Zero-Downtime MySQL to PostgreSQL Migration on Azure

Executive Summary

This guide provides a proven, step-by-step approach for migrating from MySQL to PostgreSQL on Azure with zero downtime using a hybrid strategy combining AWS DMS (for cross-engine support) with Azure-native services.

Migration Architecture Overview

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────────┐
│  MySQL Source   │───▶│    AWS DMS      │───▶│  PostgreSQL Target  │
│   (Azure DB)    │    │  (Migration)    │    │    (Azure DB)       │
└─────────────────┘    └─────────────────┘    └─────────────────────┘
        │                       ▲
        │              ┌─────────────────┐                │
        └─────────────▶│   Application   │────────────────┘
                       │  (Dual-Write)   │
                       └─────────────────┘
```

Phase 1: Prerequisites and Environment Preparation

1.1 Azure Infrastructure Setup

```bash
```

```
# Create resource group for migration
az group create --name mysql-to-pg-migration --location eastus

# Create Azure Database for PostgreSQL
az postgres flexible-server create \
  --resource-group mysql-to-pg-migration \
  --name myapp-postgresql \
  --admin-user pgadmin \
  --admin-password 'SecurePassword123!' \
  --sku-name Standard_D4s_v3 \
  --tier GeneralPurpose \
  --storage-size 512 \
  --version 14

# Enable logical replication on source MySQL
az mysql flexible-server parameter set \
  --resource-group mysql-to-pg-migration \
  --server-name myapp-mysql \
  --name binlog_format \
  --value ROW

az mysql flexible-server parameter set \
  --resource-group mysql-to-pg-migration \
  --server-name myapp-mysql \
  --name binlog_row_image \
  --value FULL
```

1.2 Network Configuration

```
bash
```

```
# Create VNet for secure communication
az network vnet create \
  --resource-group mysql-to-pg-migration \
  --name migration-vnet \
  --address-prefix 10.0.0.0/16 \
  --subnet-name database-subnet \
  --subnet-prefix 10.0.1.0/24

# Configure private endpoints
az network private-endpoint create \
  --resource-group mysql-to-pg-migration \
  --name mysql-private-endpoint \
  --vnet-name migration-vnet \
  --subnet database-subnet \
  --private-connection-resource-id $MYSQL_RESOURCE_ID \
  --connection-name mysql-connection \
  --group-id mysqlServer
```

1.3 MySQL Source Preparation

```sql
-- Create replication user with necessary privileges
CREATE USER 'replication_user'@'%' IDENTIFIED BY 'ReplicationPass123!';
GRANT REPLICATION SLAVE ON *.* TO 'replication_user'@'%';
GRANT SELECT ON *.* TO 'replication_user'@'%';
GRANT SHOW VIEW ON *.* TO 'replication_user'@'%';
GRANT TRIGGER ON *.* TO 'replication_user'@'%';
FLUSH PRIVILEGES;

-- Enable binary logging (if not already enabled)
-- This requires server restart in Azure MySQL
SET GLOBAL log_bin = ON;
SET GLOBAL binlog_format = 'ROW';
SET GLOBAL binlog_row_image = 'FULL';

-- Verify binary log configuration
SHOW VARIABLES LIKE 'log_bin';
SHOW VARIABLES LIKE 'binlog_format';
SHOW VARIABLES LIKE 'binlog_row_image';
```

Phase 2: Schema Migration and Conversion

2.1 Schema Analysis and Export

```bash
```

```
#!/bin/bash
# schema_export.sh

# Export MySQL schema without data
mysqldump --no-data \
    --routines \
    --triggers \
    --events \
    --single-transaction \
    --host=myapp-mysql.mysql.database.azure.com \
    --user=mysqladmin \
    --password \
    myapp_database > mysql_schema.sql

# Export data for validation
mysqldump --no-create-info \
    --single-transaction \
    --where="1 limit 1000" \
    --host=myapp-mysql.mysql.database.azure.com \
    --user=mysqladmin \
    --password \
    myapp_database > mysql_sample_data.sql
```

2.2 Schema Conversion Script

```
python
```

```python
#!/usr/bin/env python3
# mysql_to_postgresql_converter.py

import re
import sys

def convert_mysql_to_postgresql(mysql_schema):
    """Convert MySQL schema to PostgreSQL compatible schema"""

    conversions = {
        # Data type conversions
        r'\bTINYINT\(1\)\b': 'BOOLEAN',
        r'\bTINYINT\b': 'SMALLINT',
        r'\bBIGINT\(\d+\)\s+UNSIGNED': 'BIGINT',
        r'\bINT\(\d+\)\s+UNSIGNED': 'BIGINT',
        r'\bINT\(\d+\)': 'INTEGER',
        r'\bDATETIME\b': 'TIMESTAMP',
        r'\bTEXT\b': 'TEXT',
        r'\bLONGTEXT\b': 'TEXT',
        r'\bMEDIUMTEXT\b': 'TEXT',
        r'\bVARCHAR\((\d+)\)\s+COLLATE\s+\w+': r'VARCHAR(\1)',

        # AUTO_INCREMENT conversion
        r'\bAUTO_INCREMENT\b': 'GENERATED ALWAYS AS IDENTITY',

        # Engine and charset removal
        r'\s*ENGINE\s*=\s*\w+': '',
        r'\s*DEFAULT\s+CHARSET\s*=\s*\w+': '',
        r'\s*COLLATE\s*=\s*\w+': '',

        # Backtick removal
        r'`([^`]+)`': r'"\1"',

        # Index syntax
        r'\bKEY\s+`?(\w+)`?\s*\(([^)]+)\)': r'CREATE INDEX \1 ON table_name (\2);',
        r'\bUNIQUE\s+KEY\s+`?(\w+)`?\s*\(([^)]+)\)': r'CREATE UNIQUE INDEX \1 ON table_name (\2);',
    }

    postgresql_schema = mysql_schema

    for pattern, replacement in conversions.items():
        postgresql_schema = re.sub(pattern, replacement, postgresql_schema, flags=re.IGNORECASE)

    # Handle ENUM types
    enum_pattern = r"ENUM\s*\(([^)]+)\)"
    def enum_replacement(match):
        values = match.group(1)
        return f"VARCHAR(255) CHECK (column_name IN ({values}))"

    postgresql_schema = re.sub(enum_pattern, enum_replacement, postgresql_schema, flags=re.IGNORECASE)

    return postgresql_schema

def main():
    if len(sys.argv) != 3:
        print("Usage: python3 mysql_to_postgresql_converter.py input.sql output.sql")
        sys.exit(1)

    input_file = sys.argv[1]
    output_file = sys.argv[2]

    with open(input_file, 'r') as f:
        mysql_schema = f.read()

    postgresql_schema = convert_mysql_to_postgresql(mysql_schema)

    with open(output_file, 'w') as f:
        f.write(postgresql_schema)

    print(f"Schema conversion completed. Output saved to {output_file}")

if __name__ == "__main__":
    main()
```

2.3 Execute Schema Creation

```bash
# Convert schema
python3 mysql_to_postgresql_converter.py mysql_schema.sql postgresql_schema.sql

