options

Source database type	Migration with downtime	Near-zero downtime migration
Amazon RDS MySQL	**Option 1**: RDS snapshot migration **Option 2**: Manual migration using native tools*	**Option 1**: Migration using native tools + binlog replicati on **Option 2**: Migrate using Aurora Read Replica

Source database type	Migration with downtime	Near-zero downtime migration
	Option 3: Schema migration using native tools and data load using AWS DMS	**Option 3**: Schema migration using native tools + AWS DMS for data movement
MySQL Amazon EC2 or on-premises	**Option 1**: Migration using native tools **Option 2**: Schema migration with native tools + AWS DMS for data load	**Option 1**: Migration using native tools + binlog replicati on **Option 2**: Schema migration using native tools + AWS DMS to move data
Oracle/SQL server	**Option 1**: AWS Schema Conversion Tool + AWS DMS (recommended) **Option 2**: Manual or third-party tool for schema conversion + manual or third-party data load in target	**Option 1**: AWS Schema Conversion Tool + AWS DMS (recommended) **Option 2**: Manual or third-par ty tool for schema conversion + manual or third-party data load in target + third-party tool for replication.
Other non-MySQL databases	**Option**: Manual or third-par ty tool for schema conversion + manual or third-party data load in target	**Option**: Manual or third-par ty tool for schema conversion + manual or third-party data load in target + third-party tool for replication (GoldenGa te and so on.)

*MySQL Native tools: mysqldump, SELECT INTO OUTFILE, third-party tools like `mydumper/myloader`.

RDS snapshot migration

To use RDS snapshot migration to move to Aurora, your MySQL database must be running on Amazon RDS MySQL 5.6 or 5.7, and you must make an RDS snapshot of the database. This migration method does not work with on-premises databases or databases running on Amazon Elastic Compute Cloud (Amazon EC2). Also, if you are running your Amazon RDS MySQL database on a version earlier than 5.6, you would need to upgrade it to 5.6 as a prerequisite.

The biggest advantage to this migration method is that it is the simplest and requires the fewest number of steps. In particular, it migrates over all schema objects, secondary indexes, and stored procedures along with all of the database data.

During snapshot migration without binlog replication, your source database must either be offline or in a read-only mode (so that no changes are being made to the source database during migration). To estimate downtime, you can simply use the existing snapshot of your database to do a test migration. If the migration time fits within your downtime requirements, then this may be the best method for you. Note that in some cases, migration using AWS DMS or native migration tools can be faster than using snapshot migration.

If you can't tolerate extended downtime, you can achieve near-zero downtime by creating an Aurora Read Replica from a source RDS MySQL. This migration option is explained in the *Migrating using Aurora Read Replica* section in this document.

You can migrate either a manual or an automated DB snapshot. The general steps you must take are as follows:

1. Determine the amount of space that is required to migrate your Amazon RDS MySQL instance to an Aurora DB cluster. For more information, see the next section.
2. Use the Amazon RDS console to create the snapshot in the Region where the Amazon RDS MySQL instance is located.
3. Use the **Migrate Database** feature on the console to create an Amazon Aurora DB cluster that will be populated using the DB snapshot from the original DB instance of MySQL.

> **Note**
>
> Some MyISAM tables might not convert without errors and may require manual changes. For instance, the InnoDB engine does not permit an autoincrement field to be part of a composite key. Also, spatial indexes are not currently supported.

Estimating space requirements for snapshot migration

When you migrate a snapshot of a MySQL DB instance to an Aurora DB cluster, Aurora uses an Amazon Elastic Block Store (Amazon EBS) volume to format the data from the snapshot before migrating it. There are some cases where additional space is needed to format the data for migration. The two features that can potentially cause space issues during migration are MyISAM tables and using the `ROW_FORMAT=COMPRESSED` option. If you are not using either of these features in your source database, then you can skip this section because you should not have space issues. During migration, MyISAM tables are converted to InnoDB and any compressed tables are uncompressed. Consequently, there must be adequate room for the additional copies of any such tables.

The size of the migration volume is based on the allocated size of the source MySQL database that the snapshot was made from. Therefore, if you have MyISAM or compressed tables that make up a small percentage of the overall database size and there is available space in the original database, then migration should succeed without encountering any space issues. However, if the original database would not have enough room to store a copy of converted MyISAM tables as well as another (uncompressed) copy of compressed tables, then the migration volume will not be big enough. In this situation, you would need to modify the source Amazon RDS MySQL database to increase the database size allocation to make room for the additional copies of these tables, take a new snapshot of the database, and then migrate the new snapshot.

When migrating data into your DB cluster, observe the following guidelines and limitations:

- Although Amazon Aurora supports up to 128 TB of storage, the process of migrating a snapshot into an Aurora DB cluster is limited by the size of the Amazon EBS volume of the snapshot, and therefore is limited to a maximum size of 16 TB.
- Non-MyISAM tables in the source database can be up to 16 TB in size. However, due to additional space requirements during conversion, make sure that none of the MyISAM and compressed tables being migrated from your MySQL DB instance exceed 8 TB in size.

You might want to modify your database schema (convert MyISAM tables to InnoDB and remove `ROW_FORMAT=COMPRESSED`) prior to migrating it into Amazon Aurora. This can be helpful in the following cases:

- You want to speed up the migration process.
- You are unsure of how much space you need to provision.
- You have attempted to migrate your data and the migration has failed due to a lack of provisioned space.

Make sure that you are not making these changes in your production Amazon RDS MySQL database but rather on a database instance that was restored from your production snapshot. For more details, refer to the Amazon Relational Database Service User Guide.

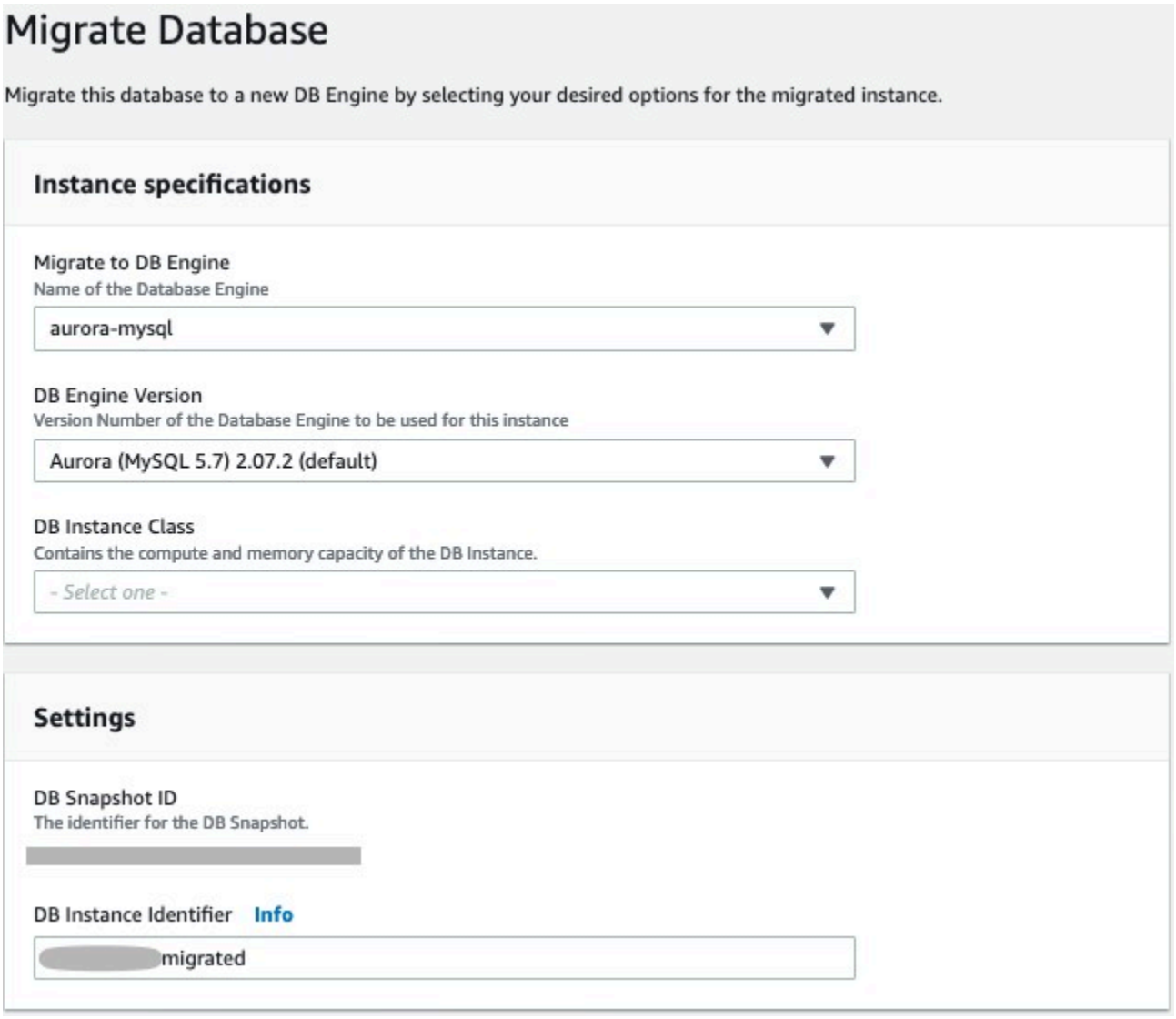
Migrating a DB snapshot using the console

You can migrate a DB snapshot of an Amazon RDS MySQL DB instance to create an Aurora DB cluster. The new DB cluster is populated with the data from the original Amazon RDS MySQL DB instance. The DB snapshot must have been made from an RDS DB instance running MySQL 5.6 or 5.7. For information about creating a DB snapshot, refer to Creating a DB snapshot in the Amazon RDS User Guide.

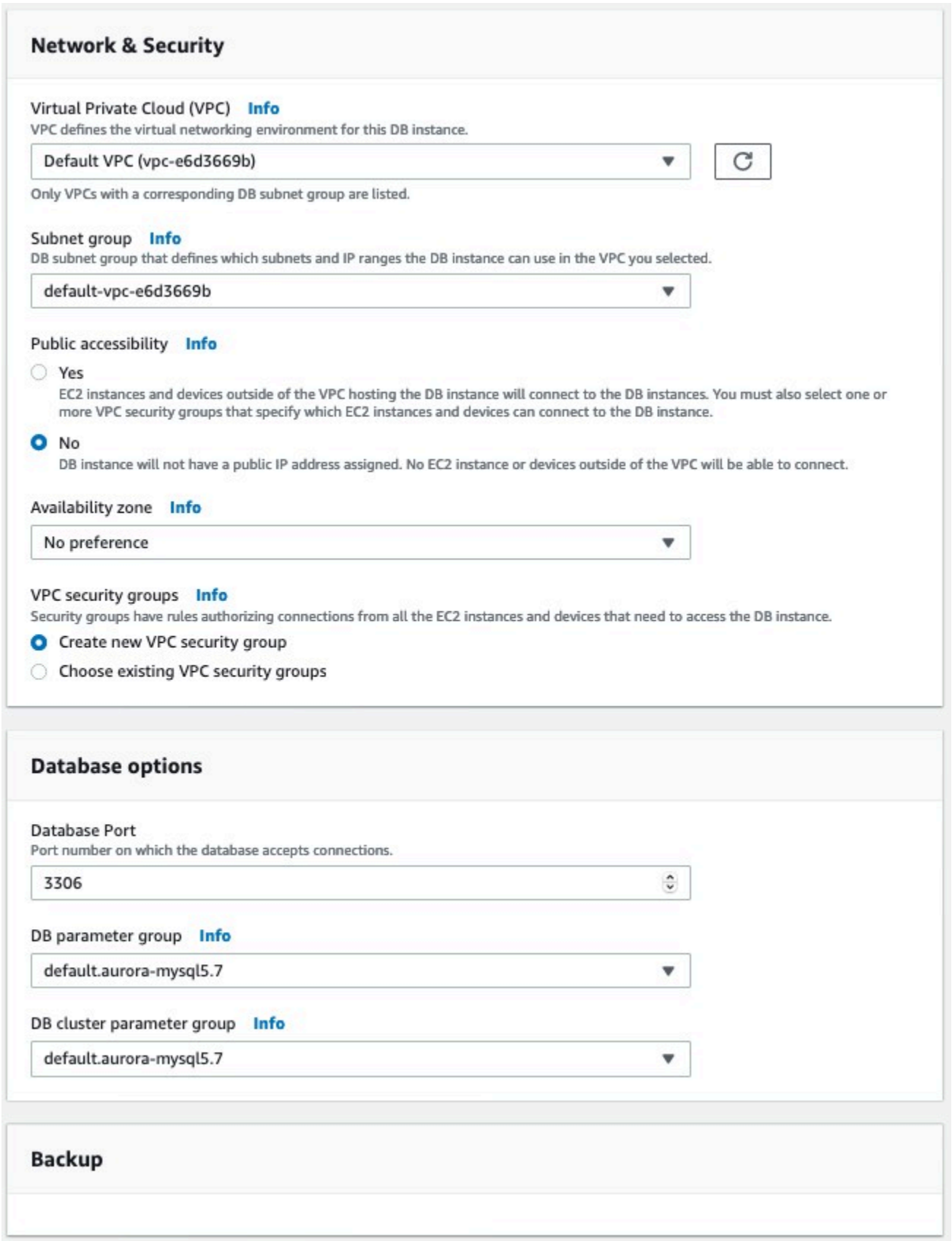
If the DB snapshot is not in the Region where you want to locate your Aurora DB cluster, use the Amazon RDS console to copy the DB snapshot to that Region. For information about copying a DB snapshot, refer to Copying a snapshot in Amazon RDS User Guide.

