**Migration Strategy of Microsoft SQL server 2016 with SSIS jobs and service broker to PostgreSQL**

Contents

[Existing environment details 1](#_Toc155987180)

[Assumptions: 1](#_Toc155987181)

[Summary 2](#_Toc155987182)

[Prerequisites and limitations 2](#_Toc155987183)

[Tools 3](#_Toc155987184)

[Epics 4](#_Toc155987185)

[Plan the migration 4](#_Toc155987186)

[Configure the infrastructure 4](#_Toc155987187)

[Migrate data - option 1 4](#_Toc155987188)

[Migrate data - option 2 5](#_Toc155987189)

[Migrate the application 5](#_Toc155987190)

[Cut over 5](#_Toc155987191)

[Close the project 5](#_Toc155987192)

[Privileges for PostgreSQL as a target database 6](#_Toc155987193)

[SQL Server to PostgreSQL conversion settings 6](#_Toc155987194)

[Converting SQL Server partitions to PostgreSQL version 10 partitions 7](#_Toc155987195)

[Migration considerations 8](#_Toc155987196)

[PostgreSQL Equivalent for MS SQL server message broker and SSIS jobs 14](#_Toc155987197)

Existing environment details:

CPU Cores: 24 Core CPU

Memory: 256 GB

Database Size: 10 TB

Transaction Size per month: 10 GB

## Assumptions:

Let’s assume Microsoft SQL server 2016 is running on On-premise server. It needs to be migrated to PostgreSQL on AWS RDS (Cloud). From the above mentioned existing environment details we need to provision PostgreSQL AWS RDS instance with

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Model | Core Count | vCPU\* | Memory (GiB) | Storage | Dedicated EBS Bandwidth (Gbps) | Network Performance (Gbps) |
| db.m7g.16xlarge | - | 64 | 256 | EBS-Only | 20 | 30 |

|  |  |  |
| --- | --- | --- |
| **Environment:**PoC or pilot | **Source:**Microsoft SQL Server | **Target:**Amazon RDS for SQL Server |
| **R Type:**Replatform | **Workload:**Microsoft | **Technologies:**Migration; Databases |
| **AWS services:**Amazon RDS |  |  |

## Summary

This pattern provides guidance for migrating from an on-premises Microsoft SQL Server database to Amazon Relational Database Service (Amazon RDS) for SQL Server. It describes two options for migration: using AWS Data Migration Service (AWS DMS) or using native Microsoft SQL Server tools such as Copy Database Wizard.

## Prerequisites and limitations

**Prerequisites**

* An active AWS account
* A source Microsoft SQL Server database in an on-premises data center

**Limitations**

* Database size limit: 16 TB

**Product versions**

* SQL Server 2014-2019, Enterprise, Standard, Workgroup, and Developer editions. For the latest list of supported versions and features,

**Source technology stack**

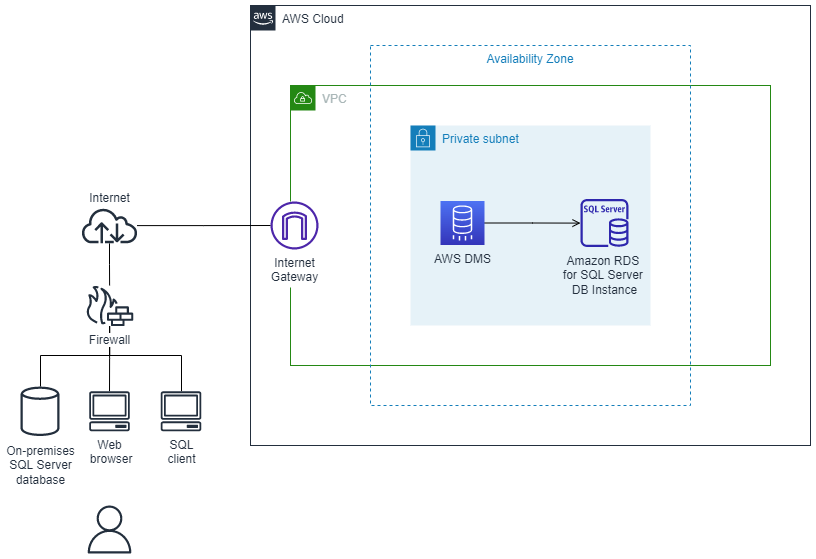
* An on-premises Microsoft SQL Server database