# Create schema in PostgreSQL
PGPASSWORD='SecurePassword123!' psql \
  -h myapp-postgresql.postgres.database.azure.com \
  -U pgadmin \
  -d myapp_database \
  -f postgresql_schema.sql
```

Phase 3: AWS DMS Setup for Cross-Engine Migration

3.1 AWS DMS Infrastructure

bash

```bash
# Create AWS DMS replication instance
aws dms create-replication-instance \
  --replication-instance-identifier mysql-to-postgresql-migration \
  --replication-instance-class dms.t3.medium \
  --allocated-storage 20 \
  --vpc-security-group-ids sg-xxxxxxxxx \
  --replication-subnet-group-identifier default-vpc-xxxxxxxx \
  --publicly-accessible \
  --multi-az false

# Create source endpoint (MySQL on Azure)
aws dms create-endpoint \
  --endpoint-identifier mysql-source-azure \
  --endpoint-type source \
  --engine-name mysql \
  --server-name myapp-mysql.mysql.database.azure.com \
  --port 3306 \
  --username mysqladmin \
  --password 'MySQLPassword123!' \
  --database-name myapp_database

# Create target endpoint (PostgreSQL on Azure)
aws dms create-endpoint \
  --endpoint-identifier postgresql-target-azure \
  --endpoint-type target \
  --engine-name postgres \
  --server-name myapp-postgresql.postgres.database.azure.com \
  --port 5432 \
  --username pgadmin \
  --password 'SecurePassword123!' \
  --database-name myapp_database
```

3.2 DMS Task Configuration

json

```
{
 "rules": [
  {
   "rule-type": "selection",
   "rule-id": "1",
   "rule-name": "1",
   "object-locator": {
    "schema-name": "myapp_database",
    "table-name": "%"
   },
   "rule-action": "include",
   "filters": []
  },
  {
   "rule-type": "transformation",
   "rule-id": "2",
   "rule-name": "2",
   "rule-target": "column",
   "object-locator": {
    "schema-name": "myapp_database",
    "table-name": "%",
    "column-name": "%"
   },
   "rule-action": "convert-lowercase"
  }
 ]
}
```

bash

```bash
# Create migration task
aws dms create-replication-task \
  --replication-task-identifier mysql-to-postgresql-continuous \
  --source-endpoint-arn arn:aws:dms:us-east-1:account:endpoint:mysql-source-azure \
  --target-endpoint-arn arn:aws:dms:us-east-1:account:endpoint:postgresql-target-azure \
  --replication-instance-arn arn:aws:dms:us-east-1:account:rep:mysql-to-postgresql-migration \
  --migration-type full-load-and-cdc \
  --table-mappings file://table-mappings.json \
  --replication-task-settings file://task-settings.json
```

Phase 4: Alternative CDC Setup (Debezium + Kafka)

4.1 Kafka and Debezium Setup

yaml

```yaml
# docker-compose.yml for Kafka + Debezium
version: '3.8'
services:
 zookeeper:
  image: confluentinc/cp-zookeeper:latest
  environment:
   ZOOKEEPER_CLIENT_PORT: 2181
   ZOOKEEPER_TICK_TIME: 2000

 kafka:
  image: confluentinc/cp-kafka:latest
  depends_on:
   - zookeeper
  ports:
   - "9092:9092"
  environment:
   KAFKA_BROKER_ID: 1
   KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
   KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
   KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1

 debezium:
  image: debezium/connect:latest
  depends_on:
   - kafka
  ports:
   - "8083:8083"
  environment:
   BOOTSTRAP_SERVERS: kafka:9092
   GROUP_ID: 1
   CONFIG_STORAGE_TOPIC: debezium_configs
   OFFSET_STORAGE_TOPIC: debezium_offsets
   STATUS_STORAGE_TOPIC: debezium_statuses
```

4.2 Debezium MySQL Connector Configuration

```
json
```

```
{
 "name": "mysql-postgres-connector",
 "config": {
  "connector.class": "io.debezium.connector.mysql.MySqlConnector",
  "database.hostname": "myapp-mysql.mysql.database.azure.com",
  "database.port": "3306",
  "database.user": "replication_user",
  "database.password": "ReplicationPass123!",
  "database.server.id": "12345",
  "database.server.name": "azure-mysql-server",
  "database.include.list": "myapp_database",
  "table.include.list": "myapp_database.users,myapp_database.orders,myapp_database.products",
  "database.history.kafka.bootstrap.servers": "kafka:9092",
  "database.history.kafka.topic": "dbhistory.azure-mysql-server",
  "include.schema.changes": "true",
  "transforms": "unwrap",
  "transforms.unwrap.type": "io.debezium.transforms.ExtractNewRecordState",
  "transforms.unwrap.drop.tombstones": "false",
  "key.converter": "org.apache.kafka.connect.json.JsonConverter",
  "value.converter": "org.apache.kafka.connect.json.JsonConverter",
  "key.converter.schemas.enable": "false",
  "value.converter.schemas.enable": "false"
 }
}
```

4.3 Kafka to PostgreSQL Sync Service

```python
```

```python
#!/usr/bin/env python3
# kafka_to_postgres_sync.py

import json
import asyncio
import asyncpg
from kafka import KafkaConsumer
from typing import Dict, Any
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class KafkaPostgreSQLSync:
    def __init__(self, kafka_config: Dict, postgres_config: Dict):
        self.kafka_config = kafka_config
        self.postgres_config = postgres_config
        self.pool = None

    async def init_postgres_pool(self):
        """Initialize PostgreSQL connection pool"""
        self.pool = await asyncpg.create_pool(
            host=self.postgres_config['host'],
            port=self.postgres_config['port'],
            user=self.postgres_config['user'],
            password=self.postgres_config['password'],
            database=self.postgres_config['database'],
            min_size=5,
            max_size=20
        )

    def create_kafka_consumer(self) -> KafkaConsumer:
        """Create Kafka consumer"""
        return KafkaConsumer(
            'azure-mysql-server.myapp_database.users',
            'azure-mysql-server.myapp_database.orders',
            'azure-mysql-server.myapp_database.products',
            bootstrap_servers=self.kafka_config['bootstrap_servers'],
            value_deserializer=lambda m: json.loads(m.decode('utf-8')),
            key_deserializer=lambda m: json.loads(m.decode('utf-8')) if m else None,
            auto_offset_reset='earliest',
            enable_auto_commit=True,
            group_id='postgres-sync-group'
        )

    async def process_message(self, message):
        """Process Kafka message and sync to PostgreSQL"""
        try:
            # Extract table name from topic
            topic_parts = message.topic.split('.')
            table_name = topic_parts[-1]