To migrate a MySQL DB snapshot using the AWS Management Console, do the following:

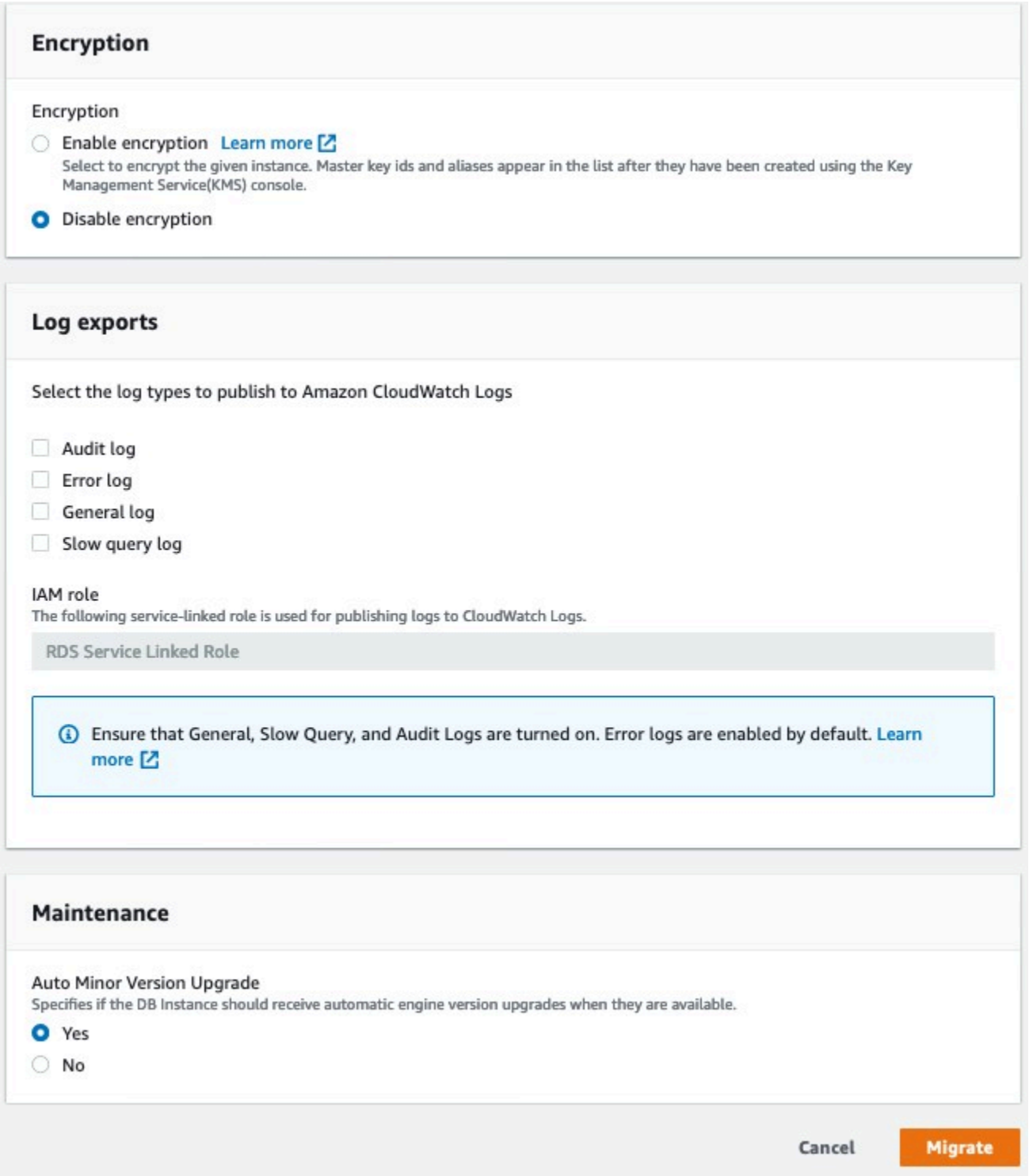
1. Sign in to the AWS Management Console and open the Amazon RDS console (sign in required).
2. Choose **Snapshots**.
3. On the **Snapshots** page, choose the Amazon RDS MySQL snapshot that you want to migrate into an Aurora DB cluster.
4. Choose **Migrate Database.**
5. On the **Migrate Database** page, specify the values that match your environment and processing requirements as shown in the following illustration. For descriptions of these options, refer to Migrating an RDS for MySQL snapshot to Aurora in the Amazon Aurora User Guide.



Amazon RDS console: snapshot migration screens


Network & Security
Virtual Private Cloud (VPC) Info
VPC defines the virtual networking environment for this DB instance.
Default VPC (vpc-e6d3669b)
Only VPCs with a corresponding DB subnet group are listed.
Subnet group Info
DB subnet group that defines which subnets and IP ranges the DB instance can use in the VPC you selected.
default-vpc-e6d3669b
Public accessibility Info
Yes
EC2 instances and devices outside of the VPC hosting the DB instance will connect to the DB instances. You must also select one or more VPC security groups that specify which EC2 instances and devices can connect to the DB instance.
No
DB instance will not have a public IP address assigned. No EC2 instance or devices outside of the VPC will be able to connect.
Availability zone Info
No preference
VPC security groups Info
Security groups have rules authorizing connections from all the EC2 instances and devices that need to access the DB instance.
Create new VPC security group
Choose existing VPC security groups
Database options
Database Port
Port number on which the database accepts connections.
3306
DB parameter group Info
default.aurora-mysql5.7
DB cluster parameter group Info
default.aurora-mysql5.7
Backup

Amazon RDS console: snapshot migration screens



Amazon RDS console: snapshot migration screens

6. Choose **Migrate** to migrate your DB snapshot.

In the list of instances, choose the appropriate arrow icon to show the DB cluster details and monitor the progress of the migration. This details panel displays the cluster endpoint used to connect to the primary instance of the DB cluster. For more information on connecting to an Amazon Aurora DB cluster, refer to Connecting to an Amazon Aurora DB cluster in the Amazon Aurora User Guide.

Migration using Aurora Read Replica

Aurora uses MySQL DB engines binary log replication functionality to create a special type of DB cluster called an Aurora read replica for a source MySQL DB instance. Updates made to the source instance are asynchronously replicated to Aurora Read Replica.

To use Aurora Read Replica to migrate from RDS MySQL, your MySQL database must be running on Amazon RDS MySQL 5.6 or 5.7. This migration method does not work with on-premises databases or databases running on Amazon Elastic Compute Cloud (Amazon EC2). Also, if you are running your Amazon RDS MySQL database on a version earlier than 5.6, you would need to upgrade it to 5.6 as a prerequisite.

You can migrate your existing Amazon RDS MySQL databases to Amazon Aurora using Aurora Read Replica. This solution is beneficial since it's completely managed and does not involve manually configuring replication functionality to reduce downtime during migration.

These are the high-level steps to be performed:

1. Begin writes on the source Amazon RDS database (to simulate traffic in the real world).
2. Create an Amazon Aurora read replica from the existing Amazon RDS MySQL instance.
3. Stop writes to the Amazon RDS MySQL instance.
4. Wait for the replication lag between Amazon RDS MySQL and Amazon Aurora read replica to be zero.
5. Promote Amazon Aurora read replica to be a standalone database cluster.
6. Begin writes on Amazon Aurora read replica, **immediately after starting promotion**.

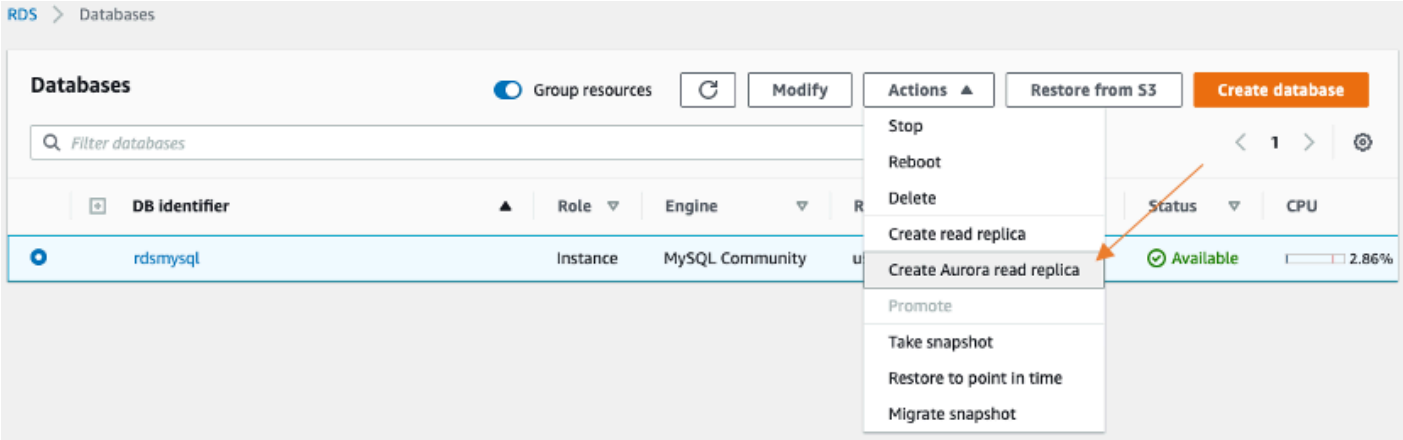
Although the promotion process is fairly quick, it can still add valuable seconds in the downtime window of your application, especially if the recovery time objective (RTO) for cutover is only for a few seconds instead of approximately 30 seconds. The key point in the above process which reduces the downtime window is the ability for the applications to read and write to the Amazon Aurora read replica immediately after the promotion process is started instead of waiting for it to complete.

The only time writes on the Aurora read replica should be on hold is when writing has stopped on source Amazon RDS and the replica lag reaches zero.

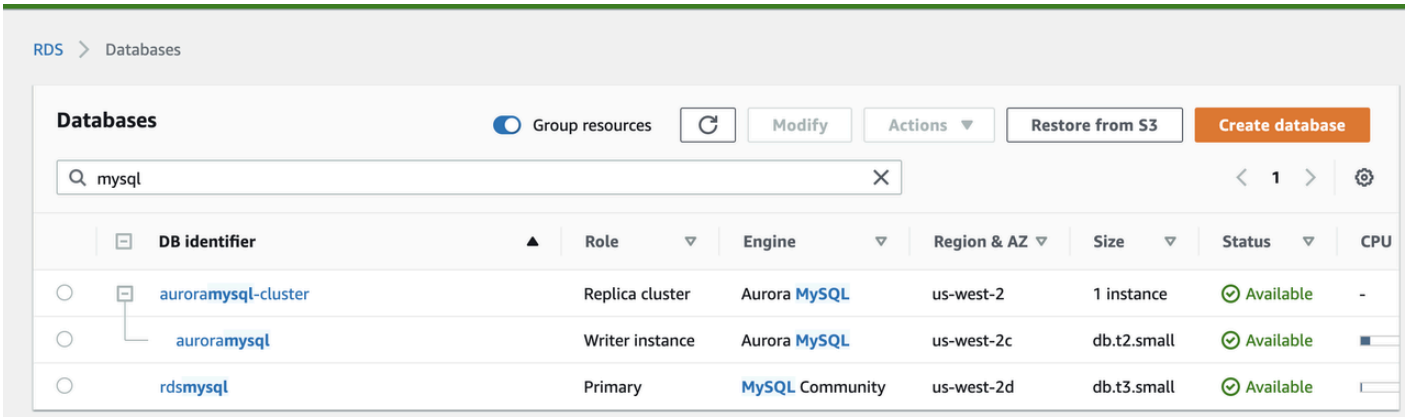
To get started, you must have an existing Amazon RDS MySQL instance.

Create Amazon Aurora read replica from an existing Amazon RDS instance

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the MySQL DB instance that you want to use as the source for your Aurora read replica.
4. For **Actions**, choose **Create Aurora read replica**.



