

Domain 2: Secure, optimize, and deploy database solutions

Implement data security and compliance with SQL

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/implement-data-security-compliance/1-introduction>

Introduction

Completed

Data security isn't just about preventing external attackers. You must protect sensitive information from unauthorized internal access, maintain compliance with regulations, and create accountability through detailed audit trails. Microsoft's SQL platforms provide built-in security features that address these requirements without requiring extensive application changes.

Modern development practices require you to build security into your database designs from the start. When you create tables that store personal information, you need to consider who should see that data and how to protect it. When you write stored procedures, you must think about what permissions they require and whether they could expose sensitive information. Security decisions made during development are far easier to implement than adding them later.

Modern database security requires a defense-in-depth approach. A single security measure isn't enough. You need encryption to protect data even if storage is compromised, masking to limit exposure of sensitive values, row-level filtering to enforce data boundaries, and auditing to track who accessed what and when. Each layer addresses different threat vectors and compliance requirements.

What you'll learn

In this module, you'll learn how to:

- Design and implement data encryption using Always Encrypted and column-level encryption
- Configure Dynamic Data Masking to protect sensitive data from unauthorized viewers
- Implement Row-Level Security to filter data access based on user context
- Design object-level permissions using roles and schemas
- Implement secure, passwordless database access using Microsoft Entra ID and Managed Identity

- Configure auditing to track database activity and maintain compliance
- Secure model endpoints for AI-enabled database solutions
- Protect GraphQL, REST, and MCP endpoints from unauthorized access

Prerequisites

- Experience writing T-SQL queries and developing databases in SQL Server, Azure SQL, or SQL databases in Microsoft Fabric
- Familiarity with database security concepts such as authentication and authorization
- Understanding of Microsoft Entra ID (formerly Azure Active Directory) basics
- Access to a SQL Server, Azure SQL Database, or SQL database in Microsoft Fabric for testing

2. Protect data with encryption

<https://learn.microsoft.com/en-us/training/modules/implement-data-security-compliance/2-design-implement-data-encryption>

Protect data with encryption

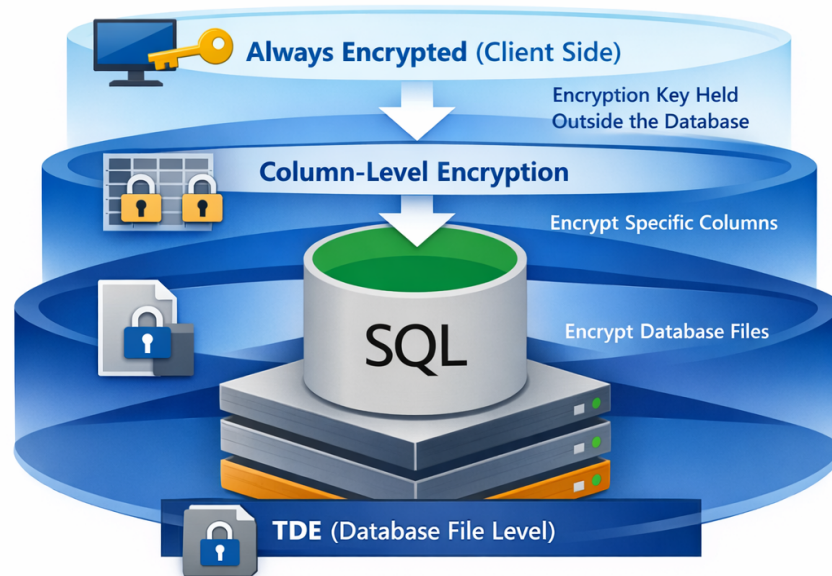
Completed

Data encryption forms the foundation of database security. Even if attackers gain access to your underlying storage, encryption keeps your sensitive information unreadable. Microsoft's SQL platforms offer multiple encryption options, from protecting data at rest to securing data while it's being processed.

Understanding when to use each method helps you balance protection with performance. Let's explore the encryption technologies available in SQL Server, Azure SQL, and SQL databases in Microsoft Fabric.

Understand encryption layers

Database encryption operates at different layers, each solving a specific problem. [Transparent Data Encryption \(TDE\)](#) encrypts data at rest—think of it as protecting your database files on disk. [Column-level encryption](#) targets specific sensitive columns, while [Always Encrypted](#) goes further by protecting data throughout its lifecycle, even during query processing.



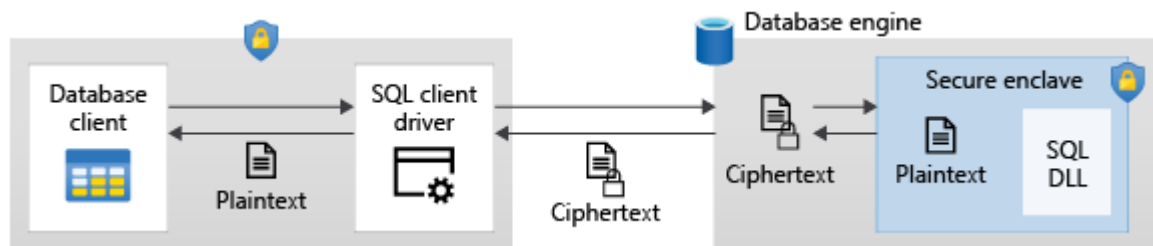
When you enable TDE, SQL Server automatically encrypts database files, transaction logs, and backups. Your applications don't need any code changes—encryption happens transparently behind the scenes. TDE uses a database encryption key protected by a certificate stored in the **master** database.

Column-level encryption works differently. You explicitly encrypt and decrypt data in your T-SQL code or application, giving you granular control over which columns contain sensitive data and who can decrypt them.

Always Encrypted takes yet another approach by keeping encryption keys outside the database engine entirely. The database never sees your plaintext data, which means even database administrators with high-level access can't view protected information.

Configure Always Encrypted

Always Encrypted ensures the database engine never processes plaintext values. Your client applications hold the encryption keys and handle all encryption and decryption. This separation means that even someone with administrative access to the database can't view the protected data.



To get started with Always Encrypted, you first create a column master key (CMK) that protects your column encryption keys. Store the CMK in a secure [key store](#) such as Azure Key Vault, Windows Certificate Store, or a hardware security module.

The following T-SQL statement creates a metadata entry pointing to your key in Azure Key Vault. The actual key material remains in the vault, never stored in the database.

```
CREATE COLUMN MASTER KEY MyCMK
WITH (
    KEY_STORE_PROVIDER_NAME = 'AZURE_KEY_VAULT',
    KEY_PATH = 'https://mykeyvault.vault.azure.net/keys/MyCMK/abc123'
);
```

Next, create a column encryption key (CEK) protected by the column master key:

```
CREATE COLUMN ENCRYPTION KEY MyCEK
WITH VALUES (
    COLUMN_MASTER_KEY = MyCMK,
    ALGORITHM = 'RSA_OAEP',
    ENCRYPTED_VALUE = 0x017000000016C006F00...
);
```

Notice that the CEK itself is stored in encrypted form. When your application needs to work with encrypted data, it retrieves this value and uses the CMK to decrypt it locally.

When creating or altering tables, you specify the encryption type for sensitive columns:

```
CREATE TABLE Employees (
    EmployeeID int PRIMARY KEY,
    SSN char(11) COLLATE Latin1_General_BIN2
        ENCRYPTED WITH (
            ENCRYPTION_TYPE = DETERMINISTIC,
            ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256',
            COLUMN_ENCRYPTION_KEY = MyCEK
        ),
    Salary money
        ENCRYPTED WITH (
            ENCRYPTION_TYPE = RANDOMIZED,
            ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256',
            COLUMN_ENCRYPTION_KEY = MyCEK
        )
);
```

You have two encryption types to choose from. Use **deterministic** when you need to perform equality comparisons, joins, or filter with `WHERE` clauses—the same plaintext always produces the same ciphertext. Use **randomized** for stronger security when you don't need those query operations.

Implement column-level encryption

Column-level encryption using T-SQL functions gives you an alternative when you need more control over the encryption process, or when Always Encrypted isn't the right fit. With this approach, you manage symmetric or asymmetric keys stored within the database.

Start by creating a database master key and certificate:

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD = 'StrongPassword123!';

CREATE CERTIFICATE SensitiveDataCert
WITH SUBJECT = 'Certificate for sensitive data encryption';
```

Create a symmetric key protected by the certificate:

```
CREATE SYMMETRIC KEY SensitiveDataKey
WITH ALGORITHM = AES_256
ENCRYPTION BY CERTIFICATE SensitiveDataCert;
```

To encrypt data, open the symmetric key and use the `ENCRYPTBYKEY` function:

```
OPEN SYMMETRIC KEY SensitiveDataKey
DECRYPTION BY CERTIFICATE SensitiveDataCert;

INSERT INTO CustomerData (CustomerID, CreditCardNumber)
VALUES (1, ENCRYPTBYKEY(KEY_GUID('SensitiveDataKey'), '4111-1111-1111-1111'));

CLOSE SYMMETRIC KEY SensitiveDataKey;
```

Decryption follows a similar pattern using `DECRYPTBYKEY`:

```
OPEN SYMMETRIC KEY SensitiveDataKey
DECRYPTION BY CERTIFICATE SensitiveDataCert;

SELECT CustomerID,
       CONVERT(varchar(20), DECRYPTBYKEY(CreditCardNumber)) AS CardNumber
FROM CustomerData;

CLOSE SYMMETRIC KEY SensitiveDataKey;
```

Yes, this approach requires more work—you're managing keys explicitly in your code. But that complexity comes with flexibility. You can grant or deny permissions to the symmetric key, giving you precise control over who can decrypt your data.

Choose the right encryption approach

Which encryption method should you use? It depends on your security requirements and application constraints.

TDE is your best choice when you need to protect data at rest without touching your application code. It's great for compliance requirements that mandate encryption of database files and backups. Keep in mind, though, that TDE doesn't protect data from users who can connect to the database with the right permissions.

Always Encrypted shines when you need to protect data from database administrators, or when sensitive data must stay encrypted even during query processing. The tradeoff? You need client driver support, and you're limited in what operations you can perform on encrypted columns.

Column-level encryption works well when you need granular control over encryption and decryption, or when you want to encrypt specific columns without the infrastructure overhead of Always Encrypted. It takes more development effort, but you get maximum flexibility in key management.

Tip

You can combine encryption methods. For example, enable TDE for baseline protection of data at rest, then add Always Encrypted for your most sensitive columns.

For SQL databases in Microsoft Fabric, TDE is enabled by default and managed automatically. Focus your encryption design decisions on column-level protection using Always Encrypted or symmetric key encryption based on your application requirements.

3. Configure dynamic data masking

<https://learn.microsoft.com/en-us/training/modules/implement-data-security-compliance/3-design-implement-dynamic-data-masking>

Configure dynamic data masking

Completed

Dynamic Data Masking provides a way to limit exposure of sensitive data without changing your application code or the underlying data. When users query masked columns, SQL Server returns obfuscated values based on masking rules you define. The actual data remains unchanged in the database, but unauthorized users see masked values in query results.

This approach works well when you need to protect sensitive information from certain users while allowing others to see the complete data. Database administrators, developers, and support staff can work with production data without exposing actual customer information, credit card numbers, or other sensitive values.

Understand masking functions

Dynamic Data Masking supports four masking functions, each designed for different data types and scenarios. Understanding these functions helps you choose the right masking approach for your sensitive columns.

Dynamic Data Masking Functions in SQL Server		
Function	Original Value	Masked Value
Default	John Smith	XXXX
Email	john.doe@example.com	jXXX@XXXX.com
Random	46295378	84930217
Partial	206-987-6543	206-XXX-XX89

The **default** function replaces the entire value with a fixed string. For strings, you see "XXXX" (or fewer X characters for shorter columns). Numbers display as zero, and dates show "01-01-1900". Use this when you want complete obfuscation.

The **email** function reveals just the first character of an email address, replaces the middle with "XXX", and preserves the domain suffix. So "*john.smith@contoso.com*" appears as "*jXXX@XXXX.com*". The data still looks like a valid email, but the actual address stays hidden.

The **random** function displays numeric values as a random number within a range you specify. Each query returns a different value. This works great for financial or statistical data where you need data that looks genuine.

The **partial** function gives you precise control. You specify how many characters to show at the start, what padding characters to use in the middle, and how many to show at the end. For example, a phone number might appear as "206-XXX-XX89".

Configure column masks

You apply masks when creating tables or by altering existing columns. The `MASKED WITH` clause specifies which function to use.

Here's a table with several masked columns:

```
CREATE TABLE Customers (  
    CustomerID int PRIMARY KEY,  
    FirstName varchar(50),  
    LastName varchar(50),  
    Email varchar(100) MASKED WITH (FUNCTION = 'email()'),  
    Phone varchar(20) MASKED WITH (FUNCTION = 'partial(3, "-XXX-XX", 2)'),  
    CreditCardNumber varchar(19) MASKED WITH (FUNCTION = 'partial(0, "XXXX-XXXX-XX", 2)'),  
    Income decimal(18,2) MASKED WITH (FUNCTION = 'random(10000, 100000)'),  
    SSN char(11) MASKED WITH (FUNCTION = 'default()')  
);
```

Notice how each column uses a different masking function based on what makes sense for that data:

- `Email` uses email masking to preserve the email format
- `Phone` shows the first three digits and last two digits
- `CreditCardNumber` reveals only the last four digits
- `Income` displays a random value between 10,000 and 100,000
- `SSN` uses default masking for complete obfuscation

To add masking to an existing column, use `ALTER COLUMN`:

```
ALTER TABLE Customers  
ALTER COLUMN DateOfBirth ADD MASKED WITH (FUNCTION = 'default()');
```

You can remove masking from a column when it's no longer needed:

```
ALTER TABLE Customers  
ALTER COLUMN DateOfBirth DROP MASKED;
```

Control mask visibility with permissions

By default, users see masked data unless they have elevated permissions. The `UNMASK` permission controls who sees the real values behind the masks.

To allow a user to see all unmasked data in the database:

```
GRANT UNMASK TO DataAnalyst;
```

This database-level permission reveals all masked columns to the specified user. However, you might want more granular control, allowing users to unmask only specific tables or columns.

Starting with SQL Server 2022, you can grant `UNMASK` at the schema, table, or even column level:

```
-- Grant unmask on a specific schema
GRANT UNMASK ON SCHEMA::Sales TO SalesManager;

-- Grant unmask on a specific table
GRANT UNMASK ON Customers TO CustomerService;

-- Grant unmask on a specific column
GRANT UNMASK ON Customers(Phone) TO TelemarketingTeam;
```

This granular approach helps you follow the principle of least privilege—users see only the sensitive data they actually need for their job.

Important

Users with `SELECT` permission on a table combined with `ALTER` permission can potentially bypass masking by modifying the column definition. Carefully manage permissions to prevent users from removing masks.

Implement masking strategies

When planning your masking strategy, think about how different user roles interact with your data. Which columns contain sensitive information? Who legitimately needs to see the actual values?

A common pattern is to create database roles for different access levels:

```
-- Role for users who see masked data
CREATE ROLE MaskedDataViewers;
GRANT SELECT ON Customers TO MaskedDataViewers;

-- Role for users who see unmasked data
CREATE ROLE UnmaskedDataViewers;
GRANT SELECT ON Customers TO UnmaskedDataViewers;
GRANT UNMASK ON Customers TO UnmaskedDataViewers;