            # Get message payload
            payload = message.value
            operation = payload.get('op')

            if operation == 'c':  # Create
                await self.handle_insert(table_name, payload['after'])
            elif operation == 'u':  # Update
                await self.handle_update(table_name, payload['before'], payload['after'])
            elif operation == 'd':  # Delete
                await self.handle_delete(table_name, payload['before'])
            elif operation == 'r':  # Read (initial snapshot)
                await self.handle_insert(table_name, payload['after'])

        except Exception as e:
            logger.error(f"Error processing message: {e}")

    async def handle_insert(self, table_name: str, data: Dict[str, Any]):
        """Handle INSERT operation"""
        async with self.pool.acquire() as conn:
            columns = list(data.keys())
            values = list(data.values())
            placeholders = [f'${i+1}' for i in range(len(values))]

            query = f"""
                INSERT INTO {table_name} ({', '.join(columns)})
                VALUES ({', '.join(placeholders)})
                ON CONFLICT DO NOTHING
            """

            await conn.execute(query, *values)
            logger.info(f"Inserted record into {table_name}")

    async def handle_update(self, table_name: str, before: Dict[str, Any], after: Dict[str, Any]):
        """Handle UPDATE operation"""
        async with self.pool.acquire() as conn:
            # Assume 'id' is primary key
            where_clause = "id = $1"
            set_clauses = []
            values = [after['id']]

            param_counter = 2
            for key, value in after.items():
                if key != 'id':
                    set_clauses.append(f"{key} = ${param_counter}")
                    values.append(value)
                    param_counter += 1

            query = f"""
                UPDATE {table_name}
                SET {', '.join(set_clauses)}
                WHERE {where_clause}
            """

            await conn.execute(query, *values)
            logger.info(f"Updated record in {table_name}")

    async def handle_delete(self, table_name: str, data: Dict[str, Any]):
        """Handle DELETE operation"""
        async with self.pool.acquire() as conn:
            query = f"DELETE FROM {table_name} WHERE id = $1"
            await conn.execute(query, data['id'])
            logger.info(f"Deleted record from {table_name}")

    async def start_sync(self):
        """Start the synchronization process"""
        await self.init_postgres_pool()
        consumer = self.create_kafka_consumer()

        logger.info("Starting Kafka to PostgreSQL sync...")

        try:
            for message in consumer:
                await self.process_message(message)
        except KeyboardInterrupt:
            logger.info("Stopping sync...")
        finally:
            consumer.close()
            await self.pool.close()

# Configuration
kafka_config = {
    'bootstrap_servers': ['localhost:9092']
}

postgres_config = {
    'host': 'myapp-postgresql.postgres.database.azure.com',
    'port': 5432,
    'user': 'pgladmin',
    'password': 'SecurePassword123!',
    'database': 'myapp_database'
}

# Run the sync service
async def main():
    sync_service = KafkaPostgreSQLSync(kafka_config, postgres_config)
    await sync_service.start_sync()

if __name__ == "__main__":
    asyncio.run(main())
```

Phase 5: Application Dual-Write Strategy

5.1 Database Abstraction Layer

python

```
#!/usr/bin/env python3
# database_abstraction.py

import asyncio
import asyncpg
import aiomysql
from typing import Optional, Dict, Any, List
import logging
from contextlib import asynccontextmanager

logger = logging.getLogger(__name__)

class DatabaseManager:
  def __init__(self, mysql_config: Dict, postgres_config: Dict):
    self.mysql_config = mysql_config
    self.postgres_config = postgres_config
    self.mysql_pool = None
    self.postgres_pool = None
    self.dual_write_enabled = False
    self.primary_db = 'mysql'  # Start with MySQL as primary

  async def init_connections(self):
    """Initialize database connections"""
    # MySQL connection
    self.mysql_pool = await aiomysql.create_pool(
      host=self.mysql_config['host'],
      port=self.mysql_config['port'],
      user=self.mysql_config['user'],
      password=self.mysql_config['password'],
      db=self.mysql_config['database'],
      minsize=5,
      maxsize=20,
      autocommit=False
    )

    # PostgreSQL connection
    self.postgres_pool = await asyncpg.create_pool(
      host=self.postgres_config['host'],
      port=self.postgres_config['port'],
      user=self.postgres_config['user'],
      password=self.postgres_config['password'],
      database=self.postgres_config['database'],
      min_size=5,
      max_size=20
    )
```

```
        logger.info("Database connections initialized")

    def enable_dual_write(self):
        """Enable dual-write mode"""
        self.dual_write_enabled = True
        logger.info("Dual-write mode enabled")

    def switch_primary(self, primary: str):
        """Switch primary database"""
        if primary in ['mysql', 'postgres']:
            self.primary_db = primary
            logger.info(f"Primary database switched to {primary}")
        else:
            raise ValueError("Primary must be 'mysql' or 'postgres'")

    async def execute_read(self, query: str, params: Optional[List] = None) -> List[Dict[str, Any]]:
        """Execute read query on primary database"""
        if self.primary_db == 'mysql':
            return await self._execute_mysql_read(query, params)
        else:
            return await self._execute_postgres_read(query, params)

    async def execute_write(self, query: str, params: Optional[List] = None) -> bool:
        """Execute write query with dual-write support"""
        success = True

        if self.primary_db == 'mysql':
            success &= await self._execute_mysql_write(query, params)
            if self.dual_write_enabled:
                # Convert MySQL query to PostgreSQL
                pg_query = self._convert_mysql_to_postgres(query)
                success &= await self._execute_postgres_write(pg_query, params)
        else:
            success &= await self._execute_postgres_write(query, params)
            if self.dual_write_enabled:
                # Convert PostgreSQL query to MySQL
                mysql_query = self._convert_postgres_to_mysql(query)
                success &= await self._execute_mysql_write(mysql_query, params)

        return success

    async def _execute_mysql_read(self, query: str, params: Optional[List] = None) -> List[Dict[str, Any]]:
        """Execute MySQL read query"""
        async with self.mysql_pool.acquire() as conn:
            async with conn.cursor(aiomysql.DictCursor) as cursor:
                await cursor.execute(query, params or [])
                results = await cursor.fetchall()
                return list(results)

    async def _execute_mysql_write(self, query: str, params: Optional[List] = None) -> bool:
        """Execute MySQL write query"""
        try:
            async with self.mysql_pool.acquire() as conn:
                async with conn.cursor() as cursor:
                    await cursor.execute(query, params or [])
                    await conn.commit()
                    return True
        except Exception as e:
            logger.error(f"MySQL write error: {e}")
            return False

    async def _execute_postgres_read(self, query: str, params: Optional[List] = None) -> List[Dict[str, Any]]:
        """Execute PostgreSQL read query"""
        async with self.postgres_pool.acquire() as conn:
            rows = await conn.fetch(query, *(params or []))
            return [dict(row) for row in rows]