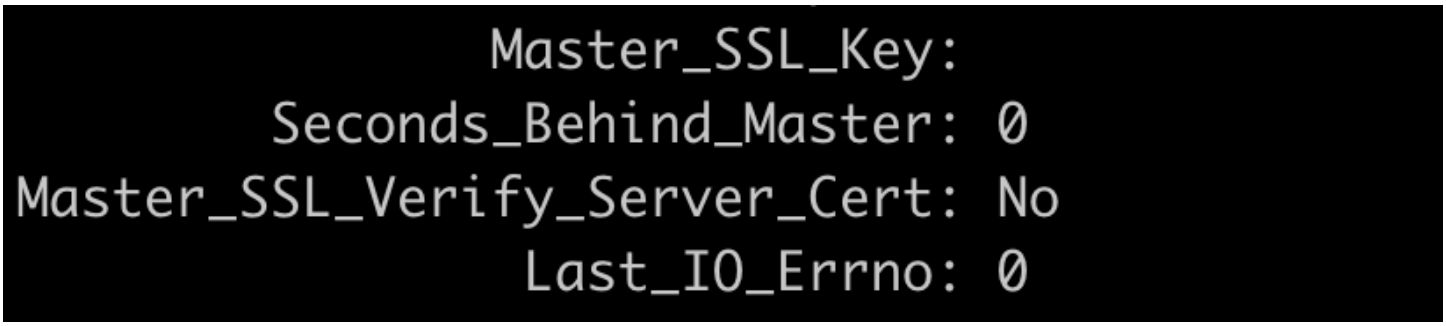
5. Choose the DB cluster specifications as mentioned in Migrating data from a MySQL DB instance to an Amazon Aurora MySQL DB cluster by using an Aurora read replica and click **Create read replica**.
6. After Aurora replica is successfully completed, you should see a replica cluster along with RDS MySQL instance.



Stop writes to the Amazon RDS MySQL instance

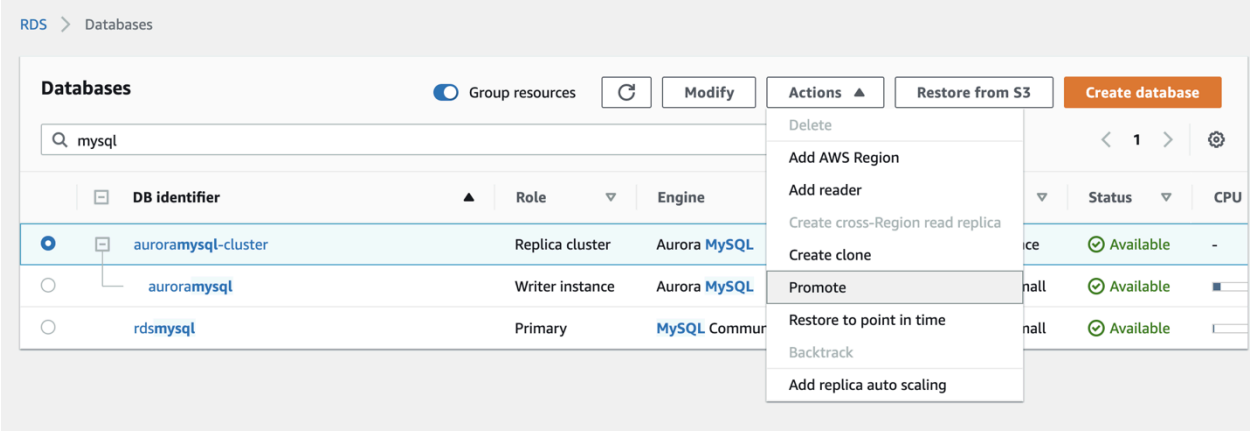
Stop writing to the source RDS database and wait for the replica lag between Amazon RDS MySQL and Amazon Aurora read replica to come down to zero. You can check replica lag by running the `SHOW SLAVE STATUS` command on Aurora read replica and checking the **Seconds behind Master** value.

```
SHOW SLAVE STATUS \G
```



Promote Amazon Aurora read replica to be a standalone database cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the DB cluster for the Aurora read replica.
4. For **Actions**, choose **Promote**.
5. Choose **Promote read replica**.



As an alternative, you can issue the following AWS CLI command instead of using the AWS Management Console:

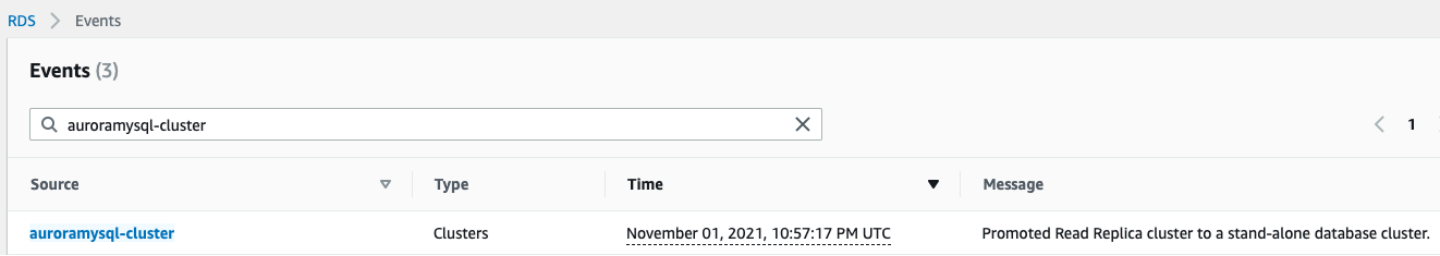
```
aws rds promote-read-replica-db-cluster \
    --db-cluster-identifier myreadreplicacluster
```

Begin writes on Amazon Aurora read replica immediately after starting promotion

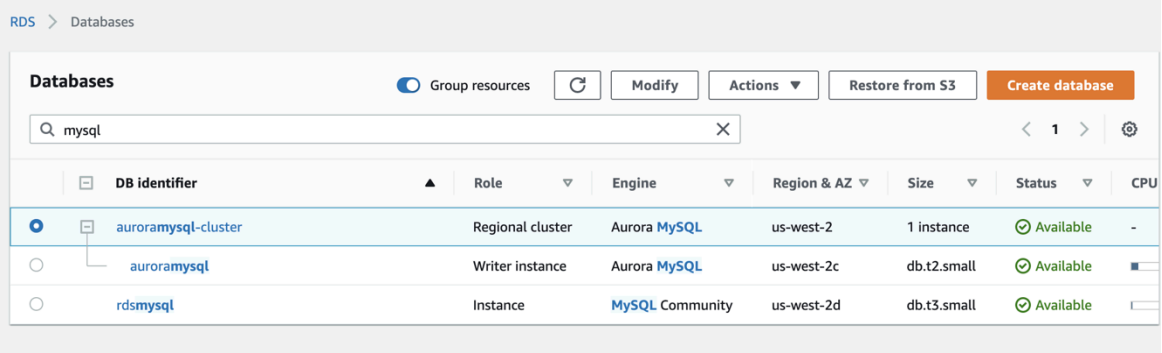
Immediately after starting the promotion process, issue writes to the Amazon Aurora read replica. To validate if your writes are anyway affected during the promotion process, we will start inserting records into Aurora read replica as soon as we start the promotion.

After promotion is complete, you can confirm that the promotion has completed by using the following procedure.

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Events**.
3. On the **Events** page, verify that there is a **Promoted Read Replica cluster to a stand-alone database cluster** event for the cluster that you promoted.



It took around 15 to 20 seconds for the promotion process (replica cluster to change status to regional cluster).



After promotion is complete, the primary MySQL DB instance and the Aurora read replica are unlinked, and you can safely delete the DB instance if you want.

> **Note**
>
> PostgreSQL follows the same process as the one described previously for MySQL for migration.

Important points to consider

- If your application uses a DNS solution like Amazon Route53, consider the DNS TTL (Time-To-Live) of approximately five seconds when switching the Amazon RDS endpoint with Amazon Aurora endpoint.
- Perform this operation during non-peak hours or at another time when writes to the primary DB cluster are minimal.
- You cannot delete the primary MySQL DB instance or unlink the DB Instance and the Aurora read replica during promotion time.
- If you write before replica lag reaches zero, then the replication is at the risk of breaking and you'll have to delete and recreate the Aurora read replica.
- Be prepared for the Amazon Aurora Read Replica to catch up with Amazon RDS when you initially create it. It could take several hours per tebibyte (TiB) of data. The source RDS instance is available throughout the initial load.
- Version compatibility - The RDS for PostgreSQL version must be lower than or equal to a supported Aurora PostgreSQL version in the same major version. For example, you can replicate data between an RDS for PostgreSQL version 11.7 DB instance and an Aurora PostgreSQL version 11.7 or higher 11 version DB cluster, but not an Aurora PostgreSQL version 11.6 DB

cluster. You cannot downgrade your version, so if you running on MySQL 8 in RDS, you can't create an Aurora read replica with MySQL 5.7.

- Rollback option: If you wish to conduct sanity checks on the new primary on Amazon Aurora, you can let the Amazon RDS MySQL instance run after promotion of Amazon Aurora and rollback to Amazon RDS in case of any issues Identified by pointing the application back to the Amazon RDS endpoint. The time required for sanity checks will add to application downtime as once the switchover is done, Amazon Aurora does not replicate back to Amazon RDS in a managed fashion, although this can be done natively.

Aurora read replica vs other methods:

Using Amazon Aurora read replica is great when you are looking to make use of a managed solution while migrating from Amazon RDS (MySQL or PostgreSQL) to Amazon Aurora. This method also applies only in case of homogenous migration of MySQL and PostgreSQL engines. With this method, you can only move an Amazon RDS MySQL to an Amazon Aurora MySQL instance and an Amazon RDS PostgreSQL to an Amazon Aurora PostgreSQL instance, not MySQL or PostgreSQL instances running on EC2. Heterogenous migration across engines is not supported when using an Aurora read-replica method such as Amazon RDS Oracle, SQL Server, MariaDB.

If the above doesn't fit your requirement, you could adopt other options for near-zero downtime migration from your source MySQL to Aurora MySQL platform:

1. **Native tools + binlog replication:** This method is a manual one and applies to homogenous migrations and can be very effective in that scenario. Using native tools such as mysqldump and binary logging for MySQL near-zero downtime migration can be achieved. This method can be used for small-scale migrations when the MySQL database is running on-premises or on MySQL EC2 prior to 5.6 or 5.7 versions.
2. **RDS snapshot + binlog replication:** Instead of using native tools, another method for homogenous migration is to use RDS snapshot of the RDS read replica and create an Aurora database cluster from it. Along with binary logging, this method will help is a near-zero downtime migration as well. This method however is only supported for RDS MySQL 5.6 or 5.7 and migrating only to the same version is supported, not 5.6 to 5.7. This method again requires some amount of manual work by configuring the binary logging between source and target.
3. **Database Migration Service (DMS):** You can use AWS Database Migration Service (AWS DMS) to migrate your data among homogeneous migrations such as MySQL to MySQL. DMS supports both heterogeneous migrations between different platform such as Oracle to MySQL

and homogeneous migrations such as MySQL to MySQL. AWS DMS can do a one-time data migration, or it can do a continuous replication of the data for near-zero downtime migration using its Change Data Capture (CDC) feature Please refer to migrating from MySQL to Amazon Aurora. AWS DMS might also be advantageous if your migration project requires advanced data transformations such as remapping schema or table names, advanced data filtering, migrating and replicating multiple database servers into a single Aurora DB cluster.

Review the limitations for AWS DMS before using this method.

> **Note**
>
> Since AWS DMS CDC uses plain SQL statements from binlog to apply the changes in target database, it might be slower and more resource-intensive than native primary/replica binary log replication in MySQL. Hence, it is recommended to use the native self-managed or managed features for homogenous migrations and DMS for heterogenous migrations.

Migrating the database schema

RDS DB snapshot migration migrates both the full schema and data to the new Aurora instance. However, if your source database location or application uptime requirements do not allow the use of RDS snapshot migration, then you first need to migrate the database schema from the source database to the target database before you can move the actual data. A database schema is a skeleton structure that represents the logical view of the entire database, and typically includes the following:

- **Database storage objects** — Tables, columns, constraints, indexes, sequences, user-defined types, and data types
- **Database code objects** — Functions, procedures, packages, triggers, views, materialized views, events, SQL scalar functions, SQL inline functions, SQL table functions, attributes, variables, constants, table types, public types, private types, cursors, exceptions, parameters, and other objects

In most situations, the database schema remains relatively static, and therefore you don't need downtime during the database schema migration step. The schema from your source database can be extracted while your source database is up and running without affecting the performance. If your application or developers do make frequent changes to the database schema, make sure that these changes are either paused while the migration is in process, or are accounted for during the schema migration process.