**Target technology stack**

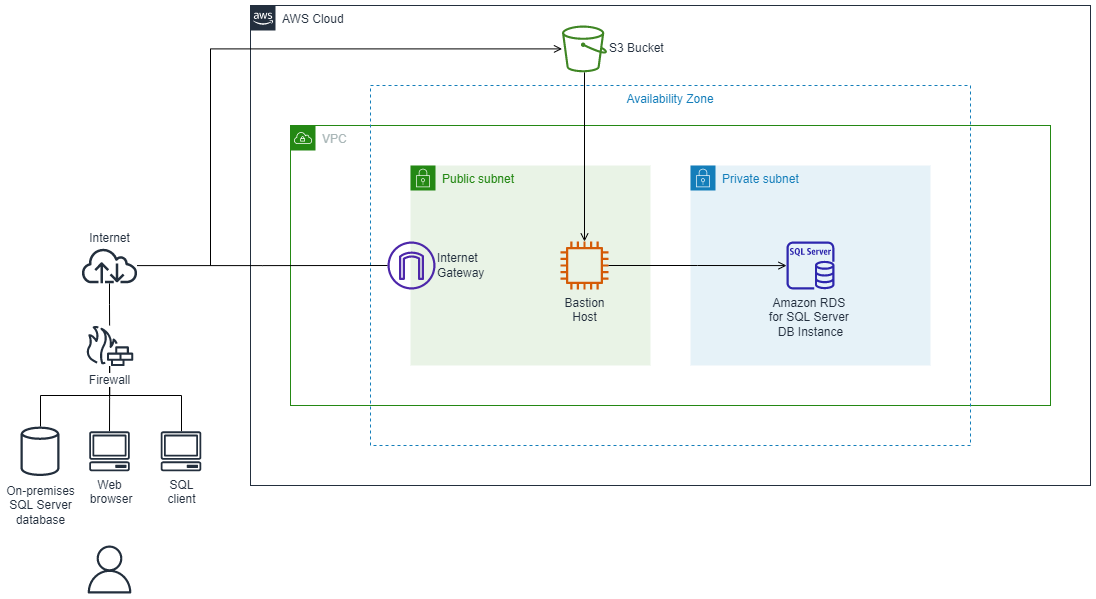
* An Amazon RDS for PostgreSQL DB instance

**Source and target architecture**

*Using AWS DMS:*



*Using native SQL Server tools:*



## Tools

* AWS DMS supports several types of source and target databases. If AWS DMS doesn't support the source database, select another method for migrating the data.
* Native Microsoft SQL Server tools include backup and restore, Copy Database Wizard, copy and attach database.
* AWS Schema conversion tool SCT

## Epics

## Plan the migration

| **Task** | **Description** | **Skills required** |
| --- | --- | --- |
| Validate the source and target database version and engine. |  | DBA |
| Identify the hardware requirements for the target server instance. |  | DBA, Systems administrator |
| Identify the storage requirements (storage type and capacity). |  | DBA, Systems administrator |
| Choose the proper instance type based on capacity, storage features, and network features. |  | DBA, Systems administrator |
| Identify the network access security requirements for source and target databases. |  | DBA, Systems administrator |
| Identify the application migration strategy. |  | DBA, Systems administrator |

## Configure the infrastructure

| **Task** | **Description** | **Skills required** |
| --- | --- | --- |
| Create a virtual private cloud (VPC). |  | Systems administrator |
| Create security groups. |  | Systems administrator |
| Configure and start an Amazon RDS DB instance. |  | DBA, Systems administrator |