    async def _execute_postgres_write(self, query: str, params: Optional[List] = None) -> bool:
        """Execute PostgreSQL write query"""
        try:
            async with self.postgres_pool.acquire() as conn:
                async with conn.transaction():
                    await conn.execute(query, *(params or []))
                    return True
        except Exception as e:
            logger.error(f"PostgreSQL write error: {e}")
            return False

    def _convert_mysql_to_postgres(self, query: str) -> str:
        """Convert MySQL query syntax to PostgreSQL"""
        # Basic conversions
        conversions = {
            'LIMIT %s, %s': 'LIMIT %s OFFSET %s',
            '`': '"',
            'AUTO_INCREMENT': 'GENERATED ALWAYS AS IDENTITY'
        }

        converted_query = query
        for mysql_syntax, postgres_syntax in conversions.items():
            converted_query = converted_query.replace(mysql_syntax, postgres_syntax)

        return converted_query

    def _convert_postgres_to_mysql(self, query: str) -> str:
        """Convert PostgreSQL query syntax to MySQL"""
        # Basic conversions
        conversions = {
            'LIMIT %s OFFSET %s': 'LIMIT %s, %s',
            '"': '`',
            'GENERATED ALWAYS AS IDENTITY': 'AUTO_INCREMENT'
        }

        converted_query = query
        for postgres_syntax, mysql_syntax in conversions.items():
            converted_query = converted_query.replace(postgres_syntax, mysql_syntax)

        return converted_query

    async def validate_data_consistency(self, table_name: str) -> Dict[str, Any]:
        """Validate data consistency between MySQL and PostgreSQL"""
        # Count rows
        mysql_count_query = f"SELECT COUNT(*) as count FROM `{table_name}`"
        postgres_count_query = f'SELECT COUNT(*) as count FROM "{table_name}"'

        mysql_count = await self._execute_mysql_read(mysql_count_query)
        postgres_count = await self._execute_postgres_read(postgres_count_query)

        # Checksum comparison (simplified)
        mysql_checksum_query = f"""
            SELECT BIT_XOR(CRC32(CONCAT_WS('|',
                COALESCE(id, ''),
                COALESCE(name, ''),
                COALESCE(email, '')
            ))) as checksum
            FROM `{table_name}`
        """

        postgres_checksum_query = f"""
            SELECT BIT_XOR(
                ('x' || substr(md5(COALESCE(id::text, '') || '|' ||
                        COALESCE(name, '') || '|' ||
                        COALESCE(email, '')), 1, 8))::bit(32)::int
            ) as checksum
            FROM "{table_name}"
        """

        try:
            mysql_checksum = await self._execute_mysql_read(mysql_checksum_query)
            postgres_checksum = await self._execute_postgres_read(postgres_checksum_query)
        except Exception as e:
            logger.warning(f"Checksum comparison failed: {e}")
            mysql_checksum = [{'checksum': 0}]
            postgres_checksum = [{'checksum': 0}]

        return {
            'table_name': table_name,
            'mysql_count': mysql_count[0]['count'],
            'postgres_count': postgres_count[0]['count'],
            'count_match': mysql_count[0]['count'] == postgres_count[0]['count'],
            'mysql_checksum': mysql_checksum[0]['checksum'],
            'postgres_checksum': postgres_checksum[0]['checksum'],
            'checksum_match': mysql_checksum[0]['checksum'] == postgres_checksum[0]['checksum']
        }

    async def close_connections(self):
        """Close database connections"""
        if self.mysql_pool:
            self.mysql_pool.close()
            await self.mysql_pool.wait_closed()

        if self.postgres_pool:
            await self.postgres_pool.close()

        logger.info("Database connections closed")
```

5.2 Application Integration Example

python

```python
#!/usr/bin/env python3
# application_service.py

import asyncio
from database_abstraction import DatabaseManager
from typing import Optional, Dict, Any, List

class UserService:
    def __init__(self, db_manager: DatabaseManager):
        self.db = db_manager

    async def create_user(self, user_data: Dict[str, Any]) -> Optional[int]:
        """Create a new user"""
        query = """
            INSERT INTO users (name, email, created_at)
            VALUES (%s, %s, NOW())
        """

        success = await self.db.execute_write(
            query,
            [user_data['name'], user_data['email']]
        )

        if success:
            # Get the ID (this is simplified)
            result = await self.db.execute_read(
                "SELECT LAST_INSERT_ID() as id" if self.db.primary_db == 'mysql'
                else "SELECT lastval() as id"
            )
            return result[0]['id']

        return None

    async def get_user(self, user_id: int) -> Optional[Dict[str, Any]]:
        """Get user by ID"""
        query = "SELECT * FROM users WHERE id = %s"
        results = await self.db.execute_read(query, [user_id])
        return results[0] if results else None

    async def update_user(self, user_id: int, user_data: Dict[str, Any]) -> bool:
        """Update user"""
        query = """
            UPDATE users
            SET name = %s, email = %s, updated_at = NOW()
            WHERE id = %s
        """
        return await self.db.execute_write(
            query,
            [user_data['name'], user_data['email'], user_id]
        )

    async def delete_user(self, user_id: int) -> bool:
        """Delete user"""
        query = "DELETE FROM users WHERE id = %s"
        return await self.db.execute_write(query, [user_id])

# Usage example
async def main():
    # Database configurations
    mysql_config = {
        'host': 'myapp-mysql.mysql.database.azure.com',
        'port': 3306,
        'user': 'mysqladmin',
        'password': 'MySQLPassword123!',
        'database': 'myapp_database'
    }

    postgres_config = {
        'host': 'myapp-postgresql.postgres.database.azure.com',
        'port': 5432,
        'user': 'pgadmin',
        'password': 'SecurePassword123!',
        'database': 'myapp_database'
    }

    # Initialize database manager
    db_manager = DatabaseManager(mysql_config, postgres_config)
    await db_manager.init_connections()

    # Enable dual-write mode during migration
    db_manager.enable_dual_write()

    # Initialize services
    user_service = UserService(db_manager)

    # Test operations
    user_id = await user_service.create_user({
        'name': 'John Doe',
        'email': 'john@example.com'
    })

    print(f"Created user with ID: {user_id}")
    # Validate consistency
    consistency_report = await db_manager.validate_data_consistency('users')
    print(f"Consistency report: {consistency_report}")

    # After migration is complete, switch to PostgreSQL
    db_manager.switch_primary('postgres')

    # Continue operations on PostgreSQL
    user = await user_service.get_user(user_id)
    print(f"Retrieved user: {user}")

    await db_manager.close_connections()

if __name__ == "__main__":
    asyncio.run(main())
```

Phase 6: Testing and Validation

6.1 Data Validation Scripts

```
bash
```

```
#!/bin/bash
# validation_suite.sh

echo "Starting comprehensive data validation..."