Depending on the type of your source database, you can use the techniques discussed in the next sections to migrate the database schema. As a prerequisite to schema migration, you must have a target Aurora database created and available.

Homogeneous schema migration

If your source database is MySQL 5.6-compliant and is running on Amazon RDS, Amazon EC2, or outside AWS, you can use native MySQL tools to export and import the schema.

- **Exporting database schema** — You can use the `mysqldump` client utility to export the database schema. To run this utility, you need to connect to your source database and redirect the output of mysqldump command to a flat file. The –no-data option ensures that only database schema is exported without any actual table data. For the complete mysqldump command reference, refer to `mysqldump` — A Database Backup Program.

```
mysqldump –u source_db_username –p --no-data --routines --triggers

–databases source_db_name > DBSchema.sql
```

- **Importing database schema into Aurora** — To import the schema to your Aurora instance, connect to your Aurora database from a MySQL command line client (or a corresponding Windows client) and direct the contents of the export file into MySQL.

```
mysql –h aurora-cluster-endpoint -u username -p < DBSchema.sql
```

Note the following:

- If your source database contains stored procedures, triggers, and views, you need to remove `DEFINER` syntax from your dump file. A simple Perl command to do that is given below. Doing this creates all triggers, views, and stored procedures with the current connected user as `DEFINER`. Be sure to evaluate any security implications this might have.

```
$perl -pe 's/\sDEFINER=`[^`]+`@`[^`]+`//' < DBSchema.sql >

DBSchemaWithoutDEFINER.sql
```

- Amazon Aurora supports InnoDB tables only. If you have `MyISAM` tables in your source database, Aurora automatically changes the engine to InnoDB when the `CREATE TABLE` command is run.
- Amazon Aurora does not support compressed tables (that is, tables created with `ROW_FORMAT=COMPRESSED`). If you have compressed tables in your source database, Aurora automatically changes `ROW_FORMAT` to `COMPACT` when the `CREATE TABLE` command is run.

Once you have successfully imported the schema into Amazon Aurora from your MySQL 5.6-compliant source database, the next step is to copy the actual data from the source to the target. For more information, refer to the *Introduction and general approach to AWS DMS* section of this document.

Heterogeneous schema migration

If your source database isn't MySQL compatible, you must convert your schema to a format compatible with Amazon Aurora. Schema conversion from one database engine to another database engine is a nontrivial task and may involve rewriting certain parts of your database and

application code. You have two main options for converting and migrating your schema to Amazon Aurora:

- **AWS Schema Conversion Tool** — The AWS Schema Conversion Tool makes heterogeneous database migrations easy by automatically converting the source database schema and a majority of the custom code, including views, stored procedures, and functions, to a format compatible with the target database. Any code that cannot be automatically converted is clearly marked so that it can be manually converted. You can use this tool to convert your source databases running on either Oracle or Microsoft SQL Server to an Amazon Aurora, MySQL, or PostgreSQL target database in either Amazon RDS or Amazon EC2. Using the AWS Schema Conversion Tool to convert your Oracle, SQL Server, or PostgreSQL schema to an Aurora-compatible format is the preferred method.
- **Manual schema migration and third-party tools** — If your source database is not Oracle, SQL Server, or PostgreSQL, you can either manually migrate your source database schema to Aurora or use third-party tools to migrate schema to a format that is compatible with MySQL 5.6. Manual schema migration can be a fairly involved process depending on the size and complexity of your source schema. In most cases, however, manual schema conversion is worth the effort given the cost savings, performance benefits, and other improvements that come with Amazon Aurora.

Schema migration using the AWS Schema Conversion Tool

The AWS Schema Conversion Tool provides a project-based user interface to automatically convert the database schema of your source database into a format that is compatible with Amazon Aurora. It is highly recommended that you use AWS Schema Conversion Tool to evaluate the database migration effort and for pilot migration before the actual production migration.

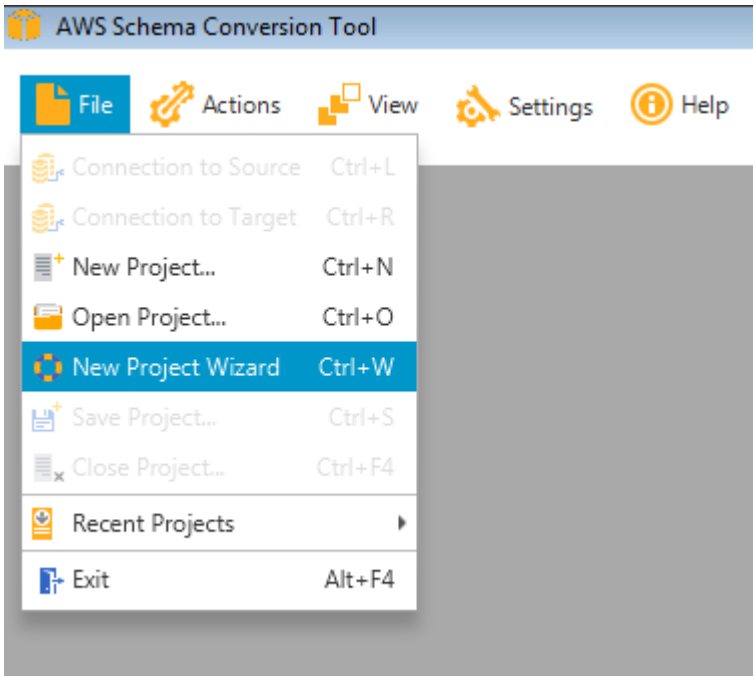
The following description walks you through the high-level steps of using AWS the Schema Conversion Tool. For detailed instructions, refer to the AWS Schema Conversion Tool User Guide.

1. First, install the tool. The AWS Schema Conversion Tool is available for the Microsoft Windows, macOS X, Ubuntu Linux, and Fedora Linux.

   Detailed download and installation instructions can be found in the Installing, verifying, and updating AWS SCT section of the user guide. Where you install AWS Schema Conversion Tool is important. The tool needs to connect to both source and target databases directly in order

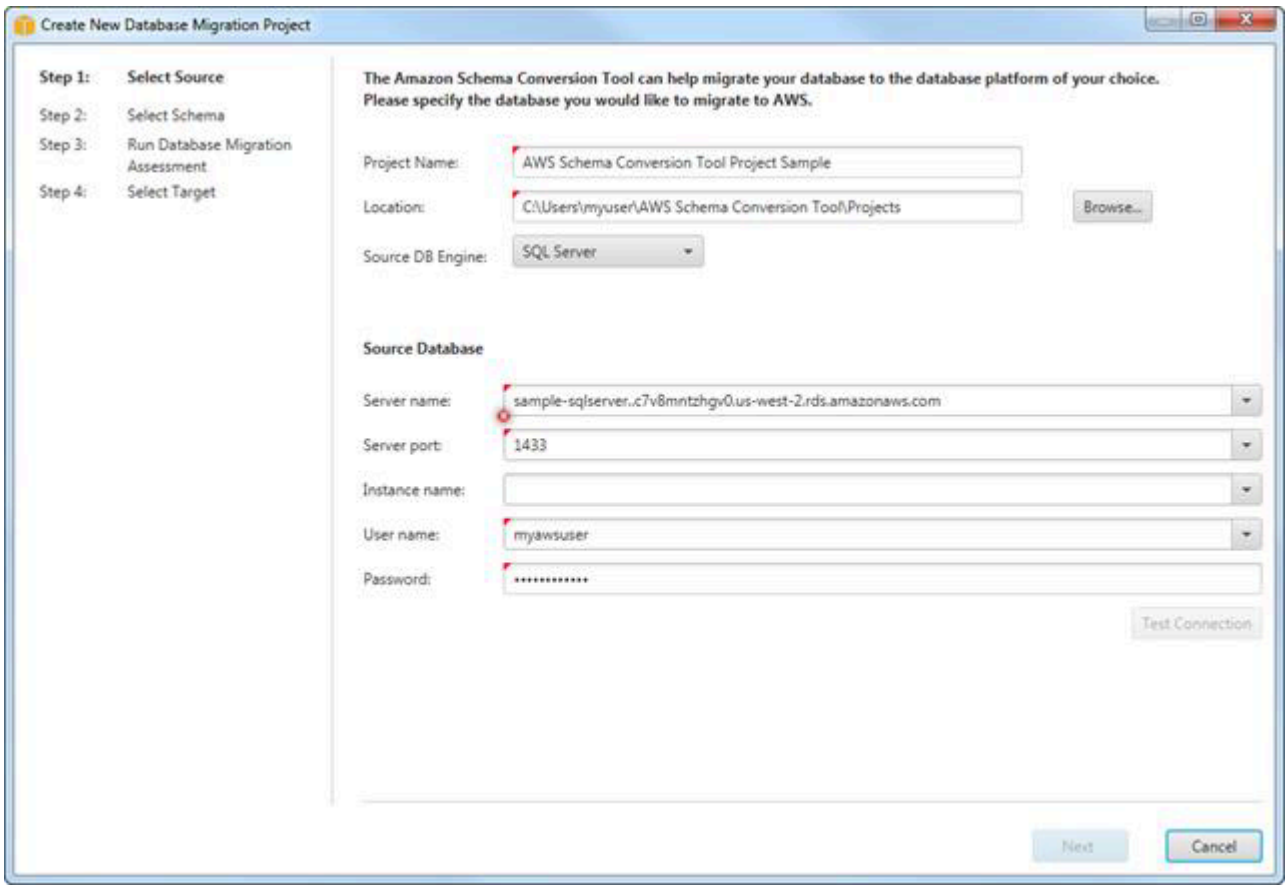
to convert and apply schema. Make sure that the desktop where you install AWS Schema Conversion Tool has network connectivity with the source and target databases.

2. Install JDBC drivers. The AWS Schema Conversion Tool uses JDBC drivers to connect to the source and target databases. In order to use this tool, you must download these JDBC drivers to your local desktop. For instructions for driver download, refer to Installing the required database drivers in the AWS Schema Conversion Tool User Guide. Also, check the AWS forum for AWS Schema Conversion Tool for instructions on setting up JDBC drivers for different database engines.
3. Create a target database. Create an Amazon Aurora target database. For instructions on creating an Amazon Aurora database, see Creating an Amazon Aurora DB Cluster in the Amazon RDS User Guide.
4. Open the AWS Schema Conversion Tool and start the New Project Wizard.



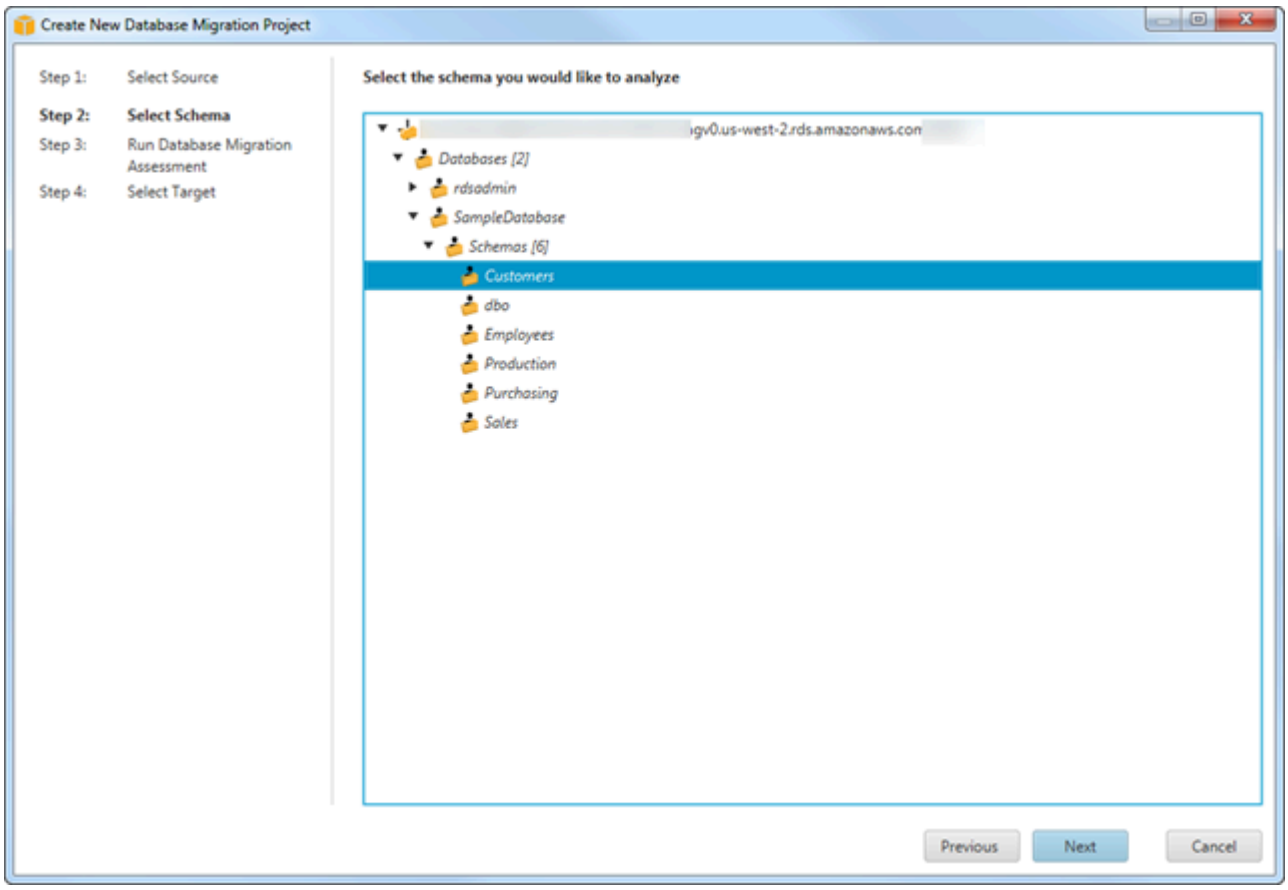
Create a new AWS Schema Conversion Tool project

5. Configure the source database and test connectivity between AWS Schema Conversion Tool and the source database. Your source database must be reachable from your desktop for this to work, so make sure that you have the appropriate network and firewall settings in place.