-- Add users to appropriate roles
ALTER ROLE MaskedDataViewers ADD MEMBER SupportStaff;
ALTER ROLE UnmaskedDataViewers ADD MEMBER ComplianceOfficer;
```

One thing to keep in mind: Dynamic Data Masking works at the presentation layer, meaning the actual data can still be inferred through certain query patterns. For highly sensitive data requiring complete protection, combine masking with other security measures like encryption or Row-Level Security.

Consider these scenarios where masking is useful:

- Development and testing environments where teams need realistic data structures without actual customer information

- Customer service applications where support staff need to verify partial account details
- Reporting scenarios where aggregate data matters but individual records should remain private
- Audit compliance where specific users require full access while others see limited information

Note

Dynamic Data Masking in SQL databases in Microsoft Fabric works the same way as Azure SQL Database. You configure masks using T-SQL through the SQL analytics endpoint.

Masking adds an important layer of defense, but don't rely on it alone for your most sensitive data. Use it alongside encryption, Row-Level Security, and proper permission management for comprehensive protection.

4. Implement row-level security

<https://learn.microsoft.com/en-us/training/modules/implement-data-security-compliance/4-design-implement-row-level-security>

Implement row-level security

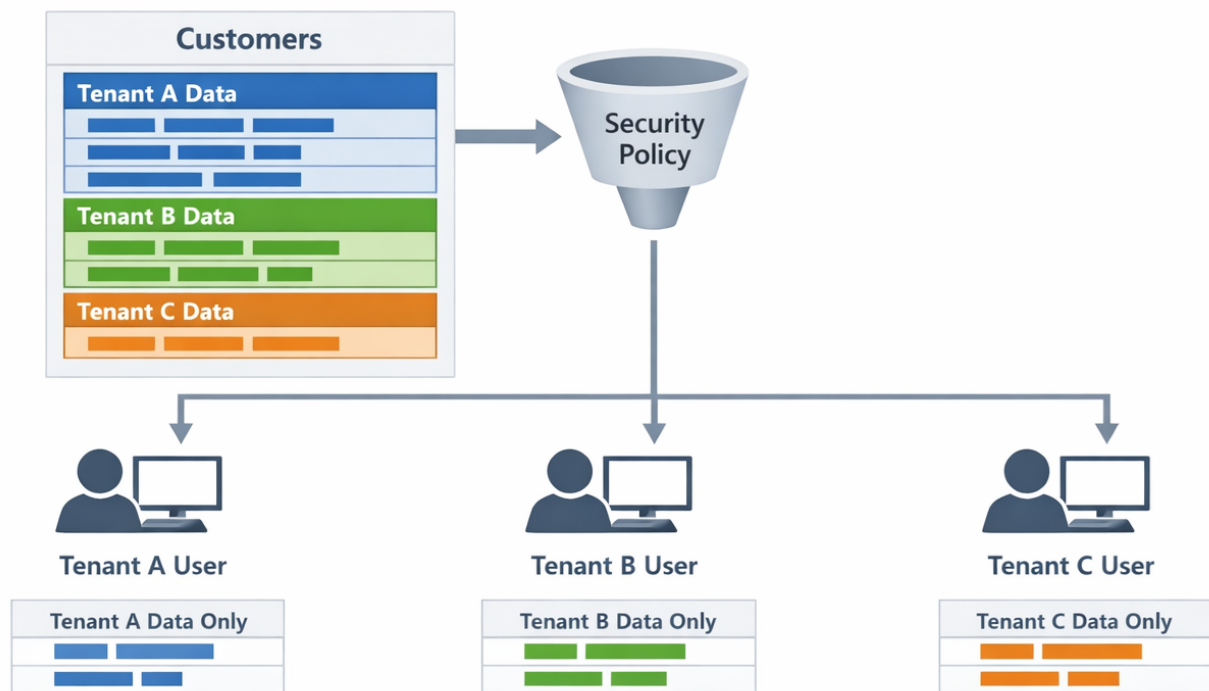
Completed

Row-Level Security (RLS) enables you to control access to rows in a database table based on the characteristics of the user executing a query. Unlike table-level permissions that grant or deny access to entire tables, RLS filters rows dynamically so users see only the data they're authorized to access.

This capability is useful when multiple users or tenants share the same tables but should only see their own data. Sales representatives see their own customer records, managers see their team's data, and regional directors see all data for their region. The filtering happens automatically without requiring application code changes.

Understand RLS components

Row-Level Security uses two components that work together: security predicates and security policies.



A **security predicate** is an inline table-valued function that returns 1 (true) or 0 (false) for each row. It receives the current user context and row values, then decides whether that user should see the row. Think of it as your business logic for data access, packaged as a function.

A **security policy** binds your predicate functions to tables and specifies the type of filtering. You can create **filter predicates** that silently exclude unauthorized rows from query results, or **block predicates** that prevent unauthorized insert, update, and delete operations.

Filter predicates affect `SELECT`, `UPDATE`, and `DELETE` statements by removing rows the user can't access. Users don't get errors—they just see a filtered result set. Block predicates work differently: they raise an error when users attempt unauthorized changes.

Create filter predicates

Let's start by creating a predicate function that evaluates row access. The function accepts parameters representing the column values to check and returns a table with a single row when access is allowed.

Here's a common scenario: a multitenant application where each row has a `TenantID` column:

```
CREATE SCHEMA Security;
GO

CREATE FUNCTION Security.fn_TenantAccessPredicate(@TenantID int)
RETURNS TABLE
WITH SCHEMABINDING
AS
```

```
RETURN SELECT 1 AS fn_TenantAccessPredicate_Result
WHERE @TenantID = CAST(SESSION_CONTEXT(N'TenantID') AS int);
```

This function checks whether the row's `TenantID` matches the value stored in session context. Your application sets this context after the user authenticates:

```
EXEC sp_set_session_context @key = N'TenantID', @value = 42;
```

For scenarios based on database users rather than session context, the predicate can reference the current user:

```
CREATE FUNCTION Security.fn_SalesRepPredicate(@SalesRepID int)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN SELECT 1 AS fn_SalesRepPredicate_Result
WHERE @SalesRepID = DATABASE_PRINCIPAL_ID()
OR IS_MEMBER('SalesManagers') = 1;
```

This predicate allows sales representatives to see their own records while managers in the `SalesManagers` role can see all records.

Create security policies

Once you've defined your predicate functions, create a security policy that applies them to tables:

```
CREATE SECURITY POLICY TenantSecurityPolicy
ADD FILTER PREDICATE Security.fn_TenantAccessPredicate(TenantID)
ON dbo.Orders,
ADD FILTER PREDICATE Security.fn_TenantAccessPredicate(TenantID)
ON dbo.OrderDetails
WITH (STATE = ON);
```

This policy filters both the `Orders` and `OrderDetails` tables using the same predicate function. When users query these tables, they only see rows matching their tenant context.

You can combine filter and block predicates in a single policy:

```
CREATE SECURITY POLICY SalesSecurityPolicy
ADD FILTER PREDICATE Security.fn_SalesRepPredicate(SalesRepID)
ON dbo.CustomerAccounts,
ADD BLOCK PREDICATE Security.fn_SalesRepPredicate(SalesRepID)
ON dbo.CustomerAccounts AFTER INSERT,
ADD BLOCK PREDICATE Security.fn_SalesRepPredicate(SalesRepID)
ON dbo.CustomerAccounts AFTER UPDATE
WITH (STATE = ON);
```

Why add block predicates? Without them, a user could insert a row with a different `SalesRepID` and then lose access to data they just created. The block predicates ensure users can only insert or update rows they'd be able to see.

Implement hierarchical access patterns

Many organizations need hierarchical data access—managers should see their subordinates' data. You can implement this by combining RLS with a management hierarchy table.

Here's a function that traverses the hierarchy:

```
CREATE FUNCTION Security.fn_HierarchyPredicate(@OwnerID int)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN
    WITH EmployeeHierarchy AS (
        SELECT EmployeeID, ManagerID
        FROM dbo.Employees
        WHERE EmployeeID = DATABASE_PRINCIPAL_ID()

        UNION ALL

        SELECT e.EmployeeID, e.ManagerID
        FROM dbo.Employees e
        INNER JOIN EmployeeHierarchy h ON e.ManagerID = h.EmployeeID
    )
    SELECT 1 AS fn_HierarchyPredicate_Result
    WHERE @OwnerID IN (SELECT EmployeeID FROM EmployeeHierarchy);
```

This recursive common table expression (CTE) builds the complete chain of employees reporting to the current user. The predicate allows access to any row owned by someone in that hierarchy.

Note

Recursive predicates can affect query performance on large datasets. Consider caching hierarchy relationships or limiting recursion depth for better performance.

Manage security policies

Security policies support several management operations for ongoing maintenance. You can disable a policy temporarily without removing it:

```
ALTER SECURITY POLICY TenantSecurityPolicy
WITH (STATE = OFF);
```

You can add new predicates to existing policies:

```
ALTER SECURITY POLICY TenantSecurityPolicy
ADD FILTER PREDICATE Security.fn_TenantAccessPredicate(TenantID)
ON dbo.Shipments;
```

Or remove predicates when tables no longer need filtering:

```
ALTER SECURITY POLICY TenantSecurityPolicy
DROP FILTER PREDICATE ON dbo.Orders;
```

To view existing security policies and their predicates:

```
SELECT p.name AS PolicyName,
       p.is_enabled,
       o.name AS TableName,
       pred.predicate_definition
FROM sys.security_policies p
INNER JOIN sys.security_predicates pred ON p.object_id = pred.object_id
INNER JOIN sys.objects o ON pred.target_object_id = o.object_id;
```

Tip

When troubleshooting RLS issues, temporarily disable the security policy and compare results. This helps determine whether unexpected results come from the RLS predicates or other query logic.

Row-Level Security works transparently with views, stored procedures, and other database objects that query the protected tables. Users accessing data through any path receive consistently filtered results based on their security context.

5. Manage permissions and secure access

<https://learn.microsoft.com/en-us/training/modules/implement-data-security-compliance/5-design-implement-object-level-permissions>

Manage permissions and secure access

Completed

Object-level permissions control what actions users can perform on database objects like tables, views, stored procedures, and functions. While Row-Level Security filters data within objects, object-

level permissions determine whether a user can access an object at all and what operations they can perform.

A well-designed permission model goes hand-in-hand with secure authentication. This unit covers both: how to grant the right permissions and how to ensure users authenticate securely, preferably without passwords.

Understand the permission hierarchy

SQL Server uses a [hierarchical permission model](#) where permissions granted at higher levels flow down to lower levels. Think of it as a cascade: server → database → schema → individual objects.

At the server level, permissions control sign-in management and server configuration. Database-level permissions govern actions within a specific database. Schema-level permissions apply to all objects within a schema, while object-level permissions target specific tables, views, or procedures.

Here's a powerful pattern: when you grant `SELECT` permission on a schema, users can select from all tables and views in that schema—including objects created in the future:

```
-- Grant SELECT on all objects in the Sales schema
GRANT SELECT ON SCHEMA::Sales TO SalesAnalyst;

-- Grant EXECUTE on all procedures in the Reports schema
GRANT EXECUTE ON SCHEMA::Reports TO ReportingUsers;
```

Grant and revoke permissions

Three statements control permissions: `GRANT` gives users specific permissions, `REVOKE` removes previously granted permissions, and `DENY` explicitly blocks permissions—overriding any grants.

```
-- Allow reading data
GRANT SELECT ON dbo.Customers TO CustomerServiceRole;

-- Allow data modifications
GRANT INSERT, UPDATE ON dbo.Orders TO OrderProcessingRole;

-- Explicitly deny access to sensitive data (overrides any grants)
DENY SELECT ON dbo.EmployeeSalaries TO HRAssistants;

-- Remove a permission without explicitly denying it
REVOKE INSERT ON dbo.Customers FROM CustomerServiceRole;
```

What's the difference between `REVOKE` and `DENY`? It matters when users have multiple permission sources. Revoking removes one grant but doesn't prevent access through other grants. Denying blocks access no matter what other grants exist.

Implement role-based access control

Instead of granting permissions directly to users, create [database roles](#) that represent job functions. Assign permissions to roles, then add users to the appropriate roles:

```
-- Create roles for different job functions
CREATE ROLE DataReaders;
CREATE ROLE DataWriters;

-- Grant appropriate permissions to each role
GRANT SELECT ON SCHEMA::dbo TO DataReaders;
GRANT INSERT, UPDATE, DELETE ON SCHEMA::dbo TO DataWriters;

-- Add users to roles
ALTER ROLE DataReaders ADD MEMBER JohnSmith;
ALTER ROLE DataWriters ADD MEMBER JaneDoc;
```

You can also nest roles to create hierarchical permission structures—adding someone to a parent role automatically gives them all child role permissions.

Tip

Document your role hierarchy and use a naming convention that makes each role's purpose clear, such as prefixing with the department or function.

Apply schema separation for security

[Schemas](#) provide a natural way to organize objects and manage permissions. Group related objects together, then grant permissions at the schema level:

```
CREATE SCHEMA Sales AUTHORIZATION dbo;
CREATE SCHEMA HR AUTHORIZATION dbo;

-- Sales team can read and write sales data
GRANT SELECT, INSERT, UPDATE ON SCHEMA::Sales TO SalesTeam;

-- HR has full control of their schema
GRANT CONTROL ON SCHEMA::HR TO HRAdministrators;
```

Configure Microsoft Entra authentication

Modern applications should use passwordless authentication whenever possible. [Microsoft Entra authentication](#) eliminates the risks associated with password management, including credential theft and the operational burden of rotating secrets.

SQL databases support multiple authentication methods. SQL authentication uses a username and password stored in the database. It's straightforward, but risky if credentials get compromised. Microsoft Entra authentication extends identity integration to cloud scenarios, working with Azure SQL Database, Azure SQL Managed Instance, SQL Server 2022+, and SQL databases in Microsoft Fabric.

Creating database users from Microsoft Entra identities is simple:

```
-- Create users for Entra accounts, managed identities, or groups
CREATE USER [app-service-identity] FROM EXTERNAL PROVIDER;
CREATE USER [developer@contoso.com] FROM EXTERNAL PROVIDER;
CREATE USER [DataAnalystsGroup] FROM EXTERNAL PROVIDER;

-- Grant permissions just like SQL users
ALTER ROLE db_datareader ADD MEMBER [developer@contoso.com];
ALTER ROLE db_datawriter ADD MEMBER [app-service-identity];
```

Implement managed identity authentication

[Managed identities](#) are the most secure option for Azure-hosted applications. Azure handles the identity lifecycle automatically, and there are no credentials that could be leaked or stolen.

For an Azure App Service connecting to Azure SQL Database, enable the system-assigned managed identity, then create a database user:

```
CREATE USER [MyWebApp] FROM EXTERNAL PROVIDER;
ALTER ROLE db_datareader ADD MEMBER [MyWebApp];
ALTER ROLE db_datawriter ADD MEMBER [MyWebApp];
```

Update your application connection string to use managed identity authentication:

```
Server=tcp:myserver.database.windows.net,1433;Database=mydb;Authentication=Active I
```

Tip

Use user-assigned managed identities when multiple applications need the same database permissions. This reduces the number of database users to manage.

Secure connection strings

No matter which authentication method you use, protect your connection strings with these best practices:

- Store them in Azure Key Vault or environment variables, not in code
- Use managed identities to eliminate credentials from connection strings entirely

- Enable encrypted connections with `Encrypt=True;TrustServerCertificate=False`
- Limit network access using firewall rules and private endpoints

Important

When using client secrets for service principals, store them in Azure Key Vault and rotate them regularly. Certificate-based authentication provides stronger security than shared secrets.

View and audit permissions

You can query system views to understand current permission assignments:

```
-- View all permissions for a specific principal
SELECT
    prin.name AS PrincipalName,
    perm.permission_name,
    perm.state_desc AS PermissionState,
    obj.name AS ObjectName
FROM sys.database_permissions perm
INNER JOIN sys.database_principals prin ON perm.grantee_principal_id = prin.principal_id
LEFT JOIN sys.objects obj ON perm.major_id = obj.object_id
WHERE prin.name = 'SalesAnalyst';

-- Check effective permissions for the current user
SELECT * FROM fn_my_permissions('Sales.Orders', 'OBJECT');
```

Make it a habit to audit permission assignments regularly to ensure they still align with current job functions. And remember, object-level permissions work best when combined with other security features like Row-Level Security and Dynamic Data Masking for comprehensive protection.

6. Implement auditing

<https://learn.microsoft.com/en-us/training/modules/implement-data-security-compliance/6-implement-auditing>

Implement auditing

Completed

Auditing provides visibility into database activity, helping you detect security threats, track compliance, and investigate incidents. By recording who accessed what data and when, auditing creates an accountability trail that supports both security monitoring and regulatory requirements.

SQL Server, Azure SQL, and SQL databases in Microsoft Fabric offer built-in auditing capabilities that capture database events without requiring application changes. Understanding how to configure and manage auditing helps you maintain appropriate oversight of your database environments.

Understand auditing options

Auditing captures database events and writes them to an audit log. The events you audit, where you store the logs, and how you analyze them depend on your security requirements and infrastructure.



[SQL Server Audit](#) uses the Extended Events infrastructure to record activity. You can write audit records to files, the Windows Security log, or the Windows Application log. This flexibility lets you integrate with existing log management systems.

[Azure SQL Database auditing](#) writes to Azure Blob Storage, Log Analytics, or Event Hubs. The managed service handles the infrastructure, so you can focus on deciding what to audit rather than managing storage.

SQL databases in Microsoft Fabric use Fabric's activity logging and Microsoft Purview for audit data. This integration with Microsoft Purview gives you unified auditing across your entire data estate.

Configure SQL Server auditing

SQL Server Audit requires creating a server audit object that defines where to write audit records, then creating audit specifications that define what to capture.

Start by creating a server audit that writes to a file:

```
CREATE SERVER AUDIT SecurityAudit
TO FILE (FILEPATH = 'C:\AuditLogs\', MAXSIZE = 100 MB, MAX_ROLLOVER_FILES = 10)
WITH (QUEUE_DELAY = 1000, ON_FAILURE = CONTINUE);
```

The `QUEUE_DELAY` parameter specifies how many milliseconds to buffer events before writing. Lower values provide more real-time logging but can affect performance. The `ON_FAILURE` setting

determines behavior when audit writing fails. Use `SHUTDOWN` for critical compliance scenarios where missing audit records is unacceptable.

Now enable the audit:

```
ALTER SERVER AUDIT SecurityAudit WITH (STATE = ON);
```

Next, create a server audit specification to capture server-level events:

```
CREATE SERVER AUDIT SPECIFICATION ServerAuditSpec
FOR SERVER AUDIT SecurityAudit
ADD (FAILED_LOGIN_GROUP),
ADD (SUCCESSFUL_LOGIN_GROUP),
ADD (SERVER_PERMISSION_CHANGE_GROUP),
ADD (DATABASE_PERMISSION_CHANGE_GROUP)
WITH (STATE = ON);
```

For database-level events, create a database audit specification:

```
USE MyDatabase;
GO

CREATE DATABASE AUDIT SPECIFICATION DatabaseAuditSpec
FOR SERVER AUDIT SecurityAudit
ADD (SELECT, INSERT, UPDATE, DELETE ON dbo.SensitiveData BY public),
ADD (EXECUTE ON SCHEMA::dbo BY public),
ADD (DATABASE_ROLE_MEMBER_CHANGE_GROUP)
WITH (STATE = ON);
```

This specification audits all data access on the `SensitiveData` table, stored procedure executions, and role membership changes.

Configure Azure SQL auditing

Azure SQL Database auditing is configured at the server or database level. Server-level policies apply to all databases on the logical server.

You can enable auditing in the Azure portal or using T-SQL:

```
-- Enable auditing to blob storage (configured in Azure portal)
ALTER DATABASE AUDIT SPECIFICATION AzureAuditSpec
ADD (SELECT, INSERT, UPDATE, DELETE ON DATABASE::MyDatabase BY public)
WITH (STATE = ON);
```

For more granular control, configure auditing through Azure Policy or ARM templates. Here's an example specifying the storage account, retention period, and audit action groups:

```
{
  "properties": {
    "state": "Enabled",
    "storageEndpoint": "https://myauditlogs.blob.core.windows.net",
    "retentionDays": 90,
    "auditActionsAndGroups": [
      "SUCCESSFUL_DATABASE_AUTHENTICATION_GROUP",
      "FAILED_DATABASE_AUTHENTICATION_GROUP",
      "BATCH_COMPLETED_GROUP"
    ]
  }
}
```

Note

Azure SQL auditing to Log Analytics enables powerful query and alerting capabilities using Kusto Query Language (KQL). This integration simplifies security monitoring and compliance reporting.

Select audit actions

Choose audit actions based on your security and compliance requirements. Common action groups include:

Authentication events:

- `SUCCESSFUL_DATABASE_AUTHENTICATION_GROUP` - Successful logins
- `FAILED_DATABASE_AUTHENTICATION_GROUP` - Failed sign-in attempts

Permission changes:

- `DATABASE_PERMISSION_CHANGE_GROUP` - Grant, revoke, deny operations
- `DATABASE_ROLE_MEMBER_CHANGE_GROUP` - Role membership changes
- `DATABASE_PRINCIPAL_CHANGE_GROUP` - User creation and modification

Data access:

- `BATCH_COMPLETED_GROUP` - All completed batches (high volume)
- `SELECT` on specific objects - Targeted read access monitoring
- `INSERT` , `UPDATE` , `DELETE` on specific objects - Data modification tracking

Schema changes:

- `SCHEMA_OBJECT_CHANGE_GROUP` - Table and object modifications
- `DATABASE_OBJECT_CHANGE_GROUP` - DDL statements

Important

Start with a focused set of audit actions rather than capturing everything. High-volume auditing impacts performance and generates logs that are difficult to analyze.

Query and analyze audit logs

For SQL Server file-based audits, use the `fn_get_audit_file` function to query your logs:

```
SELECT
    event_time,
    action_id,
    succeeded,
    session_server_principal_name AS UserName,
    database_name,
    object_name,
    statement
FROM fn_get_audit_file('C:\AuditLogs\*.sqlaudit', DEFAULT, DEFAULT)
WHERE event_time > DATEADD(day, -7, GETUTCDATE())
ORDER BY event_time DESC;
```

If you're using Azure SQL auditing with Log Analytics, you can query using KQL:

```
AzureDiagnostics
| where Category == "SQLSecurityAuditEvents"
| where TimeGenerated > ago(7d)
| where action_name_s == "SELECT"
| summarize count() by client_ip_s, server_principal_name_s
| order by count_ desc
```

This query identifies which users and IP addresses generated the most SELECT queries over the past week, helping identify unusual access patterns.

Implement audit retention and protection

Your audit logs need protection from tampering, and you need to retain them according to compliance requirements.

For SQL Server, configure immutable storage for audit files and use Windows file system permissions to prevent deletion. For Azure SQL, configure storage with immutable blob storage and set retention policies in Azure Storage lifecycle management.

Important

Store audit logs separately from the databases they monitor. This ensures that even if attackers compromise a database, they can't tamper with the audit trail.

For compliance scenarios, consider these retention practices:

- Define retention periods based on regulatory requirements (often seven years for financial data)
- Use immutable storage to prevent log deletion
- Implement log archival to cold storage for cost management
- Establish processes for audit log review and alerting

Regular review of audit data helps you identify security issues before they become incidents. Create alerts for failed sign-in attempts, permission changes outside maintenance windows, and unusual data access patterns.

7. Configure secure access to AI services

<https://learn.microsoft.com/en-us/training/modules/implement-data-security-compliance/7-secure-model-endpoints>

Configure secure access to AI services

Completed

AI-enabled database solutions often expose model endpoints that applications call to generate predictions, embeddings, or other AI-powered responses. Securing these endpoints protects both your AI models and the data they process. Managed Identity provides a secure, credential-free way to authenticate applications to model endpoints.

When applications call Azure OpenAI, Azure Machine Learning, or other AI services from within your database environment, you need to control who can invoke these calls and protect the communication channel. Let's look at strategies for securing model endpoints in SQL Server, Azure SQL, and SQL databases in Microsoft Fabric.

Understand model endpoint security concerns

Model endpoints present unique security challenges compared to traditional database operations. These endpoints often process sensitive data, can incur significant costs per call, and may expose intellectual property embedded in fine-tuned models.

What happens if someone gains unauthorized access? They could exfiltrate data through carefully crafted prompts, run up unexpected costs with excessive API calls, or probe your proprietary model behaviors. That's why protecting these endpoints requires authentication, authorization, and monitoring.

In Azure SQL and SQL databases in Microsoft Fabric, you can call external AI services using stored procedures or functions that make HTTP requests. These calls must authenticate to the AI service, typically using API keys or Managed Identity.

Configure Managed Identity for AI services

Managed Identity eliminates the need to store API keys in your database or application code. Instead, Azure manages the identity credentials automatically, and you grant that identity access to AI services.

For an Azure SQL Database that needs to call Azure OpenAI, start by enabling system-assigned managed identity:

```
-- The managed identity is enabled in Azure portal or via Azure CLI
-- Verify the identity exists
SELECT * FROM sys.dm_external_provider_certificate_store;
```

Next, grant the managed identity access to Azure OpenAI using Azure role-based access control:

```
# Get the managed identity principal ID from Azure SQL
$sqlIdentity = (Get-AzSqlServer -ServerName myserver -ResourceGroupName myrg).Identity

# Assign Cognitive Services User role on the Azure OpenAI resource
New-AzRoleAssignment -ObjectId $sqlIdentity `
  -RoleDefinitionName "Cognitive Services User" `
  -Scope "/subscriptions/{sub-id}/resourceGroups/{rg}/providers/Microsoft.CognitiveServices/openai/{openai-id}";
```

Create a [database-scoped credential](#) that uses Managed Identity:

```
CREATE DATABASE SCOPED CREDENTIAL AzureOpenAICredential
WITH IDENTITY = 'Managed Identity',
SECRET = '{"resourceId": "https://cognitiveservices.azure.com/"}';
```

This credential tells SQL to obtain a token for the Cognitive Services resource scope using its managed identity.

Implement secure endpoint calls

Now let's put it all together. Use [external REST endpoint invocation](#) to call AI services securely. First, create an external data source that references your AI endpoint:

```
CREATE EXTERNAL DATA SOURCE AzureOpenAI
WITH (
  TYPE = REST,
  LOCATION = 'https://myopenai.openai.azure.com',
  CREDENTIAL = AzureOpenAICredential
);
```

Then create a stored procedure that calls the AI endpoint:

```
CREATE PROCEDURE dbo.GetEmbedding
    @InputText nvarchar(max),
    @Embedding nvarchar(max) OUTPUT
AS
BEGIN
    DECLARE @payload nvarchar(max) = JSON_OBJECT('input': @InputText);
    DECLARE @response nvarchar(max);

    EXEC sp_invoke_external_rest_endpoint
        @url = 'https://myopenai.openai.azure.com/openai/deployments/text-embedding-ada-002/api-key?api-version=2023-05-01',
        @method = 'POST',
        @payload = @payload,
        @credential = AzureOpenAICredential,
        @response = @response OUTPUT;

    SET @Embedding = JSON_VALUE(@response, '$.data[0].embedding');
END;
```

Grant execute permission only to users who should access AI capabilities:

```
-- Create a role for AI feature access
CREATE ROLE AIFeatureUsers;
GRANT EXECUTE ON dbo.GetEmbedding TO AIFeatureUsers;

-- Add specific users to the role
ALTER ROLE AIFeatureUsers ADD MEMBER [app-service-identity];
```

Important

Limit which database users can invoke AI endpoints. AI calls often incur costs and can process sensitive data. Use role-based access to control which applications and users can use these features.

Secure Azure Machine Learning endpoints

Azure Machine Learning model endpoints follow similar patterns. Configure Managed Identity access and create credentials for authentication.

Here's how to set up access for real-time inference endpoints:

```
CREATE DATABASE SCOPED CREDENTIAL AMLCredential
WITH IDENTITY = 'Managed Identity',
SECRET = '{"resourceId": "https://ml.azure.com/"}';

CREATE EXTERNAL DATA SOURCE AMLEndpoint
WITH (
    TYPE = REST,
```

```
LOCATION = 'https://myworkspace.region.inference.ml.azure.com',
CREDENTIAL = AMLCredential
);
```

For batch inference endpoints, you typically submit data to storage and trigger a pipeline. Make sure to secure that storage access using Managed Identity.

Implement endpoint monitoring

You can monitor AI endpoint usage to detect anomalies and keep costs under control. Here's a simple logging approach:

```
-- Create a logging table for AI calls
CREATE TABLE dbo.AIEndpointLog (
    LogID int IDENTITY PRIMARY KEY,
    CallTimestamp datetime2 DEFAULT SYSUTCDATETIME(),
    CallerPrincipal nvarchar(128) DEFAULT ORIGINAL_LOGIN(),
    EndpointName nvarchar(256),
    InputLength int,
    ResponseStatus int,
    DurationMs int
);

-- Modify procedures to log calls
CREATE PROCEDURE dbo.GetEmbeddingWithLogging
    @InputText nvarchar(max),
    @Embedding nvarchar(max) OUTPUT
AS
BEGIN
    DECLARE @StartTime datetime2 = SYSUTCDATETIME();
    DECLARE @response nvarchar(max);
    DECLARE @status int;

    -- Make the AI call
    EXEC sp_invoke_external_rest_endpoint
        @url = 'https://myopenai.openai.azure.com/openai/deployments/text-embedding',
        @method = 'POST',
        @payload = JSON_OBJECT('input': @InputText),
        @credential = AzureOpenAICredential,
        @response = @response OUTPUT;

    -- Log the call
    INSERT INTO dbo.AIEndpointLog (EndpointName, InputLength, ResponseStatus, DurationMs)
    VALUES ('text-embedding', LEN(@InputText), 200, DATEDIFF(millisecond, @StartTime, SYSUTCDATETIME()));

    SET @Embedding = JSON_VALUE(@response, '$.data[0].embedding');
END;
```

With this logging in place, you can query the log to spot unusual patterns:

```
-- Find users making excessive AI calls
SELECT CallerPrincipal, COUNT(*) AS CallCount, AVG(DurationMs) AS AvgDuration
FROM dbo.AIEndpointLog
WHERE CallTimestamp > DATEADD(hour, -1, SYSUTCDATETIME())
GROUP BY CallerPrincipal
HAVING COUNT(*) > 100
ORDER BY CallCount DESC;
```

Tip

Set up alerts for unusual AI endpoint usage patterns. Sudden spikes in calls or calls from unexpected principals can indicate compromised credentials or application bugs.

Secure AI features in Microsoft Fabric

SQL databases in Microsoft Fabric handle AI endpoint access differently than Azure SQL. Instead of database-scoped credentials and Managed Identity, Fabric uses workspace roles and capacity-based access control.

In Fabric, AI capabilities are tied to the workspace. Users with **Contributor** or higher workspace roles can invoke AI features like AI skills and Copilot. Access control happens at the workspace level rather than through database permissions.

To secure AI features in Fabric:

- Assign workspace roles carefully, since they control AI access
- Monitor usage through Fabric's capacity metrics and activity logs
- Use Fabric's data protection features to control what data AI features can access

For detailed guidance on configuring AI services in Fabric, see the [Fabric AI documentation](#).

8. Secure data API endpoints

<https://learn.microsoft.com/en-us/training/modules/implement-data-security-compliance/8-secure-graphql-rest-mcp-endpoints>

Secure data API endpoints

Completed

Modern database solutions expose data through various API endpoints, enabling applications to access information without writing traditional SQL queries. GraphQL, REST, and Model Context Protocol (MCP) endpoints each present unique security considerations. Properly securing these endpoints protects your data while enabling the flexibility that modern applications require.

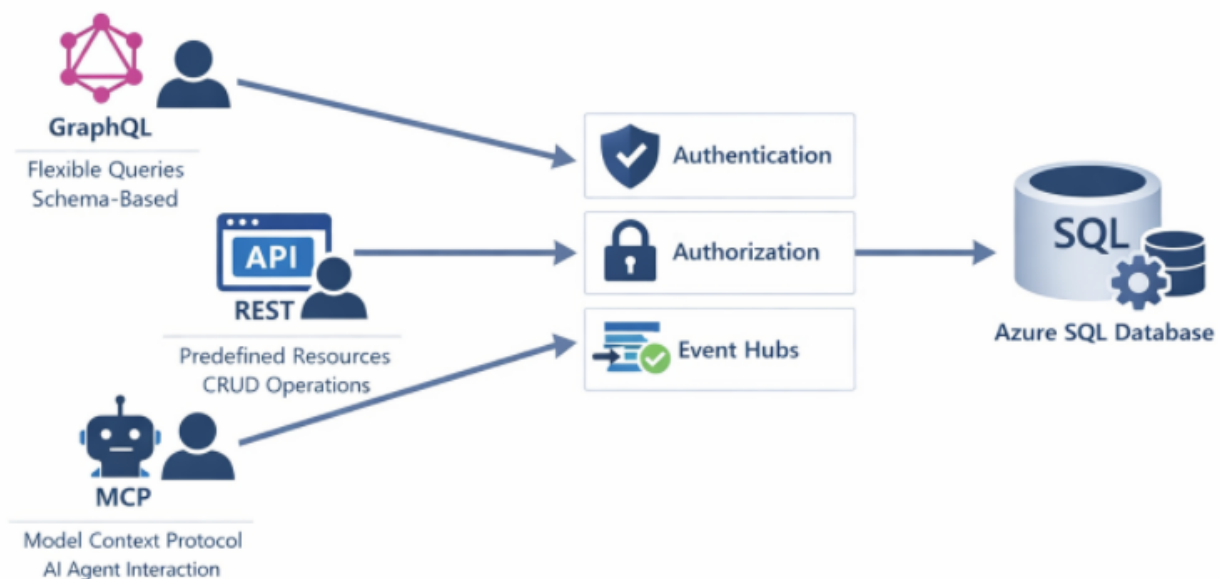
These endpoints often serve as the primary data access layer for web and mobile applications. Without appropriate security controls, they can expose sensitive data, allow unauthorized modifications, or become targets for denial-of-service attacks.

Understand endpoint types and risks

GraphQL endpoints provide flexible query capabilities, allowing clients to request exactly the data they need. This flexibility creates security challenges because clients can construct complex queries that might expose unintended data or consume excessive resources.

REST endpoints follow a more structured pattern with predefined resources and operations. Security focuses on authentication, authorization for specific endpoints, and input validation. REST APIs have been around long enough that security patterns are well established.

MCP endpoints enable AI models and agents to interact with databases, providing context and executing actions. These endpoints require careful security consideration because AI systems may attempt operations that human users wouldn't, and prompt injection attacks can manipulate AI behavior.



Secure GraphQL endpoints

Azure SQL Database and SQL databases in Microsoft Fabric support GraphQL through [Data API builder](#) and Microsoft Fabric's GraphQL API. Securing these endpoints involves authentication, authorization, and query controls.

Here's how to configure authentication requirements:

```
{
  "runtime": {
    "rest": { "enabled": false },
    "graphql": {
      "enabled": true,
      "path": "/graphql",
      "allow-introspection": false
    }
  },
  "authentication": {
    "provider": "AzureAD",
    "jwt": {
      "audience": "api://my-graphql-api",
      "issuer": "https://login.microsoftonline.com/{tenant-id}/v2.0"
    }
  }
}
```

Notice the `allow-introspection: false` setting. Disable introspection in production to prevent attackers from discovering your schema structure. Introspection queries reveal all available types, fields, and relationships.

Next, implement entity-level permissions to control which roles can access which data:

```
{
  "entities": {
    "Customer": {
      "source": "dbo.Customers",
      "permissions": [
        {
          "role": "authenticated",
          "actions": ["read"]
        },
        {
          "role": "admin",
          "actions": ["*"]
        }
      ]
    },
    "Order": {
      "source": "dbo.Orders",
      "permissions": [
        {
          "role": "authenticated",
          "actions": ["read"],
          "fields": {
            "include": ["OrderID", "OrderDate", "Status"],
            "exclude": ["InternalNotes", "CostPrice"]
          }
        }
      ]
    }
  }
}
```

```
}  
}  
}
```

See how the `Order` entity excludes sensitive columns like `InternalNotes` and `CostPrice`? Even authenticated users can't access those fields.

Tip

Implement query depth and complexity limits to prevent denial-of-service attacks through deeply nested queries. Data API builder includes built-in protections against excessive query complexity.

Secure REST endpoints

REST endpoints in Data API builder or custom implementations need authentication and endpoint-specific authorization. Here's an example:

```
{  
  "entities": {  
    "Product": {  
      "source": "dbo.Products",  
      "rest": {  
        "enabled": true,  
        "path": "/products"  
      },  
      "permissions": [  
        {  
          "role": "anonymous",  
          "actions": [  
            { "action": "read", "policy": { "database": "@item.IsPublic eq true" }  
          ]  
        },  
        {  
          "role": "inventory-manager",  
          "actions": ["create", "read", "update"]  
        },  
        {  
          "role": "admin",  
          "actions": ["*"]  
        }  
      ]  
    }  
  }  
}
```

Notice the database policy on anonymous access? It filters results to only public products. Role-based permissions then control what actions each role can perform.

For custom REST endpoints built with stored procedures, implement security at the database level:

```
CREATE PROCEDURE api.GetCustomerOrders
    @CustomerID int
AS
BEGIN
    -- Verify the caller has access to this customer
    IF NOT EXISTS (
        SELECT 1 FROM dbo.CustomerAccess
        WHERE CustomerID = @CustomerID
        AND UserPrincipal = ORIGINAL_LOGIN()
    )
    BEGIN
        THROW 50401, 'Unauthorized access to customer data', 1;
        RETURN;
    END

    SELECT OrderID, OrderDate, TotalAmount
    FROM dbo.Orders
    WHERE CustomerID = @CustomerID;
END;
```

Secure MCP endpoints

Model Context Protocol (MCP) endpoints let AI assistants and agents interact with your database. MCP requires extra security attention because AI systems might process untrusted user input.

Here's how to configure MCP server authentication:

```
{
  "mcpServers": {
    "sqlDatabase": {
      "transport": "stdio",
      "authentication": {
        "type": "azure-identity",
        "scope": "https://database.windows.net/.default"
      },
      "security": {
        "allowedOperations": ["read"],
        "deniedTables": ["dbo.Passwords", "dbo.APIKeys"],
        "maxRowsReturned": 1000
      }
    }
  }
}
```

The key is limiting what MCP endpoints can do:

- Restrict operations to read-only when write access isn't needed
- Explicitly deny access to tables containing credentials or sensitive configuration
- Limit result set sizes to prevent data exfiltration through large queries

- Log all MCP operations for security monitoring

You'll also want to implement prompt injection defenses by validating AI-generated queries:

```
CREATE PROCEDURE mcp.ExecuteQuery
    @QueryDescription nvarchar(max),
    @GeneratedQuery nvarchar(max) OUTPUT
AS
BEGIN
    -- Validate the generated query doesn't access restricted objects
    IF @GeneratedQuery LIKE '%sys.%' OR @GeneratedQuery LIKE '%INFORMATION_SCHEMA%'
    BEGIN
        THROW 50403, 'Access to system objects not permitted', 1;
        RETURN;
    END

    -- Ensure query is read-only
    IF @GeneratedQuery LIKE '%INSERT%' OR @GeneratedQuery LIKE '%UPDATE%'
    OR @GeneratedQuery LIKE '%DELETE%' OR @GeneratedQuery LIKE '%DROP%'
    BEGIN
        THROW 50403, 'Write operations not permitted', 1;
        RETURN;
    END

    -- Execute the validated query
    EXEC sp_executesql @GeneratedQuery;
END;
```

Important

Never trust AI-generated queries without validation. Implement allow lists for permitted tables and operations rather than trying to block malicious patterns.

Implement network security

Regardless of endpoint type, protect the network layer:

```
-- Azure SQL: Configure firewall rules
-- Deny all public access, allow only specific IPs or virtual networks
EXECUTE sp_set_firewall_rule
    @name = 'AllowAppService',
    @start_ip_address = '10.0.0.1',
    @end_ip_address = '10.0.0.255';
```

Use [Private Link](#) or service endpoints to keep traffic on the Microsoft network:

- Configure virtual network integration for App Services hosting API endpoints
- Use Private Endpoints for Azure SQL Database
- Enable managed private endpoints in Microsoft Fabric

These network controls add defense in depth, ensuring that even if application-level security is bypassed, attackers can't reach your database from unauthorized networks.

9. Exercise: Implement security features

<https://learn.microsoft.com/en-us/training/modules/implement-data-security-compliance/9-exercise-implement-security-features>

Exercise: Implement security features

Completed

Now it's your chance to implement data security and compliance features in Azure SQL Database.

In this exercise, you'll configure Dynamic Data Masking to protect sensitive data, implement Row-Level Security for multitenant data isolation, and set up auditing to track database activity for compliance purposes.

Note

To complete this exercise, you'll need an [Azure subscription](#).

Launch the exercise and follow the instructions.

[Launch Exercise](#)

10. Module assessment

<https://learn.microsoft.com/en-us/training/modules/implement-data-security-compliance/10-knowledge-check>

Module assessment

Completed

11. Summary

<https://learn.microsoft.com/en-us/training/modules/implement-data-security-compliance/11-summary>

Summary

Completed

In this module, you learned how to implement comprehensive data security and compliance features for SQL Server, Azure SQL Database, and SQL databases in Microsoft Fabric.

You explored how to:

- Implement data encryption using Always Encrypted for client-side encryption and column-level encryption for granular protection
- Configure Dynamic Data Masking to protect sensitive columns while maintaining data usability for authorized users
- Design Row-Level Security policies to filter data access based on user context and business rules
- Apply object-level permissions using roles and schemas to implement the principle of least privilege
- Enable passwordless authentication using Microsoft Entra ID and Managed Identity
- Set up auditing to track database activity and maintain compliance records
- Secure AI model endpoints using Managed Identity authentication
- Protect GraphQL, REST, and MCP endpoints from unauthorized access

Key takeaways

- Defense in depth requires combining multiple security features. Use encryption for data protection, masking for presentation-layer security, and Row-Level Security for row-level access control.
- Managed Identity eliminates credential management risks by letting Azure handle authentication automatically.
- Auditing provides accountability, but plan your retention strategy and storage location to meet compliance requirements while managing costs.

Learn more

- [Always Encrypted documentation](#)
- [Dynamic Data Masking in Azure SQL Database](#)
- [Row-Level Security](#)

- [Microsoft Entra authentication with Azure SQL](#)
- [Auditing for Azure SQL Database](#)

Optimize database performance

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/optimize-database-performance/01-introduction>

Introduction

Completed

A slow query during peak hours can cascade into a service outage. A poorly chosen isolation level can cause users to see stale data or block each other for seconds at a time. Performance problems in Azure SQL Database rarely have a single cause. They emerge from the interaction between hardware configuration, concurrency behavior, query execution, and workload patterns. Diagnosing these problems requires understanding each layer and knowing which tools to use at each level.

Imagine a team managing an e-commerce application on Azure SQL Database. During a holiday sale, transaction throughput drops and customers report timeouts. Is the database under-provisioned? Are queries scanning entire tables instead of seeking through indexes? Are blocking chains forming between concurrent transactions? Is the query optimizer choosing a bad plan? Each possibility requires a different investigation tool and a different resolution strategy. This module gives you the skills to work through each of those questions systematically.

After completing this module, you're able to:

- Evaluate and recommend database configurations including service tiers, compute tiers, and resource limits.
- Choose transaction isolation levels and concurrency controls that balance consistency with throughput.
- Analyze query performance using execution plans and dynamic management views.
- Monitor and tune queries with Query Store and Query Performance Insight.
- Identify and resolve blocking and deadlocks using DMVs and Extended Events.

2. Recommend database configurations

Recommend database configurations

Completed

Before you can tune queries or troubleshoot concurrency, you need the right infrastructure underneath. Azure SQL Database gives you two resource models and multiple service tiers. The combination you choose sets the ceiling on compute, memory, I/O, and storage for your workload. Choose too little and performance suffers. Choose too much and you waste budget.

Compare resource models

Azure SQL Database supports two resource models: **vCore** and **DTU**. They measure and bill resources differently, so understanding the distinction helps you make the right choice from the start.

The **vCore model** gives you direct control over virtual cores, memory, and storage. You pick the hardware generation, service tier, and compute tier independently. If you're migrating from on-premises SQL Server, this model maps cleanly to physical CPU and memory, which makes capacity planning more straightforward. It also supports reserved instance pricing and the Azure Hybrid Benefit for cost savings.

The **DTU model** bundles CPU, memory, and I/O into a single unit called a **Database Transaction Unit** (DTU). DTU-based tiers (Basic, Standard, and Premium) offer preconfigured resource packages. This model works when you don't need fine-grained control over individual resource dimensions.

For most new deployments, Microsoft recommends the vCore model. It provides higher resource limits, greater scaling granularity, and more pricing flexibility.

Understand service tiers in the vCore model

The vCore model has three service tiers: General Purpose, Business Critical, and Hyperscale. Each tier uses a different architecture, which affects storage type, I/O performance, and availability.

General Purpose separates compute and storage. The database engine runs on a compute node while data files reside on Azure Blob Storage. Storage latency is typically between 5 milliseconds and 10 milliseconds. This architecture provides budget-friendly pricing and works well for most business workloads. If the compute node fails, Azure Service Fabric moves the process to a spare node and reattaches the remote storage files.

Consider the e-commerce application from the introduction. During normal business hours, General Purpose handles the order volume without issue. But during a holiday flash sale, I/O latency of 5

milliseconds to 10 milliseconds might not be fast enough for the checkout flow.

Business Critical integrates compute and storage on each node. The database engine and data files both use locally attached SSDs in an Always On availability group with three secondary replicas. This design gives you the lowest I/O latency (1 millisecond to 2 milliseconds on average), the highest IOPS (input/output operations per second), and a free read-only replica you can use for reporting queries. The trade-off is cost, roughly 2.7 times more than General Purpose for the same vCore count. For the e-commerce team, Business Critical makes sense if their checkout transactions need consistent sub-2-millisecond latency.

Hyperscale uses a decoupled storage architecture with independent page servers and a multi-tiered cache. It supports databases up to 128 TB, allows zero to four high-availability replicas, and scales compute up or down without copying data. You're billed only for allocated storage, not maximum storage. Hyperscale removes the practical storage and scaling limits of the other tiers and is suitable for the widest variety of workloads.

The following table summarizes the key differences:

Feature	General Purpose	Business Critical	Hyperscale
Storage type	Remote (Azure Blob Storage)	Local SSD	Decoupled with local SSD cache
Max storage	4 TB	4 TB	128 TB
Max IOPS per vCore	320	4,000	5,500 (local SSD)
Availability replicas	1 (no read replicas)	3 + 1 read replica	0 to 4 (configurable)
Best for	Budget-oriented workloads	Low latency, high I/O	Large databases, flexible scaling

Choose a compute tier

Within the vCore model, you also choose between two compute tiers: **provisioned** and **serverless**.

Provisioned compute allocates a fixed number of vCores that stay available regardless of workload activity. You pay a fixed hourly rate. This tier fits workloads with consistent or predictable resource consumption, like the e-commerce application that processes orders throughout the day.

Serverless compute automatically scales vCores based on demand and bills per second for the compute used. When the database is idle, it can autopause and eliminate compute costs entirely, though autopause is currently supported only in General Purpose. Serverless compute itself is available for both General Purpose and Hyperscale tiers. It works well for development environments, internal tools, or applications with intermittent traffic.

Match the configuration to your workload

Now that you understand the options, how do you decide? Evaluate your workload against these factors:

- **Latency requirements:** If your application needs I/O latency under 2 milliseconds consistently, choose Business Critical. For moderate latency tolerance, General Purpose is sufficient.
- **Storage size:** If your database exceeds 4 TB or you expect rapid growth, Hyperscale is the only option that accommodates up to 128 TB.
- **Read-heavy workloads:** Business Critical includes a free read-only replica. Hyperscale supports named replicas for flexible read scale-out.
- **Cost sensitivity:** General Purpose with provisioned compute offers predictable pricing. Serverless compute in General Purpose or Hyperscale reduces costs for intermittent workloads.
- **Availability requirements:** Business Critical provides the highest resiliency with three synchronous replicas and the fastest failover. Hyperscale lets you configure the number of replicas to balance resiliency with cost.

Tip

When migrating from on-premises SQL Server, use the vCore model because it maps directly to physical CPU and memory. The DTU model doesn't expose individual resource dimensions, which makes capacity planning harder for migrations.

Configure database-level settings

You picked your tier and compute model. Now look inside the database itself. Several settings affect how Azure SQL Database handles parallelism, query optimization, and recovery. You adjust these settings without changing your service tier.

Control parallelism with MAXDOP

Max degree of parallelism (MAXDOP) controls how many processor threads the engine assigns to a single query. Azure SQL Database defaults to **8**, which works for the widest variety of workloads. Before September 2020, new databases defaulted to 0, unlimited parallelism, and that caused problems. A single analytical query could consume every available thread, starving the checkout flow of CPU.

You set MAXDOP at the database level with `ALTER DATABASE SCOPED CONFIGURATION SET MAXDOP` . You can also set a different value for secondary replicas when your read-write and read-only workloads have different concurrency needs. For a specific query, use the `OPTION (MAXDOP)` hint. The one rule: avoid MAXDOP 0 in production. Unlimited parallelism leads to resource exhaustion, query timeouts, and application outages.

Let automatic tuning catch regressions

The query optimizer doesn't always pick the best plan. Statistics go stale, data distributions shift, and a plan that was fast yesterday becomes slow today. **Automatic tuning** monitors query performance

and applies corrections without waiting for you to notice.

Azure SQL Database supports three options:

- **FORCE_LAST_GOOD_PLAN** detects plan regressions and forces the previous fast plan. Enabled by default.
- **CREATE_INDEX** identifies missing indexes, creates them, and verifies the improvement. Disabled by default.
- **DROP_INDEX** removes unused and duplicate indexes. Disabled by default. Unique indexes, including indexes supporting primary key and unique constraints, are never dropped.

Every change goes through a validation window, 30 minutes to 72 hours depending on query frequency. If performance gets worse, the change is automatically reverted.

```
ALTER DATABASE CURRENT
SET AUTOMATIC_TUNING (FORCE_LAST_GOOD_PLAN = ON,
                      CREATE_INDEX = ON,
                      DROP_INDEX = OFF);
```

Think about the e-commerce application during a holiday sale. Query patterns shift as different product pages spike in popularity. **FORCE_LAST_GOOD_PLAN** catches those regressions automatically, so a bad plan change at 2 AM doesn't slow down checkout until someone notices Monday morning. You probably want to leave **CREATE_INDEX** and **DROP_INDEX** off until what they propose is reviewed.

Unlock optimizer features with compatibility level

Every database has a **compatibility level** that determines which query optimizer behaviors are available. New databases in Azure SQL Database default to level **170**, or the highest available level. Each level unlocks a set of **intelligent query processing (IQP)** features:

- **Level 150**: batch mode on rowstore, table variable deferred compilation, scalar user-defined function (UDF) inlining.
- **Level 160**: parameter sensitive plan optimization (PSP), cardinality estimation feedback.
- **Level 170**: optional parameter plan optimization.

Existing databases can run at a lower compatibility level because Microsoft never automatically upgrades this setting. A database created when a lower default was in effect keeps its original level. For example, if you created an Azure SQL Database in 2024, the database is still at level 160 if the level isn't manually updated. Likewise, if you imported a database through a BACPAC file, the imported database's compatibility level is based on the source database's compatibility level. To move up:

```
ALTER DATABASE CURRENT SET COMPATIBILITY_LEVEL = 170;
```

Don't change this setting blindly in production. Use Query Store to capture a performance baseline at the current level, upgrade in a test environment, and compare. If a query regresses, you can force

the old plan while you investigate.

Reduce plan cache bloat with **OPTIMIZE_FOR_AD_HOC_WORKLOADS**

The e-commerce application generates product search queries with dozens of filter combinations. Each unique query text gets its own compiled plan in the cache, even if that query never runs again. Over time, the plan cache fills with thousands of single-use plans, pushing out the frequently executed plans that actually matter.

OPTIMIZE_FOR_AD_HOC_WORKLOADS solves this issue. When enabled, the engine stores a tiny **compiled plan stub** on first execution instead of the full plan. Only when the same query runs a second time does the engine compile and cache the full plan.