# Row count validation
python3 << EOF
import asyncio
import asyncpg
import aiomysql

async def validate_row_counts():
    # MySQL connection
    mysql_conn = await aiomysql.connect(
        host='myapp-mysql.mysql.database.azure.com',
        port=3306,
        user='mysqladmin',
        password='MySQLPassword123!',
        db='myapp_database'
    )

    # PostgreSQL connection
    pg_conn = await asyncpg.connect(
        host='myapp-postgresql.postgres.database.azure.com',
        port=5432,
        user='pgladmin',
        password='SecurePassword123!',
        database='myapp_database'
    )

    tables = ['users', 'orders', 'products', 'order_items']

    for table in tables:
        # MySQL count
        async with mysql_conn.cursor() as cursor:
            await cursor.execute(f"SELECT COUNT(*) FROM {table}")
            mysql_count = await cursor.fetchone()

        # PostgreSQL count
        pg_count = await pg_conn.fetchval(f'SELECT COUNT(*) FROM "{table}"')

        print(f"Table {table}:")
        print(f"  MySQL: {mysql_count[0]}")
        print(f"  PostgreSQL: {pg_count}")
        print(f"  Match: {mysql_count[0] == pg_count}")
        print()
    mysql_conn.close()
    await pg_conn.close()

asyncio.run(validate_row_counts())
EOF

echo "Row count validation completed."

# Schema validation
echo "Validating schema differences..."
python3 << EOF
import asyncio
import asyncpg
import aiomysql

async def validate_schemas():
    # MySQL schema info
    mysql_conn = await aiomysql.connect(
        host='myapp-mysql.mysql.database.azure.com',
        port=3306,
        user='mysqladmin',
        password='MySQLPassword123!',
        db='myapp_database'
    )

    # PostgreSQL schema info
    pg_conn = await asyncpg.connect(
        host='myapp-postgresql.postgres.database.azure.com',
        port=5432,
        user='pgadmin',
        password='SecurePassword123!',
        database='myapp_database'
    )

    # Get MySQL table structure
    async with mysql_conn.cursor(aiomysql.DictCursor) as cursor:
        await cursor.execute("""
            SELECT TABLE_NAME, COLUMN_NAME, DATA_TYPE, IS_NULLABLE, COLUMN_DEFAULT
            FROM INFORMATION_SCHEMA.COLUMNS
            WHERE TABLE_SCHEMA = 'myapp_database'
            ORDER BY TABLE_NAME, ORDINAL_POSITION
        """)
        mysql_schema = await cursor.fetchall()

    # Get PostgreSQL table structure
    pg_schema = await pg_conn.fetch("""
        SELECT table_name, column_name, data_type, is_nullable, column_default
        FROM information_schema.columns
        WHERE table_schema = 'public'
        ORDER BY table_name, ordinal_position
    """)

    print("Schema comparison:")
    print("MySQL tables:", {row['TABLE_NAME'] for row in mysql_schema})
    print("PostgreSQL tables:", {row['table_name'] for row in pg_schema})

    mysql_conn.close()
    await pg_conn.close()

asyncio.run(validate_schemas())
EOF

echo "Schema validation completed."
```

6.2 Performance Testing

```
python
```

```
#!/usr/bin/env python3
# performance_test.py

import asyncio
import time
import statistics
from database_abstraction import DatabaseManager

async def performance_test():
    """Run performance comparison between MySQL and PostgreSQL"""

    mysql_config = {
        'host': 'myapp-mysql.mysql.database.azure.com',
        'port': 3306,
        'user': 'mysqladmin',
        'password': 'MySQLPassword123!',
        'database': 'myapp_database'
    }

    postgres_config = {
        'host': 'myapp-postgresql.postgres.database.azure.com',
        'port': 5432,
        'user': 'pgadmin',
        'password': 'SecurePassword123!',
        'database': 'myapp_database'
    }

    db_manager = DatabaseManager(mysql_config, postgres_config)
    await db_manager.init_connections()

    # Test queries
    test_queries = [
        "SELECT COUNT(*) FROM users",
        "SELECT * FROM users WHERE id = 1",
        "SELECT u.*, COUNT(o.id) as order_count FROM users u LEFT JOIN orders o ON u.id = o.user_id GROUP BY u.id L
        "SELECT * FROM orders WHERE created_at >= DATE_SUB(NOW(), INTERVAL 30 DAY)"
    ]

    results = {}

    for query in test_queries:
        print(f"Testing query: {query[:50]}...")

        # Test MySQL
        mysql_times = []
        for _ in range(10):
            start_time = time.time()
            db_manager.primary_db = 'mysql'
            await db_manager.execute_read(query)
            mysql_times.append(time.time() - start_time)

        # Test PostgreSQL
        postgres_times = []
        for _ in range(10):
            start_time = time.time()
            db_manager.primary_db = 'postgres'
            await db_manager.execute_read(query)
            postgres_times.append(time.time() - start_time)

        results[query] = {
            'mysql_avg': statistics.mean(mysql_times),
            'mysql_median': statistics.median(mysql_times),
            'postgres_avg': statistics.mean(postgres_times),
            'postgres_median': statistics.median(postgres_times)
        }

    # Print results
    print("\nPerformance Test Results:")
    print("=" * 80)

    for query, metrics in results.items():
        print(f"\nQuery: {query[:50]}...")
        print(f"MySQL    - Avg: {metrics['mysql_avg']:.4f}s, Median: {metrics['mysql_median']:.4f}s")
        print(f"PostgreSQL - Avg: {metrics['postgres_avg']:.4f}s, Median: {metrics['postgres_median']:.4f}s")

        improvement = ((metrics['mysql_avg'] - metrics['postgres_avg']) / metrics['mysql_avg']) * 100
        print(f"Performance change: {improvement:+.2f}%")

    await db_manager.close_connections()

if __name__ == "__main__":
    asyncio.run(performance_test())
```

Phase 7: Cutover Process

7.1 Pre-Cutover Checklist

bash

```
#!/bin/bash
# pre_cutover_checklist.sh

echo "Pre-Cutover Validation Checklist"
echo "================================="

# 1. Verify replication lag
echo "1. Checking replication lag..."
aws dms describe-replication-tasks \
    --filters Name=replication-task-id,Values=mysql-to-postgresql-continuous \
    --query 'ReplicationTasks[0].ReplicationTaskStats'

# 2. Validate data consistency
echo "2. Running data consistency checks..."
python3 validation_suite.sh

# 3. Test PostgreSQL performance
echo "3. Running performance tests..."
python3 performance_test.py

# 4. Verify backup status
echo "4. Checking backup status..."
az postgres flexible-server backup list \
    --resource-group mysql-to-pg-migration \
    --server-name myapp-postgresql