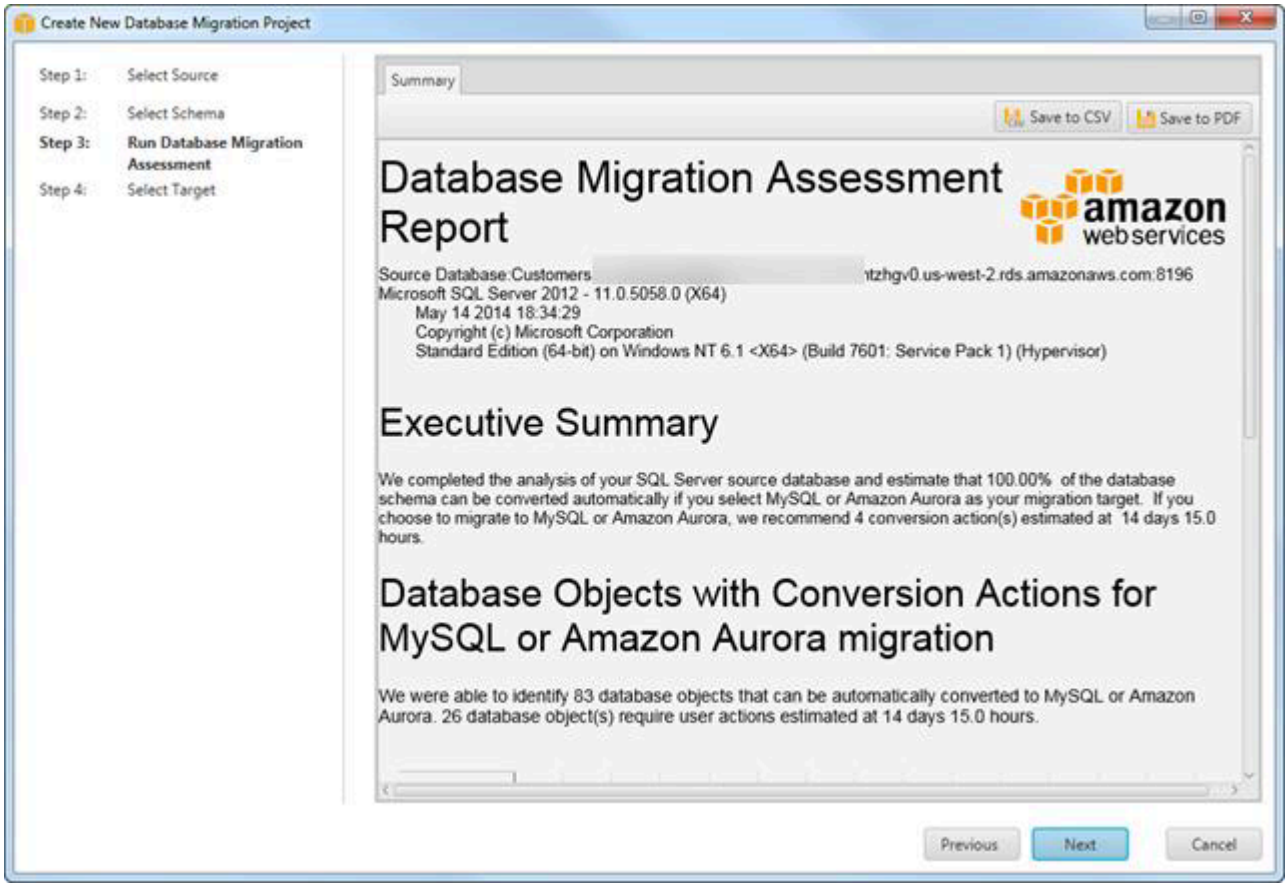
Create New Database Migration Project wizard

6. In the next screen, select the schema of your source database that you want to convert to Amazon Aurora.



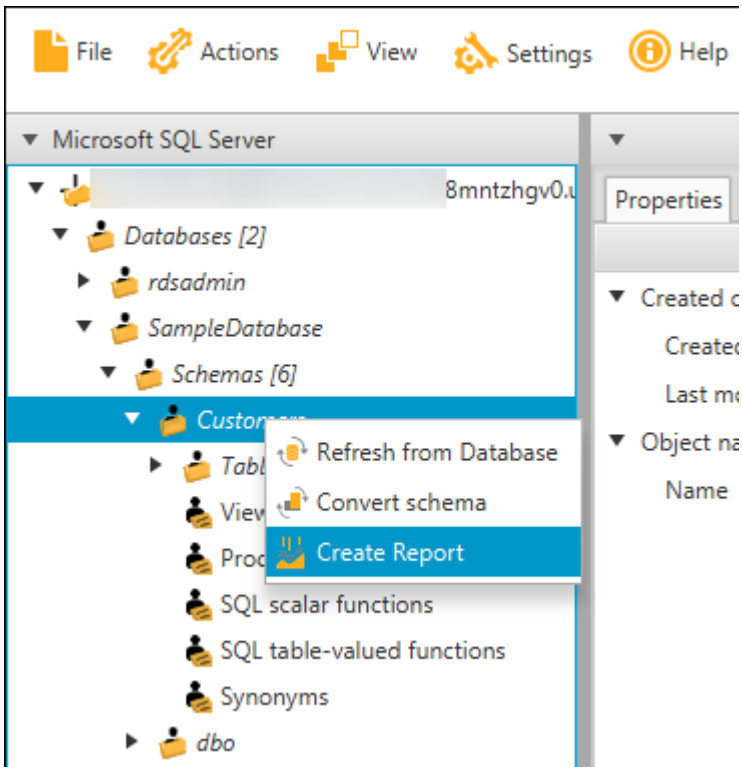
Select Schema *step of the migration wizard*

7. Run the database migration assessment report. This report provides important information regarding the conversion of the schema from your source database to your target Amazon Aurora instance. It summarizes all of the schema conversion tasks and details the action items for parts of the schema that cannot be automatically converted to Aurora. The report also includes estimates of the amount of effort that it will take to write the equivalent code in your target database that could not be automatically converted.
8. Choose **Next** to configure the target database. You can view this migration report again later.



The Create New Database Migration report

9. Configure the target Amazon Aurora database and test connectivity between the AWS Schema Conversion Tool and the source database. Your target database must be reachable from your desktop for this to work, so make sure that you have appropriate network and firewall settings in place.
10. Choose **Finish** to go to the project window.
11. Once you are at the project window, you have already established a connection to the source and target database and are now ready to evaluate the detailed assessment report and migrate the schema.
12. In the left panel that displays the schema from your source database, choose a schema object to create an assessment report for. Right-click the object and choose **Create Report**.

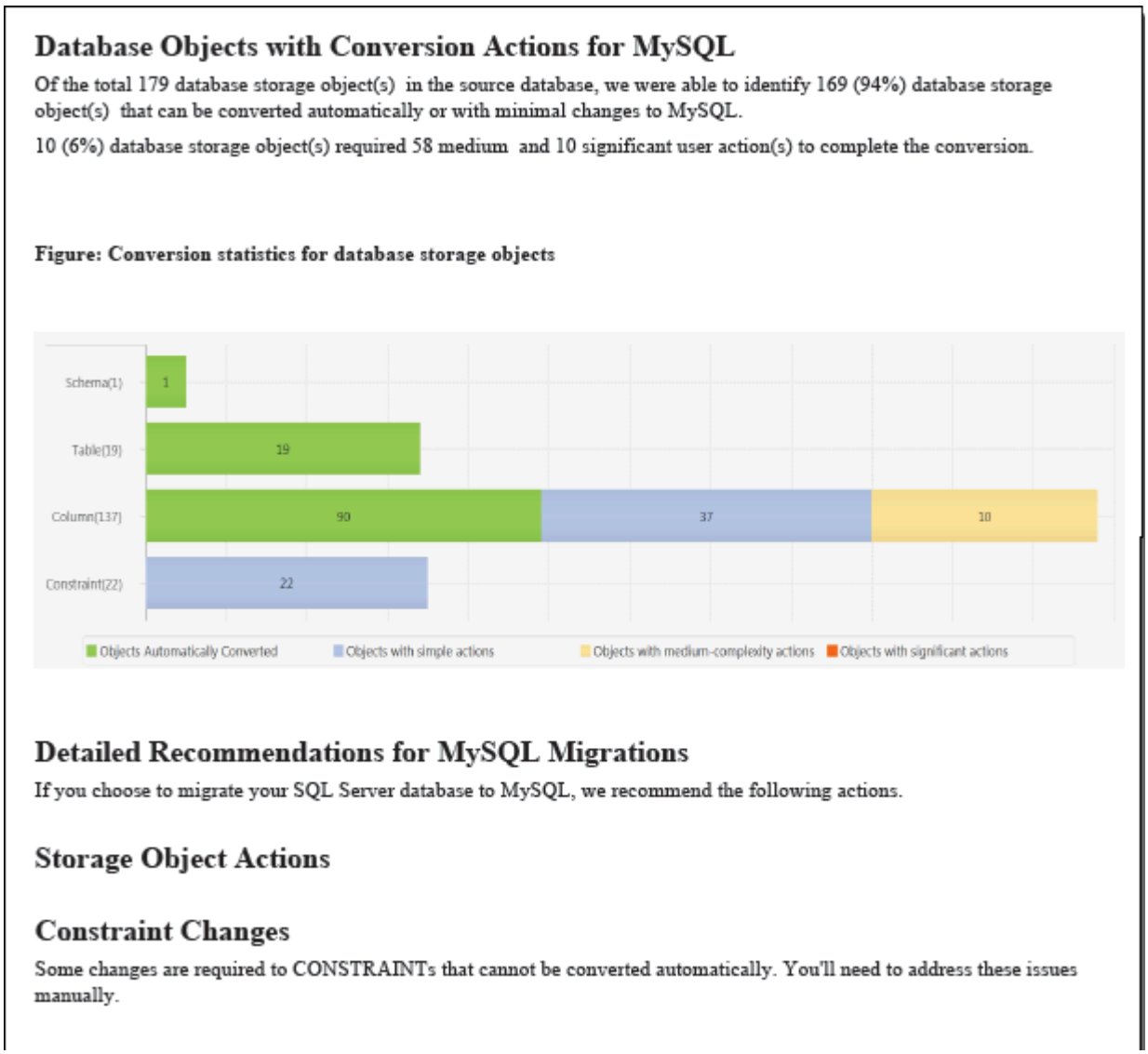


*Choose **Create Report***

The **Summary** tab displays the summary information from the database migration assessment report. It shows items that were automatically converted and items that could not be automatically converted.

For schema items that could not be automatically converted to the target database engine, the summary includes an estimate of the effort that it would take to create a schema that is equivalent to your source database in your target DB instance. The report categorizes the estimated time to convert these schema items as follows:

- **Simple** – Actions that can be completed in less than one hour.
- **Medium** – Actions that are more complex and can be completed in one to four hours.
- **Significant** – Actions that are very complex and will take more than four hours to complete.



Database Objects with Conversion Actions for MySQL

Of the total 179 database storage object(s) in the source database, we were able to identify 169 (94%) database storage object(s) that can be converted automatically or with minimal changes to MySQL.

10 (6%) database storage object(s) required 58 medium and 10 significant user action(s) to complete the conversion.

Figure: Conversion statistics for database storage objects

Detailed Recommendations for MySQL Migrations

If you choose to migrate your SQL Server database to MySQL, we recommend the following actions.