```
ALTER DATABASE SCOPED CONFIGURATION  
SET OPTIMIZE_FOR_AD_HOC_WORKLOADS = ON;
```

This setting keeps the cache lean and ensures that plans for your most important queries stay resident in memory.

Understand accelerated database recovery

Accelerated database recovery is always enabled in Azure SQL Database. You can't turn it off, and you don't need to. Accelerated database recovery (ADR) redesigns the recovery process so that recovery time stays constant regardless of how many active transactions were running when a failure occurred. It also provides instantaneous transaction rollback and aggressive log truncation.

ADR stores row versions in a **persistent version store** (PVS) inside the database rather than in `tempdb`. Depending on the size of the row being modified, versions are stored either in-row on data pages or off-row in a separate internal table. Write-intensive workloads can see increased page splits and higher log generation because every row version is logged. To minimize that overhead, keep transactions short and reduce unnecessary aborted transactions.

The PVS shares your database's allocated storage, so a growing PVS reduces the space available for your data. To monitor off-row PVS overhead, query

`sys.dm_tran_persistent_version_store_stats` and check the `persistent_version_store_size_kb` column, which reports the size of off-row versions only and doesn't include in-row versions stored on data pages. To establish a baseline during typical workloads, compare that value to your total database size. If PVS grows significantly beyond that baseline, look for long-running transactions or high abort rates that delay version cleanup.

Key takeaways

Azure SQL Database gives you two resource models, three service tiers, and two compute tiers. The vCore model with General Purpose covers most workloads. Business Critical adds sub-2-millisecond latency. Hyperscale removes storage and scaling limits. Inside the database, MAXDOP 8 is the safe default, automatic tuning catches plan regressions, and upgrading your compatibility level unlocks

the latest intelligent query processing (IQP) features. Enable `OPTIMIZE_FOR_AD_HOC_WORKLOADS` to keep the plan cache clean, and monitor PVS storage usage from ADR using `sys.dm_tran_persistent_version_store_stats`, especially in write-heavy scenarios. Next, you explore how isolation levels and concurrency control affect the queries running inside this infrastructure.

3. Preserve data integrity with transaction isolation levels and concurrency controls

<https://learn.microsoft.com/en-us/training/modules/optimize-database-performance/03-preserve-data-integrity-isolation-levels>

Preserve data integrity with transaction isolation levels and concurrency controls

Completed

Choosing the right hardware and configuration is only part of database performance. What happens when two customers try to buy the last item in stock at the exact same moment? Or when a report reads a price that another transaction is in the middle of changing? Transaction isolation levels control how the database handles these situations.

How isolation levels work

Isolation levels define how much one transaction can see of another transaction's uncommitted work. The SQL standard frames them around three **concurrency side effects**, each of which maps to a real problem in your applications. Let's use an e-commerce application to illustrate them:

- **Dirty reads:** A customer's order reads a discounted price that another transaction set but isn't committed yet. If the transaction that set the price rolls back, the order used a price that never existed. Dirty reads are defined as reading uncommitted data from another transaction, which can lead to incorrect behavior if that other transaction doesn't commit.
- **Nonrepeatable reads:** On the same transaction, your inventory service reads the stock count for an item. After the initial read, the transaction does some calculations, then reads the same row again. In between those steps, another transaction committed a change, so the count is now different. Nonrepeatable reads occur when a transaction reads the same row twice and gets different values each time.
- **Phantom reads:** A report queries all orders placed in the last hour, calculates a total, then runs the same query moments later within the same transaction. Another transaction inserted new

orders in between, so the totals don't match. Phantom reads happen when a transaction re-executes a query and sees new rows that weren't there before.

Consider our e-commerce application during a flash sale with hundreds of concurrent customers. All three side effects can show up at once. The isolation level you choose determines which ones your application tolerates and how much blocking it accepts in return.

Notice that each of these problems involves a transaction that performs multiple operations: read a row, do some work, then read again or make a decision. A single-statement transaction usually completes so quickly that these conflicts rarely surface. Isolation levels matter most when your transactions span multiple statements, which is common in business logic that reads data, applies rules, and then writes results.

Here's a typical pattern where isolation levels come into play. Imagine two customers both trying to buy the last unit of the same product:

```
-- Transaction A (Customer 1)
BEGIN TRANSACTION;

SELECT @Stock = StockCount
FROM Products
WHERE ProductID = 42;
-- @Stock = 1, proceed with order

UPDATE Products
SET StockCount = StockCount - 1
WHERE ProductID = 42;
INSERT INTO Orders (ProductID, Quantity)
VALUES (42, 1);
COMMIT TRANSACTION;

-- Transaction B (Customer 2)
BEGIN TRANSACTION;

SELECT @Stock = StockCount
FROM Products
WHERE ProductID = 42;
-- @Stock = 1, proceed with order

UPDATE Products
SET StockCount = StockCount - 1
WHERE ProductID = 42;
INSERT INTO Orders (ProductID, Quantity)
VALUES (42, 1);
COMMIT TRANSACTION;

-- StockCount is now -1. Two orders placed for one item.
```

Both transactions read a stock count of 1 and both decided to proceed. The isolation level determines whether Transaction B sees the original value or waits for Transaction A to finish first. If the isolation level allows dirty reads, Transaction B reads the uncommitted value of 1 and proceeds, which leads to overselling. If the isolation level is more restrictive, Transaction B waits until Transaction A commits. If Transaction A commits successfully, Transaction B sees the updated stock count of 0 and can prevent the second order from going through. If Transaction A rolls back for some reason, Transaction B sees the original value of 1 and can still proceed without ever reading uncommitted data.

SQL Server and Azure SQL Database support six isolation levels. The first four use **pessimistic concurrency** (locking). The last two use **optimistic concurrency** (row versioning).

Lock-based isolation levels

With lock-based isolation, the database engine uses shared and exclusive locks to keep transactions from interfering with each other. The more restrictive the level, the longer those locks are held. Fewer side effects get through, but more transactions end up waiting.

READ UNCOMMITTED is the fastest and the riskiest isolation level. It skips shared locks on reads entirely, so there's no read-write blocking. But that also means a transaction can read another transaction's uncommitted changes. If that other transaction rolls back, your application just acted on data that never existed. This level makes sense only when approximate results are acceptable, like a dashboard showing rough order counts.

READ COMMITTED is the default isolation level in SQL Server and strikes a basic balance. Each read acquires a shared lock, grabs the data, and releases the lock right away. Releasing the lock quickly is enough to prevent dirty reads. However, nothing stops another transaction from changing the row between your two reads, so nonrepeatable reads and phantom reads can still happen.

REPEATABLE READ goes a step further than READ COMMITTED. It holds shared locks on every row you read until your transaction completes. Now those rows can't change while the transaction is active, which prevents both dirty reads and nonrepeatable reads. But other transactions can still insert new rows that match your query filter, so phantom reads remain a possibility.

SERIALIZABLE takes care of everything previous levels do and more. It takes range locks that cover not just the rows you read but also the gaps between key values, blocking inserts into those ranges. Dirty reads, nonrepeatable reads, and phantom reads are all prevented. The cost is real, though: range locks cause significantly more blocking and increase the chance of deadlocks. Reserve this level for scenarios like financial reconciliation or inventory reservation where phantom reads would cause calculation errors.

Row-versioning isolation levels

What if readers didn't have to wait for writers at all? That question is what row-versioning isolation addresses. Instead of blocking behind locks, the database engine keeps previous versions of rows in a **version store**. When a transaction needs to read data that another transaction is modifying, it reads from the version store instead of waiting. In Azure SQL Database, accelerated database recovery (ADR) is always enabled, so the version store resides in the database itself using the persistent version store (PVS).

Read Committed Snapshot Isolation changes the behavior of READ COMMITTED at the database level. With Read Committed Snapshot Isolation (RCSI) enabled, each read operation sees a snapshot of the data as it existed at the start of that *statement*. Writers still take locks on the rows they modify, but readers never block behind them. The important detail: RCSI is enabled by default in Azure SQL Database, so you get this behavior out of the box without any code changes.

SNAPSHOT isolation takes this solution a step further. Instead of a per-statement snapshot, each read sees the data as it existed at the start of the entire *transaction*. If your application needs to run multiple queries within a single transaction and get consistent results across all of them, SNAPSHOT isolation provides that guarantee. To use it, you must enable `ALLOW_SNAPSHOT_ISOLATION` on the database and set the isolation level explicitly in the session. One thing to watch for: if two transactions try to modify the same row, SNAPSHOT isolation raises an update conflict rather than letting one silently overwrite the other.

The following table summarizes the behavior of each isolation level:

Isolation level	Dirty reads	Nonrepeatable reads	Phantom reads	Blocking between readers and writers
READ UNCOMMITTED	Yes	Yes	Yes	No
READ COMMITTED	No	Yes	Yes	Yes
REPEATABLE READ	No	No	Yes	Yes
SERIALIZABLE	No	No	No	Yes (range locks)
READ COMMITTED SNAPSHOT	No	Yes	Yes	No
SNAPSHOT	No	No	No	No (update conflicts possible)

Reduce blocking with optimized locking

RCSI eliminates read-write blocking. But what about write-write blocking? Traditional locking holds row and page locks for every modified row until the transaction commits. Under heavy write concurrency, those held locks add up fast.

Azure SQL Database addresses this issue with **optimized locking**, which is always enabled and works alongside RCSI. It uses two mechanisms:

- **Transaction ID (TID) locking:** Instead of holding individual key or row locks for every modified row, the engine takes a single exclusive lock on the transaction ID (TID). Other transactions that need to access the same row acquire a shared lock on the TID and wait for the modifying transaction to complete. The result is far fewer locks held in memory, which also makes lock escalation much less likely.
- **Lock after qualification:** Before a row is modified, the engine reads the latest committed version without acquiring a lock and checks whether the row matches the query predicate. Only rows that actually qualify get locked, and that exclusive lock is released as soon as the row update is complete, not at the end of the transaction. Lock after qualification (LAQ) requires RCSI to be enabled.

The practical effect is significant. Page and row locks for modifications are released as soon as each row is updated rather than held until the transaction commits. Concurrent writes that touch different

rows rarely block each other, even under peak load.

Choose the right isolation level

With six isolation levels available, how do you decide? Start with the defaults. For most Azure SQL Database workloads, the combination of RCSI and optimized locking already provides the best balance of consistency and performance. Readers don't block writers. Writers don't block readers. Lock memory stays low. You don't need to change anything.

Step up to SNAPSHOT isolation when your application runs multiple queries within a single transaction and needs consistent results across all of them. Think of a financial report that queries account balances and then totals them: you need both queries to see the same point-in-time data.

Reserve SERIALIZABLE for scenarios where phantom reads would cause real business errors, like double-booking inventory or miscalculating a reconciliation. Accept that throughput drops and deadlocks become more likely.

Important

Regardless of which level you choose, keep transactions as short as possible. Long-running transactions hold locks for extended periods, increase blocking, and consume version store space. A transaction that stays open while waiting for user input or an external API call can bring concurrency to a halt.

Key takeaways

Transaction isolation levels control the trade-off between data consistency and concurrency. Higher isolation prevents more side effects but increases blocking. Azure SQL Database enables RCSI by default, which eliminates read-write blocking by using row versioning instead of shared locks. Optimized locking further reduces blocking by releasing row and page locks as soon as each row is updated. For most workloads, the default combination of RCSI and optimized locking provides the best balance. Reserve SERIALIZABLE for scenarios where phantom reads would cause business logic errors.

4. Evaluate query performance with execution plans and DMVs

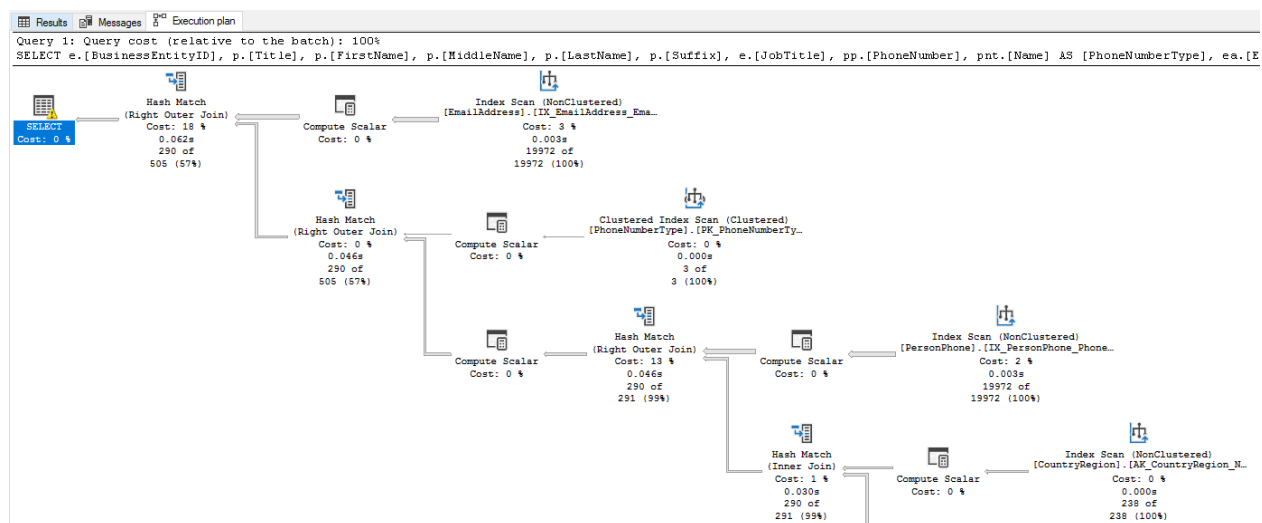
Evaluate query performance with execution plans and DMVs

Completed

When a query runs slower than expected, the first step is understanding *how* the database engine executes it. Execution plans show you the exact operators, data access methods, and resource costs the optimizer chose for a query. Dynamic management views (DMVs) complement that by exposing runtime performance data across all queries in the database, so you can find the most expensive ones before diving into any single plan.

Read execution plans

An **execution plan** is the set of instructions the query optimizer produces to retrieve and process data. It defines which tables to access first, whether to use indexes or scan tables, and how to join, filter, sort, and aggregate results. The optimizer evaluates multiple candidate plans and selects the one with the lowest estimated cost.



There are two types of execution plans:

- **Estimated execution plan:** Generated without running the query. It shows the planned operators and estimated row counts based on statistics. Use estimated plans for quick analysis without affecting the database.
- **Actual execution plan:** Captured during query execution. It includes the estimated plan plus real row counts, actual execution times, memory grants, and warnings. The actual plan reveals discrepancies between what the optimizer expected and what actually happened.

To display an estimated plan, run `SET SHOWPLAN_XML ON` before the query or select **Display Estimated Execution Plan** in SQL Server Management Studio (SSMS). To capture an actual plan, run `SET STATISTICS XML ON` or select **Include Actual Execution Plan** in SSMS before executing the query.

Even though the estimated and actual plans look similar, the actual plan's runtime metrics are crucial for diagnosing performance issues. For example, if the estimated row count for a table scan is 100 but the actual row count is 10,000, that could indicate outdated statistics leading to a bad plan choice. The optimizer compiles the plan based on statistics the first time it encounters a query. If those statistics don't reflect the current data distribution, the plan may perform poorly.

Identify common issues in execution plans

Execution plans are read left to right, top to bottom. The first operators access the base tables, and the final operator produces the query result. Look for these common issues:

Operator types tell you how the engine accesses data. There are many operator types, each representing a different method of retrieving or processing data. For example, an **Index Seek** operator represents a highly efficient method that targets specific rows using index keys. A **Table Scan** or **Index Scan** operator on the other hand, represents a less efficient method that reads every row. If you see a scan on a large table, you likely need an index. For example, if the e-commerce application queries orders by date and the plan shows a Clustered Index Scan on the `Orders` table, adding a nonclustered index on the `OrderDate` column could change that scan into a seek. Do note that not all scans are bad. If a table is small, or the search condition returns most rows in a table, a scan might be the most efficient access method. Always consider the context of the query and the size of the data. Know your data and use execution plans to confirm whether the access method makes sense.

Estimated versus actual row counts reveal whether the optimizer's assumptions match reality. The optimizer bases its plan on **statistics**, metadata that describes the distribution and density of data in your tables. If those statistics are stale, the estimated and actual row counts diverge. When the optimizer underestimates row counts, it might choose a **nested loop join** (which processes one row at a time from the inner table of a join) when a **hash join** (which builds a hash table in memory for fast lookups) would be faster, or allocate too little memory for a sort operation. Statistics can become stale after significant data changes, so updating statistics with `UPDATE STATISTICS` or enabling automatic statistics updates can help the optimizer make better decisions.

Key Lookup operators appear when the engine finds rows through a nonclustered index but needs extra columns from the clustered index. For every matching row, the engine performs an extra round trip to the clustered index to retrieve those columns. If the filter returns many rows, those extra lookups add up fast. For example, if the e-commerce application filters orders by `CustomerID` but also selects `OrderDate`, `TotalAmount`, and `ShippingAddress`, and the nonclustered index on `CustomerID` doesn't include those columns, the plan shows a Key Lookup for each matching order. You can eliminate Key Lookups by adding the missing columns as included columns in the index. Keep in mind that included columns increase index size, which can slow down writes, so weigh the read performance benefit against the write overhead.

Thick arrows between operators represent the number of rows flowing between them. An unexpectedly thick arrow early in the plan (reading from left to right, top to bottom) often means a missing filter or index is letting too many rows through.

Missing index suggestions appear as green highlighted text at the top of the graphical execution plan in SSMS. When the optimizer detects that an index could significantly reduce the cost of a query, it surfaces a recommendation directly in the plan. Right-click the suggestion and select **Missing Index Details** to generate a `CREATE INDEX` statement you can review and run. These suggestions are one of the easiest wins you can get from reading an execution plan.

Warnings appear as a yellow triangle with an exclamation mark (⚠) on the affected operator. Each warning points to an optimization opportunity. Common warnings include:

- **Missing statistics:** The optimizer couldn't find statistics for a column, so it guessed at row counts instead of using actual data distribution. To fix this issue, create statistics on the columns used in your queries or update existing statistics if they're stale.
- **Excessive memory grant:** The query requested more memory than it needed, wasting resources that other queries could use. This issue often happens when the optimizer overestimates row counts. Updating statistics or rewriting the query to filter rows earlier can help reduce memory grants.
- **No Join Predicate:** Two tables are joined without a proper condition, producing a Cartesian product that returns every possible row combination. Check your query for a missing or incorrect `ON` clause.
- **Implicit conversion:** A data type mismatch forces the engine to convert values at runtime, which can turn an index seek into a scan. For example, if a `WHERE` clause compares an `nvarchar` parameter to a `varchar` column, the engine converts every row in the column to `nvarchar` before comparing. To fix implicit conversions, match the data types in your query parameters to the column definitions.
- **Sort or Hash spill:** A sort or hash operation ran out of its granted memory and spilled intermediate results to tempdb. These operations are the second most common driver of high CPU after scans. If you see a spill warning, the optimizer likely underestimated row counts and requested too little memory. Running `UPDATE STATISTICS` to refresh the table's statistics or rewriting the query to reduce the number of rows before the sort can often eliminate the spill.

Execution plans are a powerful tool for understanding query performance. They show you exactly how the engine executes a query and where the bottlenecks are. By learning to read execution plans effectively, you can quickly identify and fix performance issues in your database queries.

Query DMVs for runtime performance data

DMVs expose real-time and accumulated performance data from the database engine. Azure SQL Database requires `VIEW DATABASE STATE` permission to query them. While execution plans show you how a single query runs, DMVs show you what's happening across all queries, which helps you find the most expensive ones first.

Find the most expensive queries

CPU time, logical reads, and execution count are the most common metrics to identify expensive queries. High CPU time or logical reads indicate a query is resource-intensive, while a high execution count means even a moderately expensive query can have a large impact on overall performance. Start by reviewing the top queries by average CPU time or logical reads to find candidates for optimization.

`sys.dm_exec_query_stats` returns aggregate performance statistics for cached query plans. Join it with `sys.dm_exec_sql_text` to see the query text and `sys.dm_exec_query_plan` to retrieve the execution plan.

The following query finds the top 10 queries by average CPU time:

```
SELECT TOP 10
    qs.total_worker_time / qs.execution_count AS avg_cpu_time,
    qs.execution_count,
    qs.total_logical_reads / qs.execution_count AS avg_logical_reads,
    SUBSTRING(st.text, (qs.statement_start_offset / 2) + 1,
        ((CASE qs.statement_end_offset
            WHEN -1 THEN DATALENGTH(st.text)
            ELSE qs.statement_end_offset
        END - qs.statement_start_offset) / 2) + 1) AS query_text
FROM sys.dm_exec_query_stats AS qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
ORDER BY avg_cpu_time DESC;
```

This script helps you identify which queries deserve your attention. High `avg_logical_reads` relative to the result set size often points to missing indexes or inefficient plans. However, be cautious when interpreting these results. A query with high average CPU time that only runs once a day might matter less than a moderate query that runs thousands of times per hour. Always consider both the average cost and the execution count when you prioritize. You can also order by `avg_logical_reads` to find queries that are heavy on I/O, which often indicates missing indexes or inefficient access methods.

Check currently executing queries

While the previous query shows you the most expensive historic queries in the plan cache, `sys.dm_exec_requests` gives you a snapshot of every request that is currently running. It includes columns for CPU time, reads, writes, wait type, wait time, and blocking session ID. Use this view to spot active queries that are consuming too many resources or stuck waiting on locks. This view is one of the most important DMVs for real-time performance monitoring and troubleshooting.

```
SELECT
    r.session_id,
    r.status,
    r.command,
    r.wait_type,
    r.wait_time,
    r.blocking_session_id,
    r.cpu_time,
```

```

        r.logical_reads,
        t.text AS query_text
FROM sys.dm_exec_requests AS r
CROSS APPLY sys.dm_exec_sql_text(r.sql_handle) AS t
WHERE r.session_id > 50
ORDER BY r.cpu_time DESC;

```

This query filters out system sessions (session IDs 1-50) and orders by CPU time. You can also order by `logical_reads` to find I/O-heavy queries. The `wait_type` and `wait_time` columns help you identify whether a query is waiting on locks, I/O, or other resources.

Discover missing indexes

Earlier, we saw how execution plans can show you missing index suggestions for a single query. The missing index DMVs give you a broader view of which indexes the optimizer would use across all queries if they existed. These views are a great way to find optimization opportunities that affect multiple queries. `sys.dm_db_missing_index_details` shows the table, equality and inequality columns, and included columns. `sys.dm_db_missing_index_group_stats` provides an improvement measure that estimates the cost reduction.

```

SELECT
    mid.statement AS table_name,
    mid.equality_columns,
    mid.inequality_columns,
    mid.included_columns,
    migs.avg_total_user_cost * migs.avg_user_impact *
        (migs.user_seeks + migs.user_scans) AS improvement_measure
FROM sys.dm_db_missing_index_groups AS mig
INNER JOIN sys.dm_db_missing_index_group_stats AS migs
    ON migs.group_handle = mig.index_group_handle
INNER JOIN sys.dm_db_missing_index_details AS mid
    ON mig.index_handle = mid.index_handle
ORDER BY improvement_measure DESC;

```

This query calculates an `improvement_measure` for each missing index recommendation, which is a product of the average cost of queries that would benefit from the index, the average percentage improvement, and the number of times those queries were executed. Sorting by this measure helps you prioritize which missing indexes to create first. However, remember that these results are just recommendations based on the queries currently in the plan cache. Always review the suggested index columns and test their impact on both query performance and write overhead before adding them to production.

Note

Missing index recommendations are suggestions, not directives. Always test the impact of a new index on both query performance and write overhead before adding it to production.

Monitor active sessions and waiting tasks

`sys.dm_exec_sessions` gives you information about all authenticated sessions, including sign in time, host name, program name, and cumulative CPU and reads. Combine it with `sys.dm_os_waiting_tasks` to see which tasks are waiting and what resources they're waiting on. These views become essential when you diagnose blocking and resource contention in a later unit.

Put it all together

Execution plans and DMVs give you a complete picture of query behavior. Start with DMVs to identify the most expensive queries. Then drill into their execution plans to understand *why* they're expensive. Is it a missing index causing a scan? Outdated statistics causing row estimate errors? A Key Lookup you can eliminate? This systematic approach, from system-wide view to individual query analysis, is the most efficient way to find and fix performance bottlenecks.

Key takeaways

Execution plans reveal the optimizer's strategy for a query, and actual plans include runtime metrics that expose discrepancies between estimated and actual row counts. When you read a plan, focus on operator types (seek vs. scan), row count estimates, warnings, and Key Lookup operators. DMVs provide system-wide performance data: use `sys.dm_exec_query_stats` to find the most expensive queries, `sys.dm_exec_requests` for currently running queries, and the missing index DMVs for optimization opportunities. Start broad with DMVs to identify where the biggest problems are, then drill into individual execution plans to understand why.

5. Monitor and tune queries with Query Store and Query Performance Insight

<https://learn.microsoft.com/en-us/training/modules/optimize-database-performance/05-monitor-tune-queries-query-store>

Monitor and tune queries with Query Store and Query Performance Insight

Completed

Real-time monitoring tells you what's happening now, but what about yesterday? Last week? After a deployment? You need a historical perspective to diagnose performance regressions, because you

can only understand what changed by comparing current behavior against a baseline. **Query Store** gives you exactly this capability.

Understand Query Store architecture

Query Store captures query text, execution plans, and runtime statistics directly inside the database engine. It stores this data in three internal stores:

- **Plan store:** Persists execution plans for each query. A single query can have multiple plans over time due to statistics updates, schema changes, or index modifications.
- **Runtime stats store:** Records aggregate execution statistics (CPU time, duration, logical reads, row counts) for each plan, grouped into configurable time intervals.
- **Wait stats store:** Captures wait statistics per query and plan, categorized by wait type. This links query performance directly to the resources the query waits on.

Query Store is enabled by default in Azure SQL Database. Data is written asynchronously to minimize overhead on the production workload. You can configure the retention period, maximum storage size, and capture mode using `ALTER DATABASE SET QUERY_STORE` options.

Consider a common scenario: your team deploys a code update on Tuesday. By Thursday, users report that a key page is slow. Without Query Store, you'd need to reproduce the issue and compare current execution plans against plans you that were never saved. With Query Store, you open the Regressed Queries view and immediately see which queries switched to slower plans after Tuesday's deployment.

Detect regressed queries

One of the most valuable capabilities of Query Store is detecting queries whose performance degraded after a plan change. The **Regressed Queries** view in SQL Server Management Studio (SSMS) surfaces queries that recently switched to a slower execution plan.

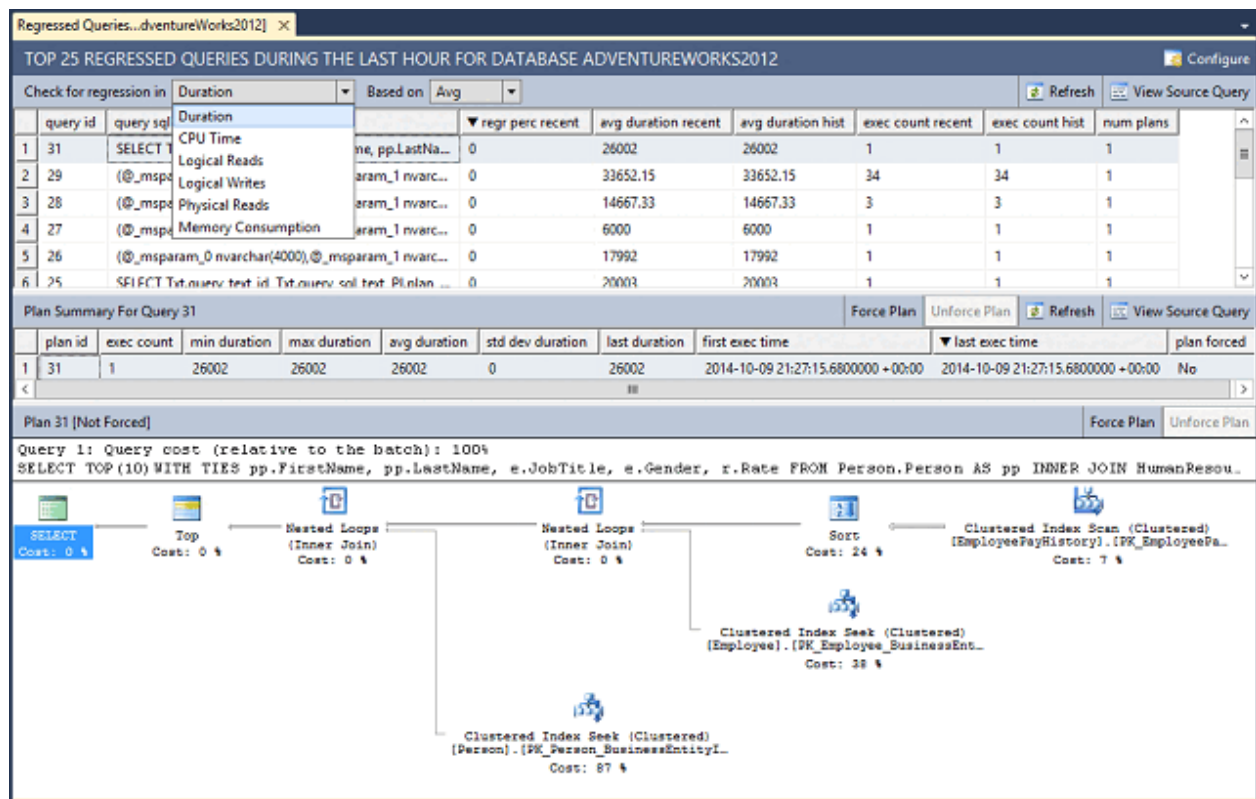
To use this view:

1. In SSMS, expand the database node in Object Explorer.
2. Expand the **Query Store** folder.
3. Select **Regressed Queries**.

This view displays queries where runtime metrics are worse than in a previous time interval. Use the dropdown lists at the top to filter by metric (CPU Time, Duration, Logical Reads, Memory Consumption, Execution Count, and more), time range, and aggregation method (Average, Maximum, Total).

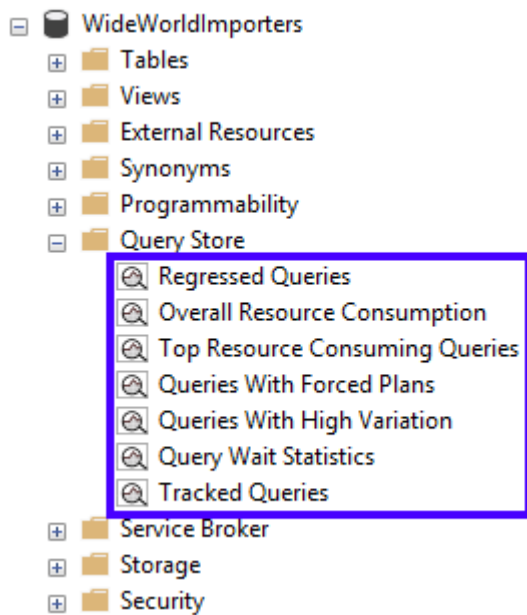
Selecting a query shows its plan history as a set of points in a chart. Each point represents a different execution plan. Select two plans to compare them side by side. This comparison is where the real investigation happens. Look for operators that changed between the fast plan and the slow plan.

Common patterns include a new plan switching from an index seek to a scan, losing a key lookup that was previously efficient, or introducing a sort operator that spills to disk.



The Query Store folder in Object Explorer contains several other views beyond Regressed Queries:

- **Top Resource Consuming Queries:** Shows the queries with the highest resource usage for a chosen metric and time range. This report is the most common starting point for day-to-day performance tuning.
- **Queries With High Variation:** Surfaces queries whose performance fluctuates significantly, which often indicates parameter sensitivity or varying data distributions.
- **Queries With Forced Plans:** Lists all currently forced plans so you can review and manage them.
- **Query Wait Statistics:** Groups *wait statistics* by category and show which queries contribute to each wait type.



Query the Query Store with T-SQL

You can also query Query Store data directly using its catalog views. The following query finds the top 10 queries by average duration in the last hour:

```
SELECT TOP 10
    qt.query_sql_text,
    q.query_id,
    p.plan_id,
    ROUND(CONVERT(FLOAT, SUM(rs.avg_duration * rs.count_executions))
        / NULLIF(SUM(rs.count_executions), 0), 2) AS avg_duration,
    SUM(rs.count_executions) AS total_executions
FROM sys.query_store_query_text AS qt
INNER JOIN sys.query_store_query AS q
    ON qt.query_text_id = q.query_text_id
INNER JOIN sys.query_store_plan AS p
    ON q.query_id = p.query_id
INNER JOIN sys.query_store_runtime_stats AS rs
    ON p.plan_id = rs.plan_id
WHERE rs.last_execution_time > DATEADD(HOUR, -1, GETUTCDATE())
GROUP BY qt.query_sql_text, q.query_id, p.plan_id
ORDER BY avg_duration DESC;
```

This query joins four catalog views: `sys.query_store_query_text` (query text), `sys.query_store_query` (query metadata), `sys.query_store_plan` (execution plans), and `sys.query_store_runtime_stats` (runtime statistics). You can adapt it to find queries by CPU time, logical reads, or execution count by changing the metric in the `SELECT` and `ORDER BY` clauses.

Force a plan

When you identify that a previous plan was better, you can tell the optimizer to use that specific plan for future executions. This approach gives you an immediate fix without modifying the query or adding hints to the application code.

To force a plan:

```
EXEC sp_query_store_force_plan
    @query_id = 42,
    @plan_id = 17;
```

To unforce a plan, and let the optimizer choose freely again:

```
EXEC sp_query_store_unforce_plan
    @query_id = 42,
    @plan_id = 17;
```

Plan forcing is useful during deployments. If a statistics update or schema change causes a plan regression, you can force the previously working plan immediately while you investigate the root cause. Think of it as a performance rollback that doesn't require an application rollback.

Apply Query Store hints

Sometimes you need to shape query execution without modifying application code. **Query Store hints** let you attach a query hint to a specific query through Query Store, and the optimizer applies it during compilation.

For example, to limit a query to a single thread:

```
EXEC sp_query_store_set_hints
    @query_id = 42,
    @query_hints = N'OPTION (MAXDOP 1)';
```

You can also force a recompile on every execution, which is useful for queries with parameter sensitivity issues:

```
EXEC sp_query_store_set_hints
    @query_id = 42,
    @query_hints = N'OPTION (RECOMPILE)';
```

You can even combine multiple hints in a single call:

```
EXEC sp_query_store_set_hints
    @query_id = 42,
    @query_hints = N'OPTION (MAXDOP 1, MAX_GRANT_PERCENT = 10)';
```

To remove a hint:

```
EXEC sp_query_store_clear_hints @query_id = 42;
```

Query Store hints override hard-coded statement-level hints and existing plan guides, so they give you full control without touching the application code.

Analyze wait statistics per query

Query Store captures **wait statistics** per query, not just at the server level. Enable wait stats capture with:

```
ALTER DATABASE CURRENT  
SET QUERY_STORE (WAIT_STATS_CAPTURE_MODE = ON);
```

The **Query Wait Statistics** view in SSMS groups waits into categories such as CPU, Lock, Buffer IO, and Memory. Selecting a wait category reveals which queries contribute the most to that wait type.

This connection between waits and queries is a significant advantage over server-level wait statistics, which can't attribute waits to specific queries. For example, if you see high Lock waits, you can drill into the specific queries causing them. You can then examine their execution plans, check their isolation levels, or evaluate whether missing indexes are causing long-held locks.

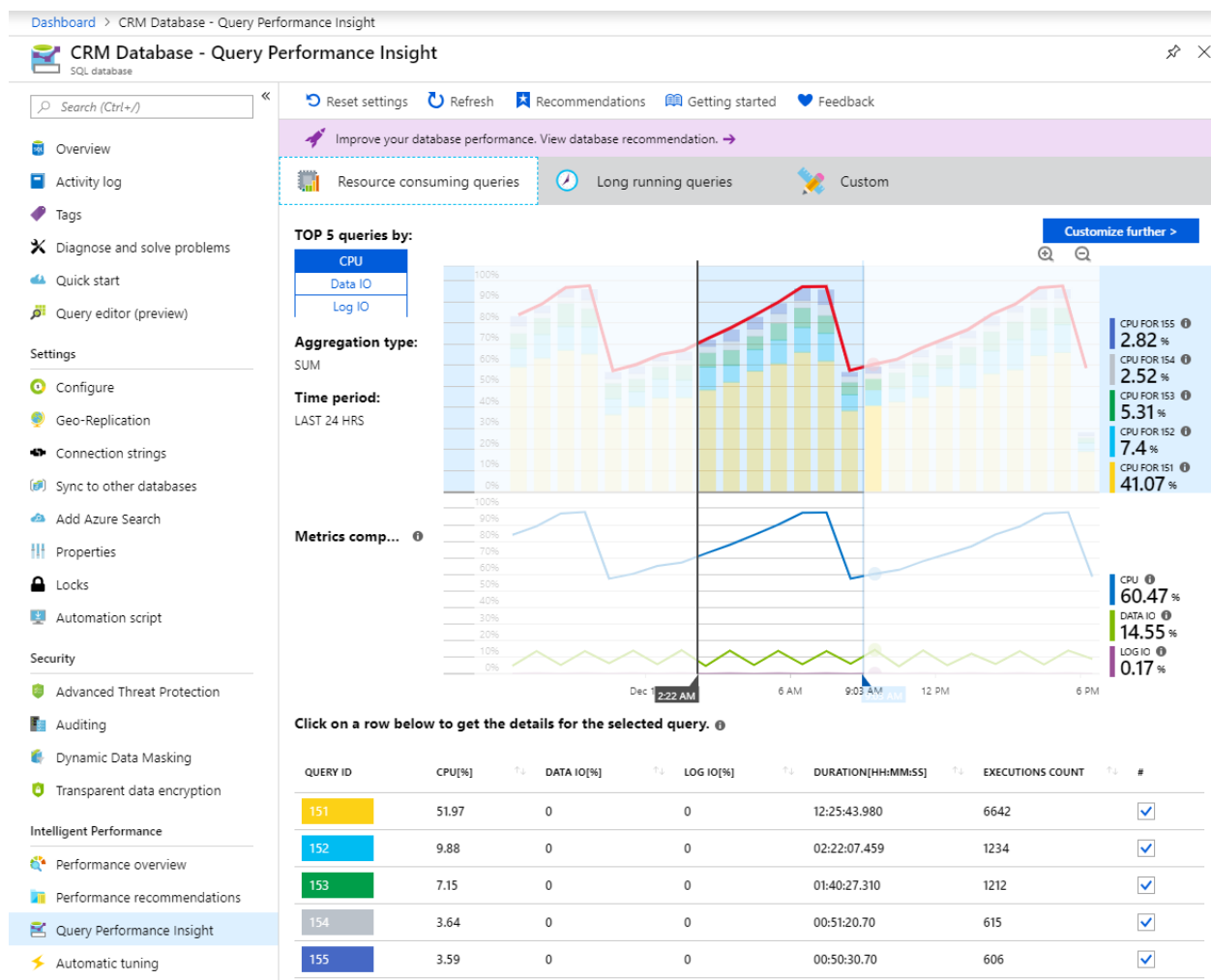
Monitor with Query Performance Insight

Query Performance Insight (QPI) is an Azure portal feature that visualizes Query Store data. It gives you a graphical view of the top resource-consuming queries without requiring SSMS.

To access QPI:

1. Open your database in the Azure portal.
2. In the left menu, select **Intelligent Performance** > **Query Performance Insight**.

By default, QPI shows the top five CPU-consuming queries. The top line in the chart represents overall Database Transaction Unit (DTU) percentage for the database. The following bars show CPU percentage consumed by each selected query during the selected interval. This layout lets you see at a glance whether a single query dominates resource usage or whether the load is spread across many queries.



Select the **Custom** tab to change the view. You can switch the metric to **Duration** or **Execution count**, adjust the time interval (last 24 hours, past week, or past month), change the number of queries displayed (5, 10, or 20), and choose an aggregation function (Sum, Max, Min, or Avg). This flexibility lets you investigate different dimensions of the same workload without leaving the portal.

Selecting an individual query opens a detail view with three charts: CPU consumption over time, duration over time, and execution count over time. This drill-down helps you distinguish between a query that's slow on every execution and one that's fast most of the time but spikes occasionally.

QPI also integrates with **Database Advisor** recommendations. Performance annotations appear on the chart as icons with a vertical line. Hover over an annotation to see a summary of the recommendation, such as creating an index or parameterizing queries. Select the icon to open the full recommendation detail and apply it directly from the portal.

Note

Query Performance Insight is limited to displaying the top 5 to 20 queries. If your workload includes many smaller queries that collectively consume significant resources, they don't appear in QPI. For deeper analysis, query the Query Store catalog views directly or use SSMS.

Note

Query Performance Insight requires Query Store to be active. If Query Store runs out of space and enters read-only mode, QPI can't display new data. Increase the Query Store maximum size or clear old data to restore normal operation.

Follow best practices

The default capture mode, **Auto**, is the right choice for most workloads. It filters out infrequent queries and queries with negligible resource consumption, which keeps storage usage manageable. Pair this setting with **size-based cleanup set to Auto**, so the database automatically removes the oldest data when Query Store approaches its maximum size.

Build a habit of checking the **Regressed Queries** view after every deployment, statistics update, or index change. Plan regressions often appear within hours and are easier to fix when you catch them early. When you do find a regression, use plan forcing as a short-term fix while you investigate the root cause. Long-term, address the underlying issue through index tuning, statistics updates, or query rewrites.

The most common operational problem with Query Store is running out of space. When that happens, Query Store silently switches to **read-only mode** and stops collecting new data. You can detect this issue by checking `sys.database_query_store_options`:

```
SELECT actual_state_desc, desired_state_desc,  
       current_storage_size_mb, max_storage_size_mb,  
       readonly_reason  
FROM sys.database_query_store_options;
```

If `actual_state_desc` shows `READ_ONLY` while `desired_state_desc` shows `READ_WRITE`, Query Store switched itself. The `readonly_reason` column tells you why. To restore normal operation, increase the maximum storage size or clear old data:

```
ALTER DATABASE CURRENT  
SET QUERY_STORE (MAX_STORAGE_SIZE_MB = 1024);
```

Key takeaways

Query Store captures query text, execution plans, and runtime statistics over time, and it's enabled by default in Azure SQL Database. The Regressed Queries view surfaces queries whose performance degraded after a plan change, making it essential to check after every deployment or schema change. When you identify a regression, plan forcing gives you an immediate fix without modifying application code, while Query Store hints let you attach query-level hints to shape execution behavior. For teams working in the Azure portal rather than SSMS, Query Performance Insight provides a graphical view of the same Query Store data.

6. Identify and resolve blocking and deadlocks

<https://learn.microsoft.com/en-us/training/modules/optimize-database-performance/06-identify-resolve-blocking-deadlocks>

Identify and resolve blocking and deadlocks

Completed

Blocking is normal in a database that uses locking. One transaction holds a lock. Another transaction waits. Brief blocking that lasts a few milliseconds is expected. It becomes a problem when it lasts long enough to affect users. Deadlocks are the more severe form: two transactions permanently block each other, and the database engine has to terminate one to break the cycle.

Blocking

Blocking occurs when one session holds a lock on a resource and another session requests a conflicting lock on the same resource. The requesting session waits until the first session releases its lock.

Resources can be rows, pages, or even entire tables. Locks can be shared (for reads) or exclusive (for writes). When a session requests a lock that conflicts with an existing lock, it becomes blocked until the first session commits or rolls back its transaction and releases its locks.

Identify blocking chains

In Azure SQL Database, Read Committed Snapshot Isolation (RCSI) is enabled by default, so read operations use row versioning instead of shared locks. This setting significantly reduces blocking between readers and writers. However, blocking between two writers, or blocking caused by explicit transactions with higher isolation levels, still happens.

The session at the top of a blocking chain is called the **head blocker**. All other blocked sessions are waiting, directly or indirectly, for the head blocker to release its locks. To find the head blocker, query `sys.dm_exec_requests` and look for sessions where `blocking_session_id` is nonzero:

```
SELECT
    r.session_id,
    r.blocking_session_id,
    r.wait_type,
    r.wait_time,
    r.wait_resource,
    t.text AS query_text,
    r.status,
```

```

r.command
FROM sys.dm_exec_requests AS r
CROSS APPLY sys.dm_exec_sql_text(r.sql_handle) AS t
WHERE r.blocking_session_id <> 0
ORDER BY r.wait_time DESC;

```

To find the head blocker in those results, look for the session ID that blocks others but isn't blocked itself. For example, suppose the query returns these rows:

session_id	blocking_session_id
55	52
60	52
52	0

Sessions 55 and 60 are both blocked by session 52, and session 52 has a `blocking_session_id` of 0, which means nothing is blocking it. Session 52 is the head blocker. Once you identify it, query `sys.dm_exec_sessions` and `sys.dm_exec_requests` filtered to that session ID to see what it's running and why it's holding locks.

Keep in mind that RCSI eliminates blocking between readers and writers, but not between two writers. Consider a scenario where session 52 runs a batch update inside an explicit transaction:

```

-- Session 52
BEGIN TRANSACTION;
UPDATE Orders SET Status = 'Processing' WHERE Region = 'West';
-- Transaction stays open while application does other work

```

This update acquires exclusive locks on every matching row. Now session 55 tries to update one of those same rows:

```

-- Session 55
UPDATE Orders SET Priority = 1 WHERE OrderId = 4820;

```

Session 55 waits because session 52 already holds an exclusive lock on that row and hasn't committed. A `SELECT` query against those rows would still succeed under RCSI because it reads a row version instead of requesting a shared lock. With RCSI removing reader-writer blocking by default, the blocking you encounter in Azure SQL Database typically involves two sessions that both need to write to the same rows.

Recognize common blocking scenarios

Understanding *why* blocking happens helps you prevent it. The Azure SQL Database engine automatically manages locks, but certain patterns lead to longer blocking:

A long-running query that holds locks for an extended period. The query is actively executing and making progress, but it holds locks the entire time. Other sessions that need conflicting locks on

the same resources wait until the query finishes. To resolve this issue, look for ways to optimize the query, such as adding indexes or rewriting it to touch fewer rows.

A sleeping session with an uncommitted transaction. A session executes a statement within an explicit transaction, then stops executing but never commits or rolls back. The locks from the transaction remain held indefinitely. This issue often happens when an application experiences a query timeout or cancellation but doesn't issue a corresponding `ROLLBACK`. Use `SET XACT_ABORT ON` so that runtime errors automatically roll back the transaction.

A session that didn't fetch all result rows. An application sends a query but doesn't retrieve all the rows from the result set. Locks can remain held on rows that haven't been fetched yet. Make sure your application fetches all result rows to completion.

A session in a rollback state. A query that was terminated (with `KILL` or by a deadlock) is rolling back its changes. Rollback can take significant time for large modifications, and the session continues to hold locks during this process. Wait for the rollback to complete, and avoid large batch modifications during busy periods.

An orphaned connection. A client application crashes or a workstation restarts without cleanly closing the database connection. The server doesn't immediately detect the disconnection, so locks from that session remain held. Terminate the orphaned session with `KILL <session_id>;`.

Note

Two of these scenarios are mitigated in Azure SQL Database by default. RCSI reduces the impact of unfetched result rows because `SELECT` queries don't acquire shared locks under row versioning, so unfetched rows don't block writers. Accelerated Database Recovery (ADR) makes lengthy rollbacks rare because it can undo changes almost instantaneously regardless of transaction size. The remaining three scenarios (long-running queries, sleeping sessions with uncommitted transactions, and orphaned connections) remain fully relevant because they involve exclusive write locks that RCSI and ADR can't release early.

Resolve active blocking

When you find active blocking:

1. Identify the head blocker using the DMV query shown earlier.
2. Determine whether the blocking session's transaction can finish on its own or whether it's waiting on external input.
3. If the blocking session is an orphaned or abandoned connection, terminate it with `KILL <session_id>;`
4. Review the blocking query's execution plan for optimization opportunities such as missing indexes.

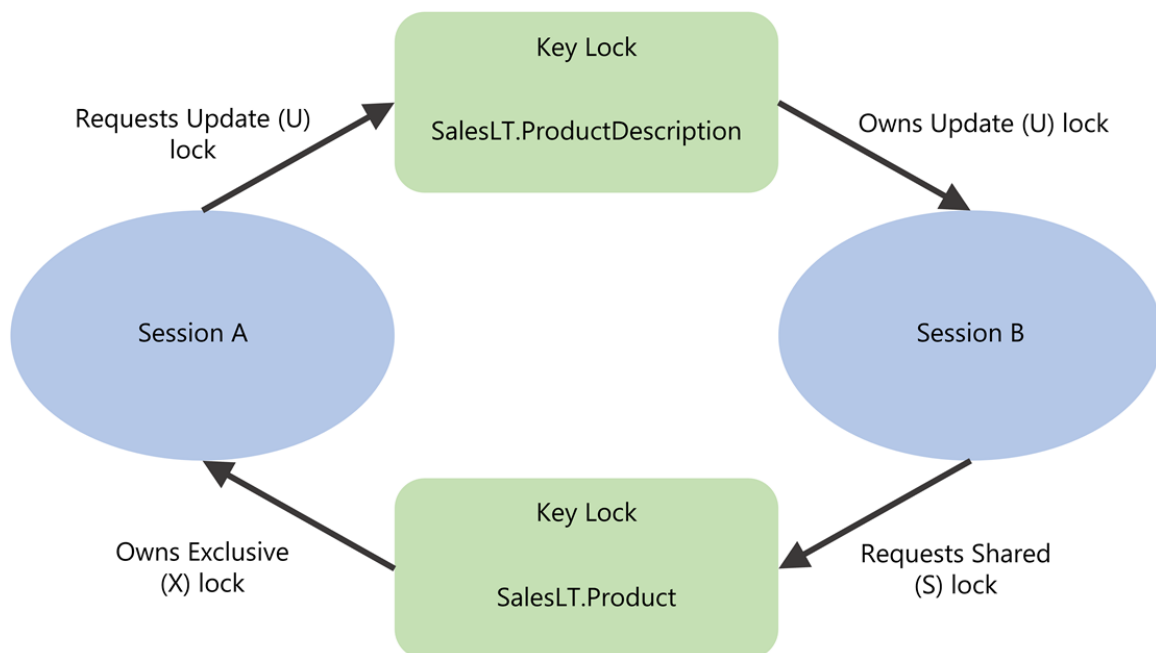
To prevent blocking from recurring, keep transactions short. Execute only the minimum required statements within a transaction and commit immediately. Use `SET XACT_ABORT ON` in your

application code so that any runtime error automatically rolls back the entire transaction, which prevents half-completed transactions from holding locks indefinitely. Move all user-facing logic outside transaction boundaries.

Deadlocks

A **deadlock** occurs when two or more transactions form a circular dependency. Each transaction holds a lock that the other needs, and neither can proceed. Here's a concrete example:

1. Transaction A updates row 1 and acquires an exclusive lock.
2. Transaction B updates row 2 and acquires an exclusive lock.
3. Transaction A tries to update row 2 and is blocked by Transaction B.
4. Transaction B tries to update row 1 and is blocked by Transaction A.



Neither transaction can finish. The database engine's **deadlock monitor** periodically checks for these cycles, with a default interval of five seconds that drops to as low as 100 milliseconds when deadlocks are frequent. When it detects a cycle, it chooses the transaction that's least expensive to roll back as the **victim**, rolls it back, and returns error 1205 to the application. This rollback allows the other transaction to complete.

Capture deadlock information

In SQL Server and Azure SQL Managed Instance, the `system_health` Extended Events session captures deadlock events by default. You can query the deadlock report from its ring buffer using `sys.dm_xe_session_targets` and `sys.dm_xe_sessions`.

In Azure SQL Database, the approach is different. You create a custom Extended Events session that captures the `sqlserver.database_xml_deadlock_report` event, and query it using the database-

scoped DMVs `sys.dm_xe_database_sessions` and `sys.dm_xe_database_session_targets`. The following example creates a deadlock capture session and queries its ring buffer:

```
-- Create and start the session
CREATE EVENT SESSION [deadlocks] ON DATABASE
ADD EVENT sqlserver.database_xml_deadlock_report
ADD TARGET package0.ring_buffer
WITH (STARTUP_STATE = ON, MAX_MEMORY = 4 MB);
GO

ALTER EVENT SESSION [deadlocks] ON DATABASE STATE = START;
GO

-- Query deadlock events from the ring buffer
DECLARE @tracename sysname = N'deadlocks';

SELECT
    d.value('/event/@timestamp')[1], 'datetime2') AS deadlock_time,
    d.query('/event/data[@name=' 'xml_report' ']/value/deadlock') AS deadlock_xml
FROM (
    SELECT CAST(target_data AS XML) AS rb
    FROM sys.dm_xe_database_sessions AS s
    INNER JOIN sys.dm_xe_database_session_targets AS t
        ON CAST(t.event_session_address AS BINARY(8)) = CAST(s.address AS BINARY(8))
    WHERE s.name = @tracename
        AND t.target_name = N'ring_buffer'
) AS ring_buffer
CROSS APPLY rb.nodes(
    '/RingBufferTarget/event[@name=' 'database_xml_deadlock_report' ']'
) AS xevent(d)
ORDER BY deadlock_time DESC;
```

The deadlock graph includes three sections. The **victim-list** identifies which transaction was terminated. The **process-list** shows each process involved, including the query text, isolation level, and lock mode. The **resource-list** shows the locked resources and which process owns and waits on each one.

In Azure SQL Database, you can also configure deadlock alerts through the Azure portal to receive notifications when deadlocks occur.

Prevent deadlocks

You can't eliminate all deadlocks, but you can significantly reduce how often they occur:

- **Access objects in a consistent order:** If all transactions modify Table A before Table B, circular dependencies can't form. Standardize access patterns through stored procedures.
- **Keep transactions short:** Shorter transactions hold locks for less time, which reduces the window for circular dependencies.
- **Use row-versioning isolation levels:** RCSI eliminates shared locks for read operations, which removes one common source of deadlock cycles. Optimized locking in Azure SQL Database further reduces deadlock likelihood.

- **Add appropriate indexes:** When queries scan many rows, they acquire locks across a wide range of data. Adding indexes that narrow the scan to fewer rows reduces lock conflicts.
- **Use plan forcing with Query Store:** If a plan change caused a query to scan more rows and acquire more locks, forcing the previous plan can reduce deadlocks while you investigate.

Handle deadlocks in application code

Applications should always include retry logic for deadlock errors. When a transaction is chosen as the deadlock victim, the database engine rolls it back and returns error 1205. Your application should catch this error, pause briefly, and resubmit the transaction.