# 5. Test application connectivity
echo "5. Testing application connectivity..."
python3 << EOF
import asyncio
from database_abstraction import DatabaseManager

async def test_connectivity():
    mysql_config = {
        'host': 'myapp-mysql.mysql.database.azure.com',
        'port': 3306,
        'user': 'mysqladmin',
        'password': 'MySQLPassword123!',
        'database': 'myapp_database'
    }

    postgres_config = {
        'host': 'myapp-postgresql.postgres.database.azure.com',
        'port': 5432,
        'user': 'pgadmin',
        'password': 'SecurePassword123!',
        'database': 'myapp_database'
    }

    db_manager = DatabaseManager(mysql_config, postgres_config)
    await db_manager.init_connections()

    # Test both connections
    try:
        mysql_result = await db_manager._execute_mysql_read("SELECT 1 as test")
        print(f"MySQL connection: ✓ {mysql_result}")
    except Exception as e:
        print(f"MySQL connection: ✗ {e}")

    try:
        postgres_result = await db_manager._execute_postgres_read("SELECT 1 as test")
        print(f"PostgreSQL connection: ✓ {postgres_result}")
    except Exception as e:
        print(f"PostgreSQL connection: ✗ {e}")

    await db_manager.close_connections()

asyncio.run(test_connectivity())
EOF

echo "Pre-cutover validation completed."
```

7.2 Cutover Script

bash

```
#!/bin/bash
# cutover_script.sh

set -e  # Exit on any error

echo "Starting MySQL to PostgreSQL Cutover Process"
echo "=============================================="

# Configuration
MAINTENANCE_START=$(date -u +"%Y-%m-%d %H:%M:%S UTC")
ROLLBACK_TIMEOUT=3600  # 1 hour rollback window

echo "Cutover started at: $MAINTENANCE_START"

# 1. Enable maintenance mode
echo "1. Enabling maintenance mode..."
# Update load balancer to show maintenance page
az network application-gateway http-settings update \
    --resource-group mysql-to-pg-migration \
    --gateway-name myapp-gateway \
    --name maintenance-settings \
    --path /maintenance.html

# 2. Stop application writes
echo "2. Stopping application writes..."
# Scale down application instances or enable read-only mode
kubectl scale deployment myapp --replicas=0

# Wait for active connections to drain
echo "Waiting 30 seconds for connections to drain..."
sleep 30

# 3. Verify replication is caught up
echo "3. Verifying replication lag..."
python3 << EOF
import boto3
import time

dms_client = boto3.client('dms')

while True:
    response = dms_client.describe_replication-tasks(
        Filters=[
            {
                'Name': 'replication-task-id',
                'Values': ['mysql-to-postgresql-continuous']
            }
        ]
    )

    task = response['ReplicationTasks'][0]
    stats = task['ReplicationTaskStats']

    lag_seconds = stats.get('ElapsedTimeMillis', 0) / 1000
    print(f"Current replication lag: {lag_seconds} seconds")

    if lag_seconds < 5:  # Less than 5 seconds lag
        print("Replication is caught up!")
        break

    time.sleep(10)
EOF

# 4. Final data validation
echo "4. Running final data validation..."
python3 validation_suite.sh

# 5. Stop replication
echo "5. Stopping replication..."
aws dms stop-replication-task \
    --replication-task-arn arn:aws:dms:us-east-1:account:task:mysql-to-postgresql-continuous

# 6. Switch application to PostgreSQL
echo "6. Switching application to PostgreSQL..."

# Update application configuration
kubectl create configmap myapp-config \
    --from-literal=DATABASE_URL="postgresql://pgadmin:SecurePassword123!@myapp-postgresql.postgres.databas
    --dry-run=client -o yaml | kubectl apply -f -

# Update deployment
kubectl set env deployment/myapp PRIMARY_DB=postgres
kubectl set env deployment/myapp DUAL_WRITE_ENABLED=false

# Scale up application
kubectl scale deployment myapp --replicas=3

# 7. Test application functionality
echo "7. Testing application functionality..."
sleep 60  # Wait for pods to be ready

# Health check
for i in {1..10}; do
```

```
  HTTP_STATUS=$(curl -s -o /dev/null -w "%{http_code}" https://myapp.example.com/health)
  if [ "$HTTP_STATUS" = "200" ]; then
    echo "Application health check passed"
    break
  else
    echo "Health check failed (attempt $i/10): HTTP $HTTP_STATUS"
    if [ $i -eq 10 ]; then
      echo "Health checks failed, initiating rollback..."
      bash rollback_script.sh
      exit 1
    fi
    sleep 10
  fi
done

# 8. Disable maintenance mode
echo "8. Disabling maintenance mode..."
az network application-gateway http-settings update \
  --resource-group mysql-to-pg-migration \
  --gateway-name myapp-gateway \
  --name default-settings \
  --path /

# 9. Monitor for issues
echo "9. Monitoring application..."
echo "Cutover completed successfully at: $(date -u +"%Y-%m-%d %H:%M:%S UTC")"
echo "Monitoring for 30 minutes before declaring success..."

# Monitor application metrics for 30 minutes
for i in {1..30}; do
  echo "Monitoring minute $i/30..."

  # Check error rates, response times, etc.
  ERROR_COUNT=$(kubectl logs deployment/myapp --since=1m | grep -c "ERROR" || echo "0")
  if [ "$ERROR_COUNT" -gt 10 ]; then
    echo "High error rate detected: $ERROR_COUNT errors in the last minute"
    echo "Consider investigating or rolling back"
  fi

  sleep 60
done

echo "Cutover monitoring completed successfully!"
echo "Migration from MySQL to PostgreSQL is complete."
```

```
# 10. Clean up old resources (optional, run later)
echo "Remember to clean up MySQL resources after confirming stability"
```

7.3 Rollback Script

```
bash
```

```bash
#!/bin/bash
# rollback_script.sh

set -e

echo "EMERGENCY ROLLBACK: Switching back to MySQL"
echo "=========================================="

# 1. Enable maintenance mode
echo "1. Enabling maintenance mode..."
az network application-gateway http-settings update \
  --resource-group mysql-to-pg-migration \
  --gateway-name myapp-gateway \
  --name maintenance-settings \
  --path /maintenance.html

# 2. Scale down application
echo "2. Scaling down application..."
kubectl scale deployment myapp --replicas=0

# 3. Switch back to MySQL configuration
echo "3. Switching back to MySQL..."
kubectl create configmap myapp-config \
  --from-literal=DATABASE_URL="mysql://mysqladmin:MySQLPassword123!@myapp-mysql.mysql.database.azure.c
  --dry-run=client -o yaml | kubectl apply -f -

kubectl set env deployment/myapp PRIMARY_DB=mysql
kubectl set env deployment/myapp DUAL_WRITE_ENABLED=true