Storage Object Actions

Constraint Changes

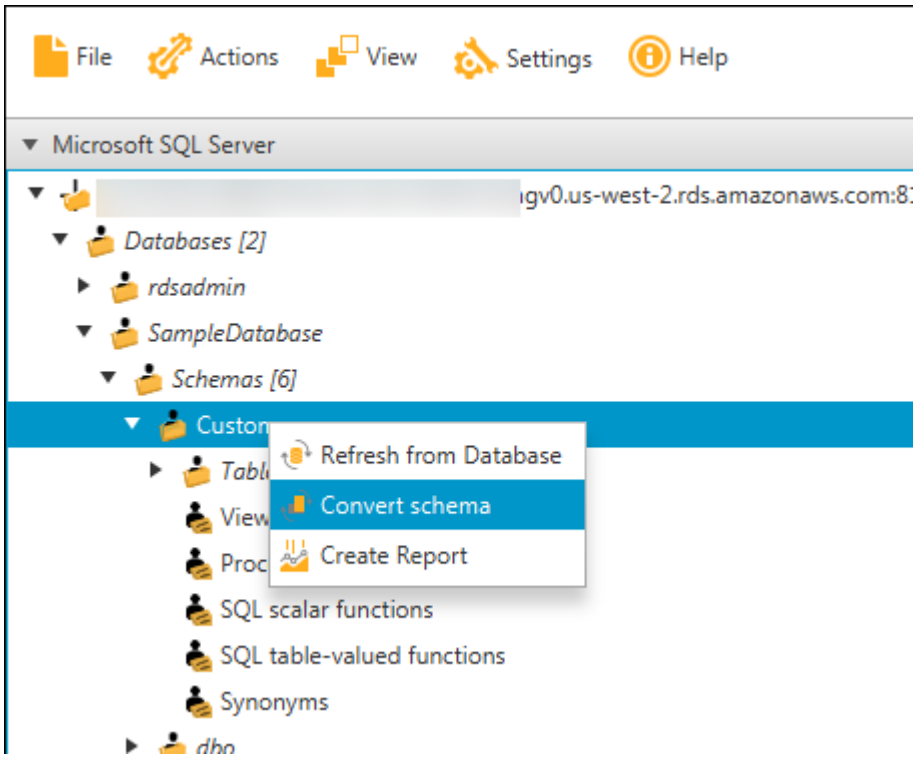
Some changes are required to CONSTRAINTs that cannot be converted automatically. You'll need to address these issues manually.

Migration report

> **Note**
>
> **Important:** If you are evaluating the effort required for your database migration project, this assessment report is an important artifact to consider. Study the assessment report in details to determine what code changes are required in the database schema and what impact the changes might have on your application functionality and design.

13. The next step is to convert the schema. The converted schema is not immediately applied to the target database. Instead, it is stored locally until you explicitly apply the converted schema to the target database. To convert the schema from your source database, choose a schema

object to convert from the left panel of your project. Right-click the object and choose **Convert schema**, as shown in the following illustration.



*Choose **Convert schema***

This action adds converted schema to the right panel of the project window and shows objects that were automatically converted by the AWS Schema Conversion Tool.

You can respond to the action items in the assessment report in different ways:

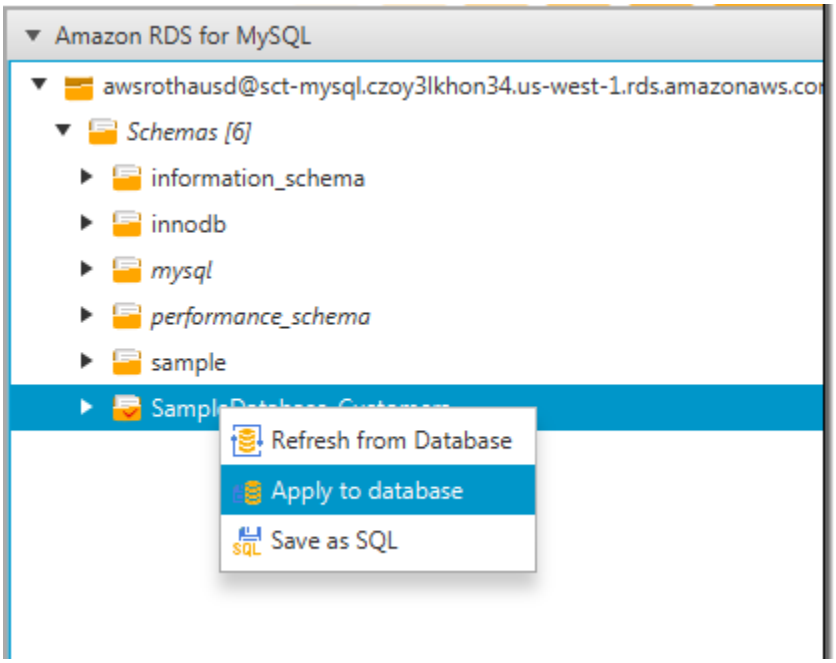
- **Add equivalent schema manually** — You can write the portion of the schema that can be automatically converted to your target DB instance by choosing Apply to database in the right panel of your project. The schema that is written to your target DB instance won't contain the items that couldn't be automatically converted. Those items are listed in your database migration assessment report.

  After applying the schema to your target DB instance, you can then manually create the schema in your target DB instance for the items that could not be automatically converted. In some cases, you may not be able to create an equivalent schema in your target DB instance. You might need to redesign a portion of your application and database to use the functionality that is available from the DB engine for your target DB instance. In other cases, you can simply ignore the schema that can't be automatically converted.

> **Note**
>
> **Caution:** If you manually create the schema in your target DB instance, do not choose **Apply to database** until after you have saved a copy of any manual work that you have done. Applying the schema from your project to your target DB instance overwrites schema of the same name in the target DB instance, and you lose any updates that you added manually.

- **Modify your source database schema and refresh the schema in your project** — For some items, you might be best served to modify the database schema in your source database to the schema that is compatible with your application architecture and that can also be automatically converted to the DB engine of your target DB instance. After updating the schema in your source database and verifying that the updates are compatible with your application, choose **Refresh from Database** in the left panel of your project to update the schema from your source database. You can then convert your updated schema and generate the database migration assessment report again. The action item for your updated schema no longer appears.

14. When you are ready to apply your converted schema to your target Aurora instance, choose the schema element from the right panel of your project. Right- click the schema element and choose **Apply to database**, as shown in the following figure.



*Choose **Apply to database***

> **Note**
>
> The first time that you apply your converted schema to your target DB instance, the AWS Schema Conversion Tool adds an additional schema (`AWS_ORACLE_EXT` or `AWS_SQLSERVER_EXT`) to your target DB instance. This schema implements system functions of the source database that are required when writing your converted schema to your target DB instance. Do not modify this schema, or you might encounter unexpected results in the converted schema that is written to your target DB instance. When your schema is fully migrated to your target DB instance, and you no longer need the AWS Schema Conversion Tool, you can delete the `AWS_ORACLE_EXT` or `AWS_SQLSERVER_EXT` schema.

The AWS Schema Conversion Tool is an easy-to-use addition to your migration toolkit. For additional best practices related to AWS Schema Conversion Tool, refer to the Best practices for the AWS SCT topic in the AWS Schema Conversion Tool User Guide.

Migrating data

After the database schema has been copied from the source database to the target Aurora database, the next step is to migrate actual data from source to target. While data migration can be accomplished using different tools, AWS recommends moving data using the AWS Database Migration Service (AWS DMS) as it provides both the simplicity and the features needed for the task at hand.

Introduction and general approach to AWS DMS

The AWS Database Migration Service (AWS DMS) makes it easy for customers to migrate production databases to AWS with minimal downtime. You can keep your applications running while you are migrating your database. In addition, the AWS Database Migration Service ensures that data changes to the source database that occur during and after the migration are continuously replicated to the target. Migration tasks can be set up in minutes in the AWS Management Console. The AWS Database Migration Service can migrate your data to and from widely used database platforms, such as Oracle, SQL Server, MySQL, PostgreSQL, Amazon Aurora, MariaDB, and Amazon Redshift.

The service supports homogenous migrations such as Oracle to Oracle, as well as heterogeneous migrations between different database platforms, such as Oracle to Amazon Aurora or SQL Server to MySQL. You can perform one-time migrations, or you can maintain continuous replication between databases without a customer having to install or configure any complex software.

AWS DMS works with databases that are on premises, running on Amazon EC2, or running on Amazon RDS. However, AWS DMS does not work in situations where both the source database and the target database are on premises; one endpoint must be in AWS.

AWS DMS supports specific versions of Oracle, SQL Server, Amazon Aurora, MySQL, and PostgreSQL. For currently supported versions, refer to Sources for data migration. However, this whitepaper is just focusing on Amazon Aurora as a migration target.

Migration methods

AWS DMS provides three methods for migrating data:

- **Migrate existing data** — This method creates the tables in the target database, automatically defines the metadata that is required at the target, and populates the tables with data from

the source database (also referred to as a “full load”). The data from the tables is loaded in parallel for improved efficiency. Tables are only created in case of homogenous migrations, and secondary indexes aren’t created automatically by AWS DMS. Read further for details.

- **Migrate existing data and replicate ongoing changes** — This method does a full load, as described above, and in addition captures any ongoing changes being made to the source database during the full load and stores them on the replication instance. Once the full load is complete, the stored changes are applied to the destination database until it has been brought up to date with the source database. Additionally, any ongoing changes being made to the source database continue to be replicated to the destination database to keep them in sync. This migration method is very useful when you want to perform a database migration with very little downtime.
- **Replicate data changes only** — This method just reads changes from the recovery log file of the source database and applies these changes to the target database on an ongoing basis. If the target database is unavailable, these changes are buffered on the replication instance until the target becomes available.
- When AWS DMS is performing a full load migration, the processing puts a load on the tables in the source database, which could affect the performance of applications that are hitting this database at the same time. If this is an issue, and you cannot shut down your applications during the migration, you can consider the following approaches:

- Running the migration at a time when the application load on the database is at its lowest point.
- Creating a read replica of your source database and then performing the AWS DMS migration from the read replica.

Migration procedure

The general outline for using AWS DMS is as follows:

1. Create a target database.
2. Copy the schema.
3. Create an AWS DMS replication instance.
4. Define the database source and target endpoints.
5. Create and run a migration task.

Create target database

Create your target Amazon Aurora database cluster using the procedure outlined in Creating an Amazon Aurora DB Cluster. You should create the target database in the Region and with an instance type that matches your business requirements. Also, to improve the performance of the migration, verify that your target database does not have multi-AZ deployment enabled; you can enable that once the load has finished.

Copy schema

Additionally, you should create the schema in this target database. AWS DMS supports basic schema migration, including the creation of tables and primary keys. However, AWS DMS doesn't automatically create secondary indexes, foreign keys, stored procedures, users, and so on, in the target database. For full schema migration details, refer to the *Migrating the database schema* section of this document.

Create an AWS DMS replication instance

In order to use the AWS DMS service, you must create an AWS DMS replication instance, which runs in your VPC. This instance reads the data from the source database, performs the specified table mappings, and writes the data to the target database. In general, using a larger replication instance size speeds up the database migration (although the migration can also be gated by other factors such as the capacity of the source and target databases, connection latency, and so on.). Also, your replication instance can be stopped once your database migration is complete.

AWS Database Migration Service

AWS DMS currently supports burstable, compute and memory-optimized instance classes for replication instances. The burstable instance classes are low-cost standard instances designed to provide a baseline level of CPU performance with the ability to burst above the baseline. They are

suitable for developing, configuring, and testing your database migration process as well as for periodic data migration tasks that can benefit from the CPU burst capability.