## Migrate data - option 1

| **Task** | **Description** | **Skills required** |
| --- | --- | --- |
| Use native SQL Server tools or third-party tools AWS SCT to migrate database objects and data. |  | DBA |

## Migrate data - option 2

| **Task** | **Description** | **Skills required** |
| --- | --- | --- |
| Migrate data with AWS DMS and AWS SCT. |  | DBA |

## Migrate the application

| **Task** | **Description** | **Skills required** |
| --- | --- | --- |
| Follow the application migration strategy. |  | DBA, App owner, Systems administrator |

## Cut over

| **Task** | **Description** | **Skills required** |
| --- | --- | --- |
| Switch the application clients over to the new infrastructure. |  | DBA, App owner, Systems administrator |

## Close the project

| **Task** | **Description** | **Skills required** |
| --- | --- | --- |
| Shut down the temporary AWS resources. |  | DBA, Systems administrator |
| Review and validate the project documents. |  | DBA, App owner, Systems administrator |
| Gather metrics such as time to migrate, percentage of manual versus automated tasks, and cost savings. |  | DBA, App owner, Systems administrator |
| Close out the project and provide feedback. |  | DBA, App owner, Systems administrator |

## Privileges for PostgreSQL as a target database

To use PostgreSQL as a target, AWS SCT requires the CREATE ON DATABASE privilege. Make sure that you grant this privilege for each target PostgreSQL database.

To use the converted public synonyms, change the database default search path to "$user", public\_synonyms, public.

You can use the following code example to create a database user and grant the privileges.

CREATE ROLE *user\_name* LOGIN PASSWORD '*your\_password*';

GRANT CREATE ON DATABASE *db\_name* TO *user\_name*;

ALTER DATABASE *db\_name* SET SEARCH\_PATH = "$user", public\_synonyms, public;

In the preceding example, replace *user\_name* with the name of your user. Then, replace *db\_name* with the name of your target database. Finally, replace *your\_password* with a secure password.

In PostgreSQL, only the schema owner or a superuser can drop a schema. The owner can drop a schema and all objects that this schema includes even if the owner of the schema doesn't own some of its objects.

When you use different users to convert and apply different schemas to your target database, you can get an error message when AWS SCT can't drop a schema. To avoid this error message, use the superuser role.

## SQL Server to PostgreSQL conversion settings

To edit SQL Server to PostgreSQL conversion settings, choose **Settings**, and then choose **Conversion settings**. From the upper list, choose **SQL Server**, and then choose **SQL Server – PostgreSQL**. AWS SCT displays all available settings for SQL Server to PostgreSQL conversion.

SQL Server to PostgreSQL conversion settings in AWS SCT include options for the following:

* To limit the number of comments with action items in the converted code.

For **Add comments in the converted code for the action items of selected severity and higher**, choose the severity of action items. AWS SCT adds comments in the converted code for action items of the selected severity and higher.

For example, to minimize the number of comments in your converted code, choose **Errors only**. To include comments for all action items in your converted code, choose **All messages**.

* To allow to use indexes with the same name in different tables in SQL Server.

In PostgreSQL, all index names that you use in the schema, must be unique. To make sure that AWS SCT generates unique names for all your indexes, select **Generate unique names for indexes**.

* To convert SQL Server procedures to PostgreSQL functions.

PostgreSQL version 10 and earlier doesn't support procedures. For customers who aren't familiar with using procedures in PostgreSQL, AWS SCT can convert procedures to functions. To do so, select **Convert procedures to functions**.

* To emulate the output of EXEC in a table.

Your source SQL Server database can store the output of EXEC in a table. AWS SCT creates temporary tables and an additional procedure to emulate this feature. To use this emulation, select **Create additional routines for handling open datasets**.

* To define the template to use for the schema names in the converted code. For **Schema name generation template**, choose one of the following options:
  + **<source\_db>** – Uses the SQL Server database name as a schema name in PostgreSQL.
  + **<source\_schema>** – Uses the SQL Server schema name as a schema name in PostgreSQL.
  + **<source\_db>\_<schema>** – Uses a combination of the SQL Server database and schema names as a schema name in PostgreSQL.
* To keep the letter case of your source object names.