# 4. Scale up application
echo "4. Scaling up application..."
kubectl scale deployment myapp --replicas=3

# 5. Wait for readiness
echo "5. Waiting for application readiness..."
kubectl wait --for=condition=available --timeout=300s deployment/myapp

# 6. Test functionality
echo "6. Testing application..."
for i in {1..5}; do
  HTTP_STATUS=$(curl -s -o /dev/null -w "%{http_code}" https://myapp.example.com/health)
  if [ "$HTTP_STATUS" = "200" ]; then
    echo "Application health check passed"
    break
  fi
  sleep 10
```

```
done

# 7. Disable maintenance mode
echo "7. Disabling maintenance mode..."
az network application-gateway http-settings update \
  --resource-group mysql-to-pg-migration \
  --gateway-name myapp-gateway \
  --name default-settings \
  --path /

echo "Rollback completed successfully!"
echo "Application is now running on MySQL again."
```

Phase 8: Post-Migration Monitoring

8.1 Monitoring Setup

```
yaml
```

```yaml
# monitoring-stack.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
data:
  prometheus.yml: |
    global:
      scrape_interval: 15s
    scrape_configs:
      - job_name: 'postgresql-exporter'
        static_configs:
          - targets: ['postgres-exporter:9187']
      - job_name: 'application'
        static_configs:
          - targets: ['myapp:8080']

    rule_files:
      - "alert_rules.yml"

    alerting:
      alertmanagers:
        - static_configs:
            - targets: ['alertmanager:9093']

  alert_rules.yml: |
    groups:
    - name: database_alerts
      rules:
      - alert: PostgreSQLDown
        expr: up{job="postgresql-exporter"} == 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "PostgreSQL is down"
          description: "PostgreSQL database is not responding"

      - alert: HighErrorRate
        expr: rate(http_requests_total{status=~"5.."}[5m]) > 0.1
        for: 2m
        labels:
          severity: warning
        annotations:
          summary: "High error rate detected"
          description: "Error rate is {{ $value }} errors per second"
```

```
- alert: SlowQueries
  expr: postgresql_stat_activity_max_tx_duration{state="active"} > 300
  for: 5m
  labels:
    severity: warning
  annotations:
    summary: "Long running queries detected"
    description: "Query running for more than 5 minutes"
```

8.2 Performance Monitoring Script

python

```python
#!/usr/bin/env python3
# post_migration_monitor.py

import asyncio
import asyncpg
import time
import json
import logging
from datetime import datetime, timedelta
from typing import Dict, List, Any

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class PostgreSQLMonitor:
    def __init__(self, connection_config: Dict[str, str]):
        self.config = connection_config
        self.pool = None

    async def init_connection(self):
        """Initialize PostgreSQL connection pool"""
        self.pool = await asyncpg.create_pool(
            host=self.config['host'],
            port=self.config['port'],
            user=self.config['user'],
            password=self.config['password'],
            database=self.config['database'],
            min_size=2,
            max_size=5
        )

    async def get_database_metrics(self) -> Dict[str, Any]:
        """Get database performance metrics"""
        async with self.pool.acquire() as conn:
            metrics = {}

            # Connection count
            metrics['connections'] = await conn.fetchval("""
                SELECT count(*) FROM pg_stat_activity
                WHERE state = 'active' AND pid != pg_backend_pid()
            """)

            # Database size
            metrics['database_size_mb'] = await conn.fetchval("""
                SELECT pg_size_pretty(pg_database_size(current_database()))
            """)
            # Cache hit ratio
            metrics['cache_hit_ratio'] = await conn.fetchval("""
                SELECT round(
                    100 * sum(blks_hit) / (sum(blks_hit) + sum(blks_read)), 2
                ) as cache_hit_ratio
                FROM pg_stat_database
                WHERE datname = current_database()
            """)

            # Long running queries
            long_queries = await conn.fetch("""
                SELECT pid, now() - pg_stat_activity.query_start AS duration, query
                FROM pg_stat_activity
                WHERE (now() - pg_stat_activity.query_start) > interval '5 minutes'
                AND state = 'active'
                AND pid != pg_backend_pid()
            """)
            metrics['long_queries'] = len(long_queries)

            # Table statistics
            table_stats = await conn.fetch("""
                SELECT schemaname, tablename, n_tup_ins, n_tup_upd, n_tup_del,
                       n_tup_hot_upd, n_live_tup, n_dead_tup
                FROM pg_stat_user_tables
                ORDER BY n_live_tup DESC
                LIMIT 10
            """)
            metrics['top_tables'] = [dict(row) for row in table_stats]

            # Index usage
            index_usage = await conn.fetch("""
                SELECT schemaname, tablename, indexname, idx_tup_read, idx_tup_fetch
                FROM pg_stat_user_indexes
                WHERE idx_tup_read > 0
                ORDER BY idx_tup_read DESC
                LIMIT 10
            """)
            metrics['index_usage'] = [dict(row) for row in index_usage]

            return metrics

    async def get_slow_queries(self) -> List[Dict[str, Any]]:
        """Get slow query statistics"""
        async with self.pool.acquire() as conn:
            # Requires pg_stat_statements extension
            try:
                slow_queries = await conn.fetch("""
                    SELECT query, calls, total_time, mean_time,
                           stddev_time, rows, 100.0 * shared_blks_hit /
                           nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
                    FROM pg_stat_statements
                    WHERE mean_time > 100  -- queries taking more than 100ms on average
                    ORDER BY mean_time DESC
                    LIMIT 20
                """)
                return [dict(row) for row in slow_queries]
            except Exception as e:
                logger.warning(f"pg_stat_statements not available: {e}")
                return []

    async def check_replication_lag(self) -> Dict[str, Any]:
        """Check replication lag if applicable"""
        async with self.pool.acquire() as conn:
            try:
                # For primary server
                lag_info = await conn.fetchrow("""
                    SELECT
                        client_addr,
                        state,
                        pg_wal_lsn_diff(pg_current_wal_lsn(), flush_lsn) as lag_bytes
                    FROM pg_stat_replication
                """)

                if lag_info:
                    return dict(lag_info)
                else:
                    return {'status': 'no_replicas'}

            except Exception as e:
                logger.warning(f"Error checking replication: {e}")
                return {'error': str(e)}

    async def generate_report(self) -> Dict[str, Any]:
        """Generate comprehensive monitoring report"""
        timestamp = datetime.utcnow().isoformat()

        metrics = await self.get_database_metrics()
        slow_queries = await self.get_slow_queries()
        replication_info = await self.check_replication_lag()

        report = {
            'timestamp': timestamp,
            'database_metrics': metrics,
            'slow_queries': slow_queries,
            'replication_info': replication_info,
            'health_status': 'healthy'  # Default, can be updated based on thresholds
        }