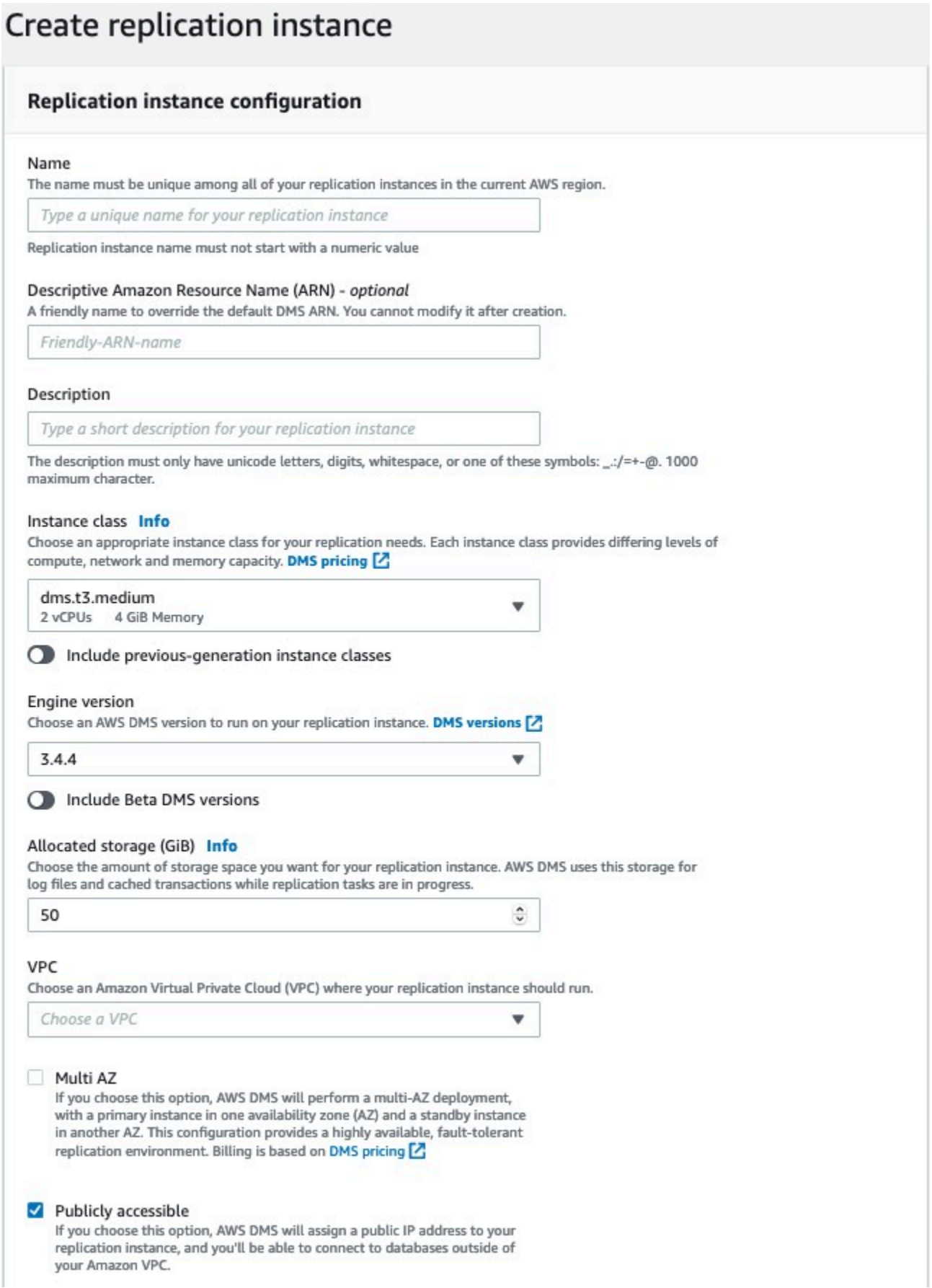
The compute-optimized instance classes are designed to deliver the highest level of processor performance and achieve significantly higher packet per second (PPS) performance, lower network jitter, and lower network latency. You should use this instance class if you are performing large heterogeneous migrations and want to minimize the migration time.

The memory-optimized instance classes are designed for migrations or replications of high-throughput transaction systems which can consume large amounts of CPU and memory.

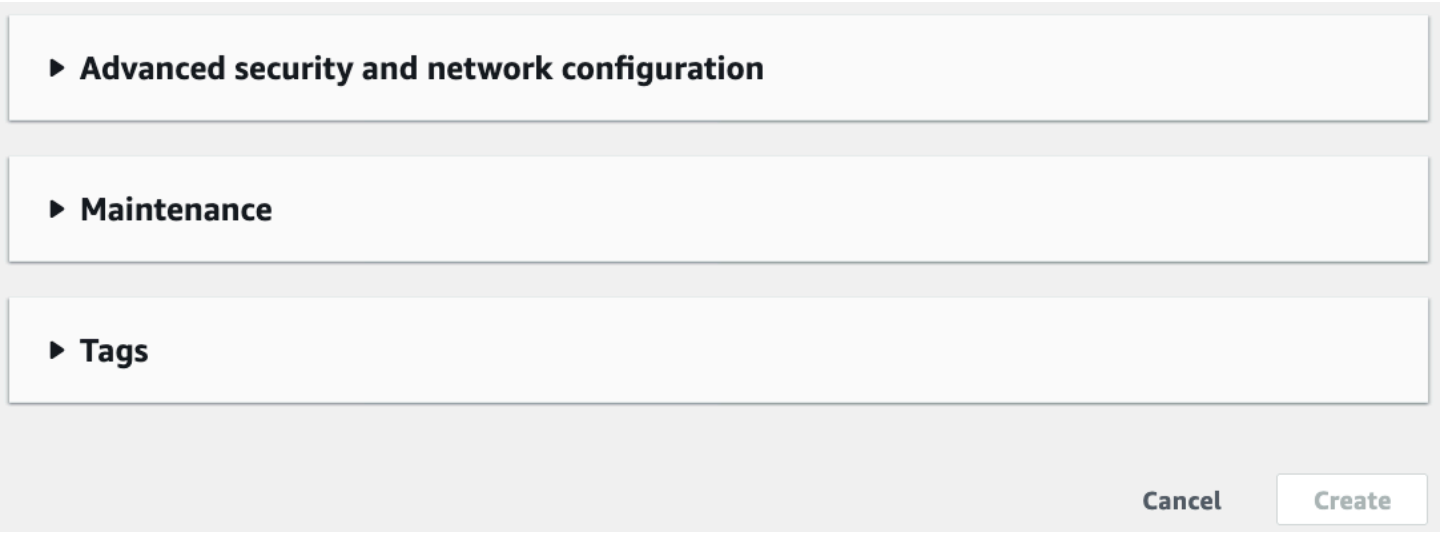
AWS DMS Storage is primarily consumed by log files and cached transactions. Normally, doing a full load does not require a significant amount of instance storage on your AWS DMS replication instance. However, if you are doing replication along with your full load, then the changes to the source database are stored on the AWS DMS replication instance while the full load is taking place. If you are migrating a very large source database that is also receiving a lot of updates while the migration is in progress, then a significant amount of instance storage could be consumed.

The instances come with 50 GB of instance storage but can be scaled up as appropriate. Normally, this amount of storage should be more than adequate for most migration scenarios. However, it's always a good idea to pay attention to storage-related metrics. Make sure to scale up your storage if you find you are consuming more than the default allocation.

Also, in some extreme cases where very large databases with very high transaction rates are being migrated with replication enabled, it is possible that the AWS DMS replication may not be able to catch up in time. If you encounter this situation, it may be necessary to stop the changes to the source database for some number of minutes in order for the replication to catch up before you repoint your application to the target Aurora DB.


Create replication instance
Replication instance configuration
Name
The name must be unique among all of your replication instances in the current AWS region.
Type a unique name for your replication instance
Replication instance name must not start with a numeric value
Descriptive Amazon Resource Name (ARN) - optional
A friendly name to override the default DMS ARN. You cannot modify it after creation.
Friendly-ARN-name
Description
Type a short description for your replication instance
The description must only have unicode letters, digits, whitespace, or one of these symbols: _.:/=+-@. 1000 maximum character.
Instance class Info
Choose an appropriate instance class for your replication needs. Each instance class provides differing levels of compute, network and memory capacity. DMS pricing
dms.t3.medium
2 vCPUs 4 GiB Memory
Include previous-generation instance classes
Engine version
Choose an AWS DMS version to run on your replication instance. DMS versions
3.4.4
Include Beta DMS versions
Allocated storage (GiB) Info
Choose the amount of storage space you want for your replication instance. AWS DMS uses this storage for log files and cached transactions while replication tasks are in progress.
50
VPC
Choose an Amazon Virtual Private Cloud (VPC) where your replication instance should run.
Choose a VPC
Multi AZ
If you choose this option, AWS DMS will perform a multi-AZ deployment, with a primary instance in one availability zone (AZ) and a standby instance in another AZ. This configuration provides a highly available, fault-tolerant replication environment. Billing is based on DMS pricing
Publicly accessible
If you choose this option, AWS DMS will assign a public IP address to your replication instance, and you'll be able to connect to databases outside of your Amazon VPC.

Create replication instance page in the AWS DMS console



Options on the create replication instance page in the AWS DMS console

Define database source and target endpoints

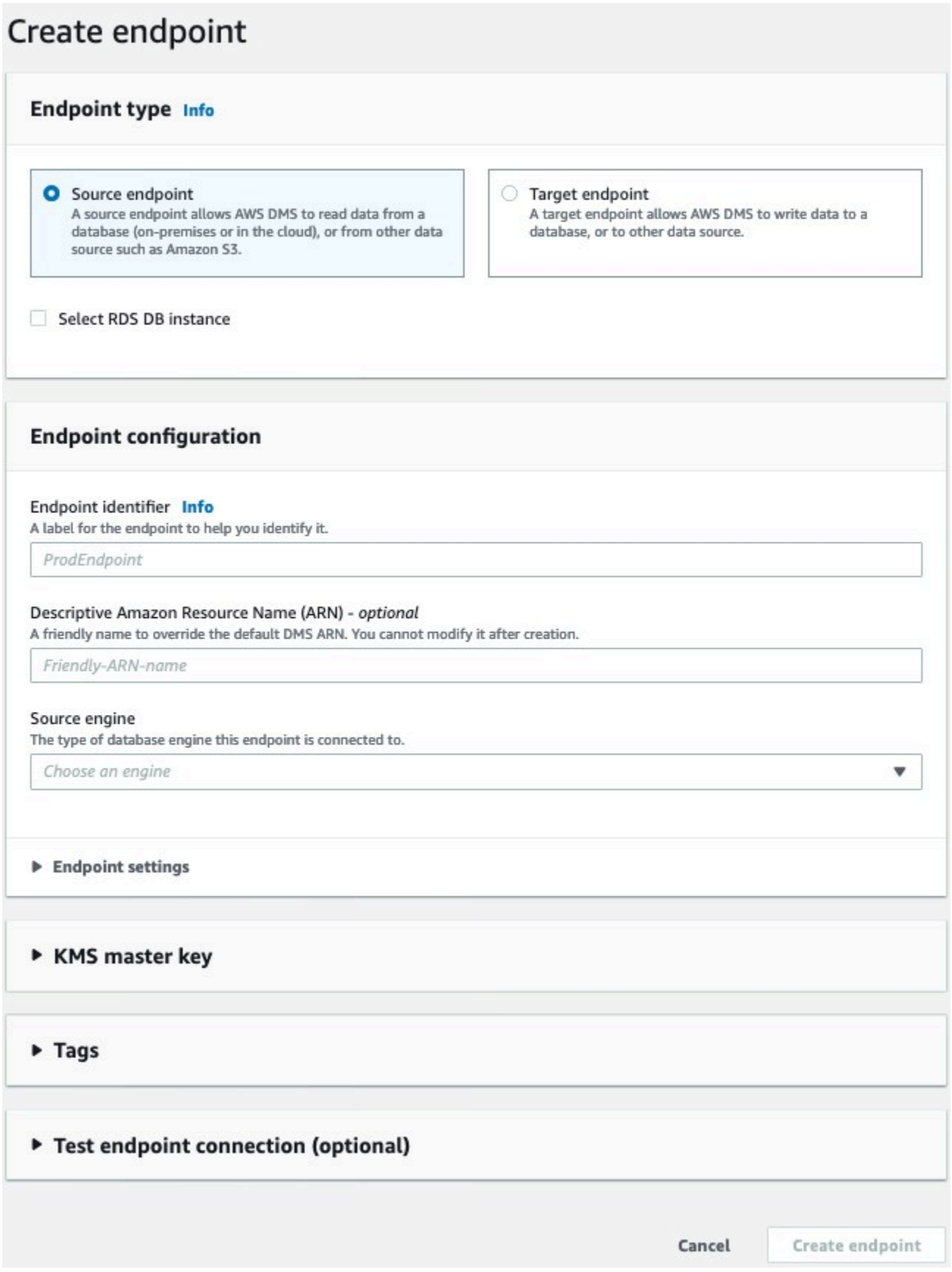
A database endpoint is used by the replication instance to connect to a database. To perform a database migration, you must create *both* a source database endpoint and a target database endpoint. The specified database endpoints can be on premises, running on Amazon EC2, or running on Amazon RDS, but the source and target cannot both be on premises.

AWS highly recommendeds that you test your database endpoint connection after you define it. The same page used to create a database endpoint can also be used to test it, as explained later in this paper.

Note: If you have foreign key constraints in your source schema, when creating your target endpoint you need to enter the following for **Extra connection attributes** in the **Advanced** section:

```
initstmt=SET FOREIGN_KEY_CHECKS=0
```

This disables the foreign key checks while the target tables are being loaded. This in turn prevents the load from being interrupted by failed foreign key checks on partially loaded tables.