To avoid conversion of object names to lower case, select **Avoid casting to lower case for case sensitive operations**. This option applies only when you turn on case sensitivity option in your target database.

* To keep the parameter names from your source database.

To add double quotation marks to the names of parameters in the converted code, select **Keep original parameter names**.

## Converting SQL Server partitions to PostgreSQL version 10 partitions

When you convert a Microsoft SQL Server database to Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) or Amazon Relational Database Service for PostgreSQL (Amazon RDS for PostgreSQL), be aware of the following.

In SQL Server, you create partitions with partition functions. When converting from a SQL Server portioned table to a PostgreSQL version 10 partitioned table, be aware of several potential issues:

* SQL Server allows you to partition a table using a column without a NOT NULL constraint. In that case, all NULL values go to the leftmost partition. PostgreSQL doesn’t support NULL values for RANGE partitioning.
* SQL Server allows you to create primary and unique keys for partitioned tables. For PostgreSQL, you create primary or unique keys for each partition directly. Thus, PRIMARY or UNIQUE KEY constraint must be removed from their parent table when migrating to PostgreSQL. The resulting key names take the format <original\_key\_name>\_<partition\_number>.
* SQL Server allows you to create foreign key constraint from and to partitioned tables. PostgreSQL doesn’t support foreign keys referencing partitioned tables. Also, PostgreSQL doesn’t support foreign key references from a partitioned table to another table.
* SQL Server allows you to create indexes for partitioned tables. For PostgreSQL, an index should be created for each partition directly. Thus, indexes must be removed from their parent tables when migrating to PostgreSQL. The resulting index names take the format <original\_index\_name>\_<partition\_number>.
* PostgreSQL doesn’t support partitioned indexes.

## Migration considerations

Some things to consider when migrating a SQL Server schema to PostgreSQL:

* In PostgreSQL, all object’s names in a schema must be unique, including indexes. Index names must be unique in the schema of the base table. In SQL Server, an index name can be the same for different tables.

To ensure the uniqueness of index names, AWS SCT gives you the option to generate unique index names if your index names are not unique. To do this, choose the option **Generate unique index names** in the project properties. By default, this option is enabled. If this option is enabled, unique index names are created using the format IX\_table\_name\_index\_name. If this option is disabled, index names aren’t changed.

* A GOTO statement and a label can be used to change the order that statements are run in. Any Transact-SQL statements that follow a GOTO statement are skipped and processing continues at the label. GOTO statements and labels can be used anywhere within a procedure, batch, or statement block. GOTO statements can also be nested.

PostgreSQL doesn’t use GOTO statements. When AWS SCT converts code that contains a GOTO statement, it converts the statement to use a BEGIN…END or LOOP…END LOOP statement. You can find examples of how AWS SCT converts GOTO statements in the table following.

| **SQL Server GOTO statements and the converted PostgreSQL statements** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **SQL Server statement** | **PostgreSQL statement** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| BEGIN  ....  statement1;  ....  GOTO label1;  statement2;  ....  label1:  Statement3;  ....  END | BEGIN  label1:  BEGIN  ....  statement1;  ....  EXIT label1;  statement2;  ....  END;  Statement3;  ....  END |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| BEGIN  ....  statement1;  ....  label1:  statement2;  ....  GOTO label1;  statement3;  ....  statement4;  ....  END | BEGIN  ....  statement1;  ....  label1:  LOOP  statement2;  ....  CONTINUE label1;  EXIT label1;  END LOOP;  statement3;  ....  statement4;  ....  END |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| BEGIN  ....  statement1;  ....  label1:  statement2;  ....  statement3;  ....  statement4;  ....  END | BEGIN  ....  statement1;  ....  label1:  BEGIN  statement2;  ....  statement3;  ....  statement4;  ....  END;  END |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