        # Determine health status
        if metrics['connections'] > 80:
            report['health_status'] = 'warning'
        if metrics['cache_hit_ratio'] < 95:
            report['health_status'] = 'warning'
        if metrics['long_queries'] > 5:
            report['health_status'] = 'critical'

        return report

    async def close_connection(self):
        """Close database connection pool"""
        if self.pool:
            await self.pool.close()

async def main():
    """Main monitoring loop"""
    postgres_config = {
        'host': 'myapp-postgresql.postgres.database.azure.com',
        'port': 5432,
        'user': 'pgadmin',
        'password': 'SecurePassword123!',
        'database': 'myapp_database'
    }

    monitor = PostgreSQLMonitor(postgres_config)
    await monitor.init_connection()

    try:
        while True:
            report = await monitor.generate_report()

            # Log report
            logger.info(f"Database Health: {report['health_status']}")
            logger.info(f"Active Connections: {report['database_metrics']['connections']}")
            logger.info(f"Cache Hit Ratio: {report['database_metrics']['cache_hit_ratio']}%")
            logger.info(f"Long Running Queries: {report['database_metrics']['long_queries']}")

            # Save report to file or send to monitoring system
            with open(f"/tmp/db_report_{int(time.time())}.json", 'w') as f:
                json.dump(report, f, indent=2, default=str)
            # Send alerts if needed
            if report['health_status'] in ['warning', 'critical']:
                logger.warning(f"Database health issue detected: {report['health_status']}")
                # Here you would integrate with your alerting system
                # await send_alert(report)

            # Wait 5 minutes before next check
            await asyncio.sleep(300)

    except KeyboardInterrupt:
        logger.info("Monitoring stopped by user")
    finally:
        await monitor.close_connection()

if __name__ == "__main__":
    asyncio.run(main())
```

Cleanup and Optimization

9.1 Post-Migration Cleanup

```bash
```

```bash
#!/bin/bash
# cleanup_script.sh

echo "Post-Migration Cleanup and Optimization"
echo "========================================"

# 1. Optimize PostgreSQL configuration
echo "1. Optimizing PostgreSQL configuration..."

# Update PostgreSQL parameters for production workload
az postgres flexible-server parameter set \
  --resource-group mysql-to-pg-migration \
  --server-name myapp-postgresql \
  --name shared_preload_libraries \
  --value 'pg_stat_statements,auto_explain'

az postgres flexible-server parameter set \
  --resource-group mysql-to-pg-migration \
  --server-name myapp-postgresql \
  --name log_statement \
  --value 'mod'

az postgres flexible-server parameter set \
  --resource-group mysql-to-pg-migration \
  --server-name myapp-postgresql \
  --name log_min_duration_statement \
  --value '1000'

# 2. Create indexes for optimal performance
echo "2. Creating optimized indexes..."
PGPASSWORD='SecurePassword123!' psql \
  -h myapp-postgresql.postgres.database.azure.com \
  -U pgadmin \
  -d myapp_database \
  -c "
  -- Analyze all tables to update statistics
  ANALYZE;

  -- Create composite indexes based on query patterns
  CREATE INDEX CONCURRENTLY IF NOT EXISTS idx_users_email_active
  ON users(email) WHERE active = true;

  CREATE INDEX CONCURRENTLY IF NOT EXISTS idx_orders_user_date
  ON orders(user_id, created_at DESC);

  CREATE INDEX CONCURRENTLY IF NOT EXISTS idx_order_items_order_product
```

```
    ON order_items(order_id, product_id);

    -- Vacuum and analyze after index creation
    VACUUM ANALYZE;
    "

# 3. Setup automated maintenance
echo "3. Setting up automated maintenance..."
cat > /tmp/pg_maintenance.sql << EOF
-- Create maintenance procedures
CREATE OR REPLACE FUNCTION automated_maintenance()
RETURNS void AS \$\$
BEGIN
    -- Update table statistics
    ANALYZE;

    -- Vacuum tables with high update/delete activity
    VACUUM (ANALYZE, VERBOSE) users;
    VACUUM (ANALYZE, VERBOSE) orders;
    VACUUM (ANALYZE, VERBOSE) order_items;

    -- Reindex if needed
    REINDEX INDEX CONCURRENTLY idx_users_email_active;

    RAISE NOTICE 'Maintenance completed at %', now();
END;
\$\$ LANGUAGE plpgsql;

-- Schedule maintenance (requires pg_cron extension)
-- SELECT cron.schedule('pg-maintenance', '0 2 * * 0', 'SELECT automated_maintenance();');
EOF

PGPASSWORD='SecurePassword123!' psql \
    -h myapp-postgresql.postgres.database.azure.com \
    -U pgadmin \
    -d myapp_database \
    -f /tmp/pg_maintenance.sql

# 4. Clean up migration resources
echo "4. Cleaning up migration resources..."

# Stop AWS DMS task
aws dms stop-replication-task \
    --replication-task-arn arn:aws:dms:us-east-1:account:task:mysql-to-postgresql-continuous

# Delete AWS DMS resources (run after confirming stability)
read -p "Delete AWS DMS resources? (y/N): " confirm
```

```
if [[ $confirm =~ ^[Yy]$ ]]; then
  aws dms delete-replication-task \
    --replication-task-arn arn:aws:dms:us-east-1:account:task:mysql-to-postgresql-continuous

  aws dms delete-endpoint \
    --endpoint-arn arn:aws:dms:us-east-1:account:endpoint:mysql-source-azure

  aws dms delete-endpoint \
    --endpoint-arn arn:aws:dms:us-east-1:account:endpoint:postgresql-target-azure

  aws dms delete-replication-instance \
    --replication-instance-arn arn:aws:dms:us-east-1:account:rep:mysql-to-postgresql-migration
fi

# 5. Update backup strategy
echo "5. Updating backup strategy..."
az postgres flexible-server backup list \
  --resource-group mysql-to-pg-migration \
  --server-name myapp-postgresql

# Configure backup retention
az postgres flexible-server parameter set \
  --resource-group mysql-to-pg-migration \
  --server-name myapp-postgresql \
  --name backup_retention_days \
  --value 30

echo "Cleanup and optimization completed!"
```

Summary

This comprehensive migration strategy provides:

1. **Zero-downtime migration** using AWS DMS with continuous replication
2. **Fallback options** with Debezium/Kafka for real-time CDC
3. **Application-level dual-write** strategy for seamless cutover
4. **Comprehensive validation** at every step
5. **Automated rollback** procedures
6. **Production-ready monitoring** and alerting
7. **Performance optimization** post-migration

Key Success Factors:

- **AWS DMS** handles cross-engine conversion automatically

- **Dual-write pattern** ensures data consistency during transition
- **Comprehensive testing** validates functionality before cutover
- **Monitoring and alerting** catch issues early
- **Rollback procedures** provide safety net

The entire process typically takes 2-4 weeks including planning, testing, and monitoring phases, but the actual downtime is less than 5 minutes during cutover.