Create endpoint
Endpoint type Info
Source endpoint
A source endpoint allows AWS DMS to read data from a database (on-premises or in the cloud), or from other data source such as Amazon S3.
Target endpoint
A target endpoint allows AWS DMS to write data to a database, or to other data source.
Select RDS DB instance
Endpoint configuration
Endpoint identifier Info
A label for the endpoint to help you identify it.
ProdEndpoint
Descriptive Amazon Resource Name (ARN) - optional
A friendly name to override the default DMS ARN. You cannot modify it after creation.
Friendly-ARN-name
Source engine
The type of database engine this endpoint is connected to.
Choose an engine
Endpoint settings
KMS master key
Tags
Test endpoint connection (optional)
Cancel
Create endpoint

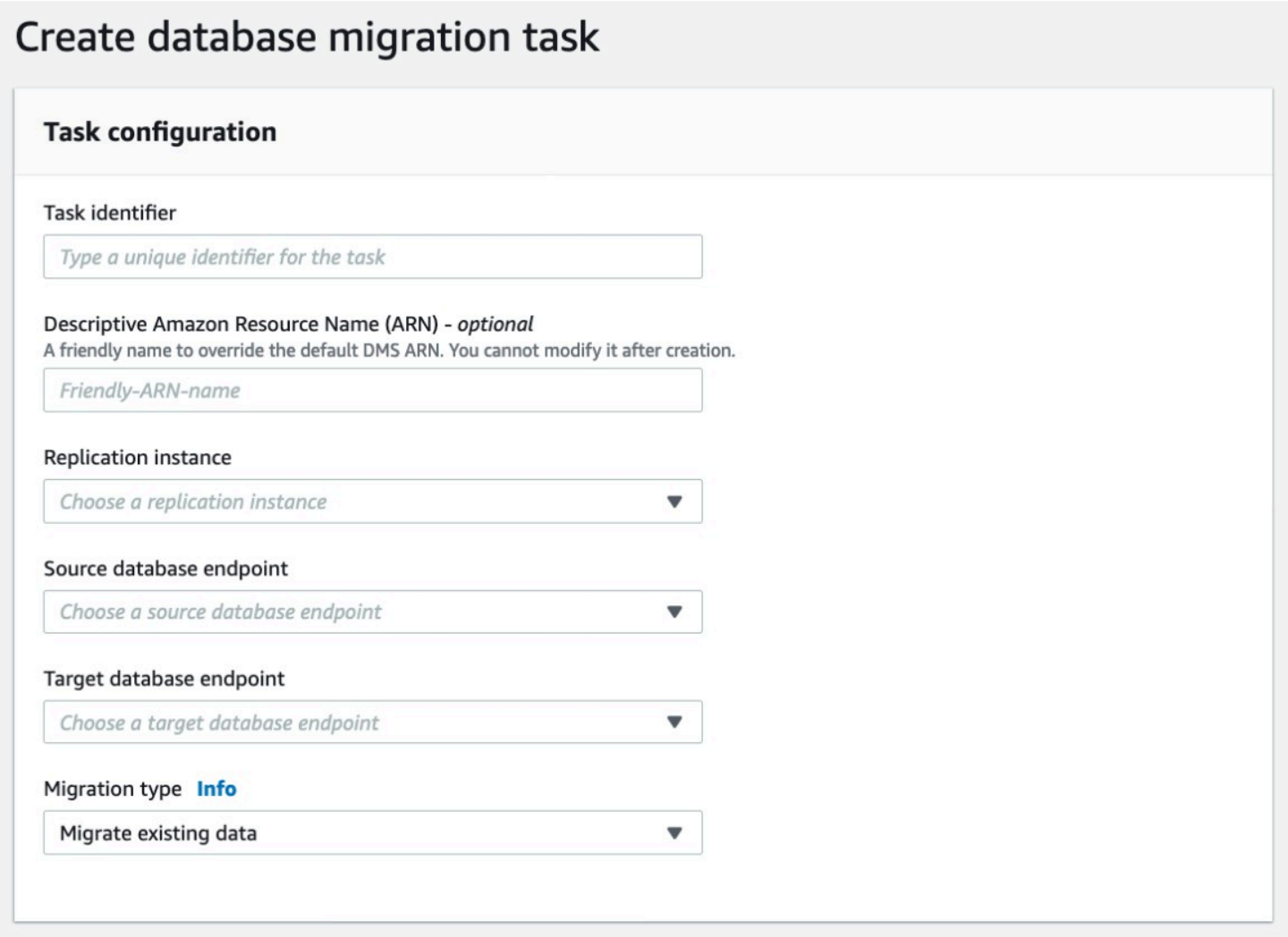
Create database endpoint page in the AWS DMS console

Create and run a migration task

Now that you have created and tested your source database endpoint and your target database endpoint, you can create a task to do the data migration. When you create a task, you specify the replication instance that you have created, the database migration method type (discussed earlier), the source database endpoint, and your target database endpoint for your Amazon Aurora database cluster.

Also, under **Task Settings**, if you have already created the full schema in the target database, then you should change the **Target table preparation mode** to **Do nothing** rather than using the default value of **Drop tables on target**. The latter can cause you to lose aspects of your schema definition like foreign key constraints when it drops and recreates tables.

When creating a task, you can create table mappings that specify the source schema along with the corresponding tables to be migrated to the target endpoint. The default mapping method migrates all source tables to target tables of the same name if they exist. Otherwise, it creates the source table(s) on the target (depending on your task settings). Additionally, you can create custom mappings (using a JSON file) if you want to migrate only certain tables or if you want to have more control over the field and table mapping process. You can also choose to migrate only one schema or all schemas from your source endpoint.



Create database migration task page in the AWS DMS console

Task settings

Editing mode **Info**

- (•) **Wizard**
  You can enter only a subset of the available task settings.
- ( ) **JSON editor**
  You can enter all available task settings directly in JSON format.

Target table preparation mode **Info**

- ( ) Do nothing
- (•) Drop tables on target
- ( ) Truncate

Include LOB columns in replication **Info**

- ( ) Don't include LOB columns
- ( ) Full LOB mode
- (•) Limited LOB mode

Maximum LOB size (KB) **Info**

32

- [ ] **Enable validation**
  Choose this setting if you want AWS DMS to compare the data at the source and the target, immediately after it performs a full data load. Validation ensures that your data was migrated accurately, but it requires additional time to complete.

- [ ] **Enable CloudWatch logs** **Info**
  DMS task logging uses Amazon CloudWatch to log information during the migration process. You can change the component activities logged and the amount of information logged for each one.

▶ **Advanced task settings**

Premigration assessment **Info**

A premigration assessment warns you of potential migration issues before starting your migration task. Premigration assessments generally have minimal impact on your databases and take minimal time to run.

- [ ] **Enable premigration assessment run**

Migration task startup configuration

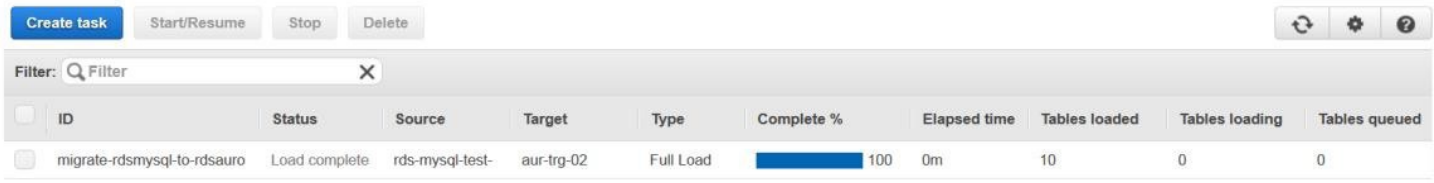
Start migration task

- (•) **Automatically on create**
  Available only if the premigration assessment is not enabled.
- ( ) **Manually later**

▶ **Tags**

Task settings page in the AWS DMS console

You can use the AWS Management Console to monitor the progress of your AWS Database Migration Service (AWS DMS) tasks. You can also monitor the resources and network connectivity used. The AWS DMS console shows basic statistics for each task, including the task status, percent complete, elapsed time, and table statistics, as the following image shows.



AWS DMS Console task statistics

Additionally, you can select a task and display performance metrics for that task, including throughput, records per second migrated, disk and memory use, and latency.

Testing and cutover

Once the schema and data have been successfully migrated from the source database to Amazon Aurora, you are now ready to perform end-to-end testing of your migration process. The testing approach should be refined after each test migration, and the final migration plan should include a test plan that ensures adequate testing of the migrated database.

Migration testing

Table 2 — Migration testing

Test category	Purpose
Basic acceptance tests	These pre-cutover tests should be automatic ally run upon completion of the data migration process. Their primary purpose is to verify whether the data migration was successful. Following are some common outputs from these tests: Total number of items processed Total number of items imported Total number of items skipped Total number of warnings Total number of errors If any of these totals reported by the tests deviate from the expected values, then it means the migration was not successful, and the issues need to be resolved before moving to the next step in the process or the next round of testing.

Test category	Purpose
Functional tests	These post-cutover tests exercise the functionality of the application(s) using Aurora for data storage. They include a combination of automated and manual tests. The primary purpose of the functional tests is to identify problems in the application caused by the migration of the data to Aurora.
Nonfunctional tests	These post-cutover tests assess the nonfuncti onal characteristics of the application, such as performance under varying levels of load.
User acceptance tests	These post-cutover tests should be run by the end users of the application once the final data migration and cutover is complete. The purpose of these tests is for the end users to decide if the application is sufficiently usable to meet its primary function in the organizat ion.

Cutover

Once you have completed the final migration and testing, it is time to point your application to the Amazon Aurora database. This phase of migration is known as *cutover*. If the planning and testing phase has been run properly, cutover should not lead to unexpected issues.

Pre-cutover actions

- **Choose a cutover window** — Identify a block of time when you can accomplish cutover to the new database with minimum disruption to the business. Normally you would select a low activity period for the database (typically nights and/or weekends).
- **Make sure changes are caught up** — If a near-zero downtime migration approach was used to replicate database changes from the source to the target database, make sure that all database

changes are caught up and your target database is not significantly lagging behind the source database.

- **Prepare scripts to make the application configuration changes** — In order to accomplish the cutover, you need to modify database connection details in your application configuration files. Large and complex applications may require updates to connection details in multiple places. Make sure you have the necessary scripts ready to update the connection configuration quickly and reliably.
- **Stop the application** — Stop the application processes on the source database and put the source database in read-only mode so that no further writes can be made to the source database. If the source database changes aren't fully caught up with the target database, wait for some time while these changes are fully propagated to the target database.
- **Run pre-cutover tests** — Run automated pre-cutover tests to make sure that the data migration was successful.

Cutover

- **Run cutover** — If pre-cutover checks were completed successfully, you can now point your application to Amazon Aurora. Run scripts created in the pre-cutover phase to change the application configuration to point to the new Aurora database.
- **Start your application** — At this point, you may start your application. If you have an ability to stop users from accessing the application while the application is running, exercise that option until you have run your post-cutover checks.

Post-cutover checks

- **Run post-cutover tests** — Run predefined automated or manual test cases to make sure your application works as expected with the new database. It's a good strategy to start testing read-only functionality of the database first before running tests that write to the database.
- **Enable user access and closely monitor** — If your test cases were run successfully, you may give user access to the application to complete the migration process. Both application and database should be closely monitored at this time.

Conclusion

Amazon Aurora is a high performance, highly available, and enterprise-grade database built for the cloud. Leveraging Amazon Aurora can result in better performance and greater availability than other open-source databases and lower costs than most commercial grade databases. This paper proposes strategies for identifying the best method to migrate databases to Amazon Aurora and details the procedures for planning and completing those migrations. In particular, AWS Database Migration Service (AWS DMS) as well as the AWS Schema Conversion Tool are the recommended tools for heterogeneous migration scenarios. These powerful tools can greatly reduce the cost and complexity of database migrations.

Contributors

Contributors to this document include:

- Puneet Agarwal, Solutions Architect, Amazon Web Services
- Chetan Nandikanti, Database Specialist Solutions Architect, Amazon Web Services
- Scott Williams, Solutions Architect, Amazon Web Services
- Jonathan Doe, Solutions Architect, Amazon Web Services
- Achin Agrawal, Senior Solutions Architect, Amazon Web Services

Further reading

For additional information, refer to:

- Amazon Aurora Product Details
- Amazon Aurora FAQs
- AWS Database Migration Service
- AWS Database Migration Service FAQs

Document history

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Whitepaper updated	Updated the migration method using Aurora Read Replica to reduce the downtime when migrating from Amazon RDS MySQL to a few seconds.	June 29, 2023
Whitepaper updated	Reviewed for technical accuracy.	July 28, 2021
Initial publication	Whitepaper published.	June 10, 2016

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.