* PostgreSQL doesn't support a MERGE statement. AWS SCT emulates the behavior of a MERGE statement in the following ways:
  + By INSERT ON CONFLICT construction.
  + By using the UPDATE FROM DML statement, such as MERGE without a WHEN NOT MATCHED clause.
  + By using CURSOR, such as with a MERGE with DELETE clause or by using a complex MERGE ON condition statement.
* AWS SCT can add database triggers to the object tree when Amazon RDS is the target.
* AWS SCT can add server-level triggers to the object tree when Amazon RDS is the target.
* SQL Server automatically creates and manages deleted and inserted tables. You can use these temporary, memory-resident tables to test the effects of certain data modifications and to set conditions for DML trigger actions. AWS SCT can convert the usage of these tables inside DML trigger statements.
* AWS SCT can add linked servers to the object tree when Amazon RDS is the target.
* When migrating from Microsoft SQL Server to PostgreSQL, the built-in SUSER\_SNAME function is converted as follows:
  + SUSER\_SNAME – Returns the login name associated with a security identification number (SID).
  + SUSER\_SNAME(<server\_user\_sid>) – Not supported.
  + SUSER\_SNAME() CURRENT\_USER – Returns the user name of the current execution context.
  + SUSER\_SNAME(NULL) – Returns NULL.
* Converting table-valued functions is supported. Table-valued functions return a table and can take the place of a table in a query.
* PATINDEX returns the starting position of the first occurrence of a pattern in a specified expression on all valid text and character data types. It returns zeros if the pattern is not found. When converting from SQL Server to Amazon RDS for PostgreSQL, AWS SCT replaces application code that uses PATINDEX with aws\_sqlserver\_ext.patindex(<pattern character>, <expression character varying>) .
* In SQL Server, a user-defined table type is a type that represents the definition of a table structure. You use a user-defined table type to declare table-value parameters for stored procedures or functions. You can also use a user-defined table type to declare table variables that you want to use in a batch or in the body of a stored procedure or function. AWS SCT emulated this type in PostgreSQL by creating a temporary table.

When converting from SQL Server to PostgreSQL, AWS SCT converts SQL Server system objects into recognizable objects in PostgreSQL. The following table shows how the system objects are converted.

| **MS SQL Server use cases** | **PostgreSQL substitution** |
| --- | --- |
| SYS.SCHEMAS | AWS\_SQLSERVER\_EXT.SYS\_SCHEMAS |
| SYS.TABLES | AWS\_SQLSERVER\_EXT.SYS\_TABLES |
| SYS.VIEWS | AWS\_SQLSERVER\_EXT.SYS\_VIEWS |
| SYS.ALL\_VIEWS | AWS\_SQLSERVER\_EXT.SYS\_ALL\_VIEWS |
| SYS.TYPES | AWS\_SQLSERVER\_EXT.SYS\_TYPES |
| SYS.COLUMNS | AWS\_SQLSERVER\_EXT.SYS\_COLUMNS |
| SYS.ALL\_COLUMNS | AWS\_SQLSERVER\_EXT.SYS\_ALL\_COLUMNS |
| SYS.FOREIGN\_KEYS | AWS\_SQLSERVER\_EXT.SYS\_FOREIGN\_KEYS |
| SYS.SYSFOREIGNKEYS | AWS\_SQLSERVER\_EXT.SYS\_SYSFOREIGNKEYS |
| SYS.FOREIGN\_KEY\_COLUMNS | AWS\_SQLSERVER\_EXT.SYS\_FOREIGN\_KEY\_COLUMNS |
| SYS.KEY\_CONSTRAINTS | AWS\_SQLSERVER\_EXT.SYS\_KEY\_CONSTRAINTS |
| SYS.IDENTITY\_COLUMNS | AWS\_SQLSERVER\_EXT.SYS\_IDENTITY\_COLUMNS |
| SYS.PROCEDURES | AWS\_SQLSERVER\_EXT.SYS\_PROCEDURES |
| SYS.INDEXES | AWS\_SQLSERVER\_EXT.SYS\_INDEXES |
| SYS.SYSINDEXES | AWS\_SQLSERVER\_EXT.SYS\_SYSINDEXES |
| SYS.OBJECTS | AWS\_SQLSERVER\_EXT.SYS\_OBJECTS |
| SYS.ALL\_OBJECTS | AWS\_SQLSERVER\_EXT.SYS\_ALL\_OBJECTS |
| SYS.SYSOBJECTS | AWS\_SQLSERVER\_EXT.SYS\_SYSOBJECTS |
| SYS.SQL\_MODULES | AWS\_SQLSERVER\_EXT.SYS\_SQL\_MODULES |
| SYS.DATABASES | AWS\_SQLSERVER\_EXT.SYS\_DATABASES |
| INFORMATION\_SCHEMA.SCHEMATA | AWS\_SQLSERVER\_EXT.INFORMATION\_SCHEMA\_SCHEMATA |
| INFORMATION\_SCHEMA.VIEWS | AWS\_SQLSERVER\_EXT.INFORMATION\_SCHEMA\_VIEWS |
| INFORMATION\_SCHEMA.TABLES | AWS\_SQLSERVER\_EXT.INFORMATION\_SCHEMA\_TABLES |
| INFORMATION\_SCHEMA.COLUMNS | AWS\_SQLSERVER\_EXT.INFORMATION\_SCHEMA\_COLUMNS |
| INFORMATION\_SCHEMA.CHECK\_CONSTRAINTS | AWS\_SQLSERVER\_EXT.INFORMATION\_SCHEMA\_CHECK\_CONSTRAINTS |
| INFORMATION\_SCHEMA.REFERENTIAL\_CONSTRAINTS | AWS\_SQLSERVER\_EXT.INFORMATION\_SCHEMA\_REFERENTIAL\_CONSTRAINTS |
| INFORMATION\_SCHEMA.TABLE\_CONSTRAINTS | AWS\_SQLSERVER\_EXT.INFORMATION\_SCHEMA\_TABLE\_CONSTRAINTS |
| INFORMATION\_SCHEMA.KEY\_COLUMN\_USAGE | AWS\_SQLSERVER\_EXT.INFORMATION\_SCHEMA\_KEY\_COLUMN\_USAGE |
| INFORMATION\_SCHEMA.CONSTRAINT\_TABLE\_USAGE | AWS\_SQLSERVER\_EXT.INFORMATION\_SCHEMA\_CONSTRAINT\_TABLE\_USAGE |
| INFORMATION\_SCHEMA.CONSTRAINT\_COLUMN\_USAGE | AWS\_SQLSERVER\_EXT.INFORMATION\_SCHEMA\_CONSTRAINT\_COLUMN\_USAGE |
| INFORMATION\_SCHEMA.ROUTINES | AWS\_SQLSERVER\_EXT.INFORMATION\_SCHEMA\_ROUTINES |
| SYS.SYSPROCESSES | AWS\_SQLSERVER\_EXT.SYS\_SYSPROCESSES |
| sys.system\_objects | AWS\_SQLSERVER\_EXT.SYS\_SYSTEM\_OBJECTS |

## PostgreSQL Equivalent for MS SQL server message broker and SSIS jobs

Amazon Aurora PostgreSQL-Compatible Edition (Aurora PostgreSQL) doesn’t provide a compatible solution to the SQL Server Service Broker. However, you can use DB Links and AWS Lambda to achieve similar functionality.

You can combine AWS Lambda with AWS SQS to reduce costs and remove some loads from the database into the AWS Lambda and Amazon Simple Queue Service (Amazon SQS). This will be much more efficient.

For example, you can create a table in each database and connect each database with a DB link to read the tables and process the data. For more information, see DB Links.

You can also use AWS Lambda to query a table from the database, process the data, and insert it to another database (even another database type). This approach is the best option for moving workloads out of the database to a less expensive instance type.

For even more decoupling and reducing workloads from the database, you can use Amazon SQS with Lambda.