```
BEGIN TRY
    BEGIN TRANSACTION;
    -- Your data modification statements
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    IF ERROR_NUMBER() = 1205
    BEGIN
        ROLLBACK TRANSACTION;
        WAITFOR DELAY '00:00:01'; -- Brief pause before retry
        -- Retry logic here
    END
    ELSE
    BEGIN
        ROLLBACK TRANSACTION;
        THROW;
    END
END CATCH;
```

Tip

Randomize the retry delay between attempts to prevent the same two transactions from deadlocking each other again immediately. A common pattern is to wait between one and three seconds with a random component.

Key takeaways

Blocking is normal, but extended blocking affects users, so you use `sys.dm_exec_requests` to find the head blocker and understand what it's doing. Common scenarios include long-running queries, sleeping sessions with uncommitted transactions, and orphaned connections, all of which you address by keeping transactions short, using `SET XACT_ABORT ON`, and ensuring applications properly manage connections and result sets. Deadlocks form when transactions create circular lock dependencies, and the database engine resolves them automatically by terminating the least expensive transaction and returning error 1205. You reduce deadlock frequency by accessing objects in a consistent order, keeping transactions short, using row-versioning isolation levels, and adding

appropriate indexes. Your application code should always include retry logic for error 1205 so it can recover automatically when chosen as a deadlock victim.

7. Exercise: Optimize query performance

<https://learn.microsoft.com/en-us/training/modules/optimize-database-performance/07-exercise-optimize-query-performance>

Exercise: Optimize query performance

Completed

In this exercise, you investigate slow queries in Azure SQL Database using execution plans, dynamic management views (DMVs), and Query Store. You identify a missing index through the execution plan, simulate a parameter sniffing regression with a stored procedure, use the Query Store GUI to detect and fix the regression, apply a Query Store hint, and diagnose a blocking scenario.

Note

To complete this exercise, you need an [Azure subscription](#).

Launch the exercise and follow the instructions.

[Launch Exercise](#)

8. Knowledge check

<https://learn.microsoft.com/en-us/training/modules/optimize-database-performance/08-knowledge-check>

Knowledge check

Completed

9. Summary

Summary

Completed

Performance optimization in Azure SQL Database is a systematic process. You start with infrastructure decisions, work through concurrency controls, and apply diagnostic tools to find and fix the problems that affect your users.

In this module, you learned how to:

- **Recommend database configurations:** Evaluate vCore versus DTU resource models. Choose among General Purpose, Business Critical, and Hyperscale service tiers based on I/O latency, storage, and availability needs. Select provisioned or serverless compute to match workload patterns.
- **Preserve data integrity with isolation levels:** Understand the trade-off between consistency and concurrency across six isolation levels. Use RCSI and optimized locking (both enabled by default in Azure SQL Database) to minimize blocking.
- **Evaluate query performance:** Read execution plans to identify scans, row estimate errors, Key Lookups, and warnings. Query DMVs to find the most expensive queries, currently running requests, and missing indexes.
- **Monitor and tune with Query Store:** Force previous plans for immediate fixes. Apply Query Store hints without modifying application code. Visualize performance in the Azure portal with Query Performance Insight.
- **Identify and resolve blocking and deadlocks:** Find head blockers with `sys.dm_exec_requests`. Capture deadlock graphs through Extended Events. Prevent concurrency issues by keeping transactions short, accessing objects in consistent order, and implementing retry logic for error 1205.

Learn more

- [vCore purchasing model - Azure SQL Database](#)
- [Transaction locking and row versioning guide](#)
- [Execution plan overview](#)
- [Monitor performance by using the Query Store](#)
- [Query Performance Insight for Azure SQL Database](#)
- [Understand and resolve blocking in Azure SQL Database](#)
- [Deadlocks guide](#)

Implement CI/CD by using SQL Database Projects

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/implement-cicd-sql-database-projects/01-introduction>

Introduction

Completed

A manual deployment process for database changes is slow and error-prone. Someone writes a script, emails it to the database administrator, and waits for it to be run during a maintenance window. If something goes wrong, rolling back is hard because no one tracked the exact state of the schema before the change. Meanwhile, developers working on different features overwrite each other's changes because there's no structured workflow for database code.

Imagine a team managing an e-commerce application on Azure SQL Database. Three developers are adding features that require schema changes. One adds a new product category table, another modifies the order processing stored procedures, and a third updates the customer lookup views. Without source control, CI/CD pipelines, and automated testing, these changes collide. A hotfix applied directly to production falls out of sync with the development environment. A deployment overwrites the hotfix, causing a production outage during peak hours.

SQL Database Projects and CI/CD pipelines solve these problems by treating database code with the same rigor as application code. You version-control every object, automate builds and deployments, detect when the live database drifts from the project, and test changes before they reach production.

After completing this module, you're able to:

- Create, build, and validate database models by using SQL Database Projects, including SDK-style.
- Configure source control for SQL Database Projects and manage reference data with predeployment and post-deployment scripts.
- Manage branching, pull requests, and conflict resolution for database code.
- Detect schema drift by using schema comparison tools and SqlPackage.
- Implement CI/CD pipelines with GitHub Actions and Azure DevOps, including secrets management and deployment controls.

- Design and implement a testing strategy with unit tests and integration tests.

2. Create, build, and validate SQL Database Projects

<https://learn.microsoft.com/en-us/training/modules/implement-cicd-sql-database-projects/02-create-build-validate-sql-database-projects>

Create, build, and validate SQL Database Projects

Completed

Before you can automate deployments, run tests, or detect schema drift, you need something a pipeline can actually build. That something is a SQL database project.

Understand SQL database projects

Think of a SQL database project as your database in file form. Every table, stored procedure, view, and function lives in its own `.sql` file. You declare each object once, and when you need to add a column or change a data type, you edit that single file.

If you worked with migration scripts before, the difference is significant. Migration scripts are sequential. You write an `ALTER` statement, hope it runs in the right order, and accumulate a growing chain of changes over time. A SQL database project takes the opposite approach: you maintain the desired end state of each object, and the tooling calculates the difference between the project and the target database at deployment time.

When you build the project, the output is a `.dacpac` file, a compiled model of your entire database schema. You publish that `.dacpac` to create a new database or update an existing one. The build process gives you two things that matter for CI/CD:

- **Validation:** object references and T-SQL syntax are checked against a specific version of SQL before anything gets deployed.
- **A deployable artifact:** the `.dacpac` becomes the single package that flows through your pipeline to every environment.

Choose between original and SDK-style projects

SQL database projects come in two formats. The **original** format is built on MSBuild (.NET Framework) and ships with SQL Server Data Tools (SSDT) in Visual Studio. The **SDK-style** format is

built on the `Microsoft.Build.Sql` project SDK and is the format used by the SQL Database Projects extension for Visual Studio Code.

For new projects, go with SDK-style. It brings capabilities the original format doesn't have:

- **.NET 8+ support**, so you can build cross-platform on Windows, Linux, and macOS. This support matters when your CI runners aren't Windows machines.
- **NuGet package references** for database references, so dependency management follows the same patterns as the rest of the .NET ecosystem.
- **Default globbing** for `.sql` files. Drop a file in the project folder and it automatically is included in the build. No manual file entries needed.

If you have an existing original project, you can convert it to SDK-style by modifying the `.sqlproj` file. Before you convert the project, back up the project file and archive a `.dacpac` from the current project. Compare a "before" and "after" `.dacpac` to confirm the conversion preserved everything.

Note

SDK-style SQL projects are generally available in Visual Studio Code and in preview in Visual Studio 2022. Visual Studio 2026 supports only the original SQL project format.

Create and populate a project

You can create a SQL database project from Visual Studio Code, Visual Studio, or the command line. The command line is especially handy for CI/CD scenarios where no graphical environment is available.

To create a new SDK-style project:

```
dotnet new sqlproj -n MyDatabaseProject
```

 This generates a `.sqlproj` file with the SDK-style format. From here, you add database objects by dropping `.sql` files into the project folder. The default globbing pattern picks them up automatically.

For example, to define a table, create `Tables/Customers.sql`:

```
CREATE TABLE [dbo].[Customers]
(
    [CustomerID] INT NOT NULL PRIMARY KEY,
    [FirstName] NVARCHAR(50) NOT NULL,
    [LastName] NVARCHAR(50) NOT NULL,
    [Email] NVARCHAR(100) NULL
);
```

Starting with an existing database? Schema comparison tools let you compare a live database against an empty project and import the differences, so you don't have to recreate every object by hand.

Build and validate the project

Building is where the safety net kicks in. The build process validates every object reference and checks T-SQL syntax against the target platform. If a view references a column that doesn't exist, the build fails. If you use a vector function added in SQL Server 2025 but your project targets SQL Server 2017 (Sql140), the build catches it.

To build from the command line:

```
dotnet build MyDatabaseProject.sqlproj
```

 The build produces a `.dacpac` file in the `bin/Debug` folder by default.

Build output includes **errors** (which block the build) and **warnings** (such as inconsistent casing in object names). Warnings don't stop the build, but they're worth paying attention to. You can also enable SQL code analysis rules to flag best practice violations during the build, catching issues like deprecated join syntax, `SELECT *` in views, or unindexed columns in `IN` predicates.

The target platform is set in the `.sqlproj` file and controls which T-SQL features pass validation. Set it to match your deployment target, whether that's SQL Server 2025 or Azure SQL Database.

Deploy the project

Once you have a `.dacpac`, `SqlPackage` handles the deployment. Install it as a .NET global tool:

```
dotnet tool install --global microsoft.sqlpackage
```

Then publish to a target database:

```
sqlpackage /Action:Publish /SourceFile:bin/Debug/MyDatabaseProject.dacpac /TargetCo
```

 `SqlPackage` is cross-platform and runs on Windows, Linux, and macOS.

What happens during deployment depends on the target. For a **new database**, `SqlPackage` navigates the object dependency graph and creates each object in the correct order, such as referenced tables before foreign keys. For an **existing database**, it calculates the diff between the `.dacpac` and the live schema, then generates only the `ALTER` statements needed to close the gap. Missing two columns? You get one `ALTER TABLE` with both additions. The process is idempotent. Deploy the same `.dacpac` five times and the fifth run changes nothing. You can also fan out a single `.dacpac` across a fleet of databases when upgrading multiple tenants.

Before deploying directly, you can preview the planned changes:

```
sqlpackage /Action:Script /SourceFile:bin/Debug/MyDatabaseProject.dacpac /TargetCo  
sqlpackage /Action:DeployReport /SourceFile:bin/Debug/MyDatabaseProject.dacpac /Ta
```

The `Script` action produces the exact T-SQL that would run. The `DeployReport` action produces an XML summary of every `CREATE`, `ALTER`, and `DROP`. Review either one before pushing changes to production.

Key takeaways

A SQL database project stores every database object as a declarative `.sql` file, and the build compiles them into a single `.dacpac` artifact. SDK-style projects (`Microsoft.Build.Sql`) support .NET 8+, cross-platform builds, NuGet package references, and default globbing for `.sql` files. The build process validates object references and T-SQL syntax against the target platform before anything reaches a database. `SqlPackage` calculates the difference between the `.dacpac` and the target database, generating only the statements needed to close the gap. Use the `Script` and `DeployReport` actions to preview planned changes before deploying to production. The SQL database project, its `.dacpac` artifact, and `SqlPackage` form the foundation for everything ahead, including source control, CI/CD pipelines, schema drift detection, and testing.

3. Configure source control and manage reference data

<https://learn.microsoft.com/en-us/training/modules/implement-cicd-sql-database-projects/03-configure-source-control-manage-reference-data>

Configure source control and manage reference data

Completed

A SQL database project without source control is just a folder of files on someone's machine. The moment that person's laptop breaks, or two developers edit the same stored procedure, you're back to square one. Placing the project in Git gives you change history, collaboration through branching and pull requests, and the foundation every CI/CD pipeline needs.

But source control alone doesn't solve everything. Your database probably has lookup tables, status codes, and default configurations that live alongside the schema. You need a strategy for that data too.

Set up source control for a SQL database project

Because a SQL database project is just `.sql` files and a `.sqlproj` file, it fits naturally into Git. Each database object lives in its own file, so Git tracks changes at the object level. A commit that modifies the `Customers` table and updates a stored procedure shows exactly those two files in the diff, not a monolithic migration script with dozens of unrelated changes.

To get started, initialize a Git repository in the project folder:

```
cd MyDatabaseProject
git init
git add .
git commit -m "Initial commit of database project"
```

Organize the project folder

A consistent folder structure makes the project navigable at a glance. The most common pattern groups files by object type:

```
MyDatabaseProject/
├── MyDatabaseProject.sqlproj
├── Tables/
│   ├── Customers.sql
│   └── Orders.sql
├── Views/
│   └── vw_ActiveCustomers.sql
├── StoredProcedures/
│   └── usp_GetCustomerOrders.sql
├── Scripts/
│   ├── PostDeployment/
│   │   └── seed-data.sql
│   ├── PreDeployment/
│   │   └── prep-db.sql
└── PostDeploy.sql
```

With SDK-style projects, the default globbing pattern picks up every `.sql` file in the folder automatically. No manual file entries needed.

Add a `.gitignore` file

Keep the repository focused on source files by excluding build outputs and user-specific settings:

```
bin/
obj/
*.dacpac
*.user
```

The `.dacpac` gets recreated from the project on every CI/CD build, so there's no reason to track it in Git.

Manage reference data with predeployment and post-deployment scripts

The declarative model handles schema objects like tables, views, and stored procedures, but some data is as essential as the schema itself. Status codes, lookup tables, default configurations, region lists. If that data disappears, the application breaks. It belongs in the project, versioned alongside the objects that depend on it.

Predeployment and post-deployment scripts solve this problem. They're SQL scripts that execute during deployment but sit outside the compiled database model:

- A **pre-deployment script** runs before the deployment plan. Use it for tasks that must complete before schema changes, such as dropping constraints or migrating data.
- A **post-deployment script** runs after the deployment plan completes. Use it to populate reference data, seed lookup tables, or set application defaults.

Add scripts to the project

A project supports exactly one predeployment script and one post-deployment script. You declare them in the `.sqlproj` file with `PreDeploy` and `PostDeploy` item entries:

```
<ItemGroup>
  <PreDeploy Include="prep-db.sql" />
</ItemGroup>
<ItemGroup>
  <PostDeploy Include="PostDeploy.sql" />
</ItemGroup>
```

Use SQLCMD includes for multiple data files

One script file doesn't mean one giant file. Use SQLCMD `:r` syntax to pull in multiple files from a single entry point. A typical `PostDeploy.sql` looks like this:

```
:r .\Scripts\PostDeployment\seed-statuses.sql
:r .\Scripts\PostDeployment\seed-regions.sql
:r .\Scripts\PostDeployment\seed-app-settings.sql
```

Each referenced file needs to be excluded from the build, otherwise the build process tries to compile it as a schema object and fails. In the `.sqlproj` file, use `Build Remove` to prevent compilation and `None Include` to keep the file visible in the project:

```
<ItemGroup>
  <Build Remove="Scripts\PostDeployment\seed-statuses.sql" />
  <None Include="Scripts\PostDeployment\seed-statuses.sql" />
</ItemGroup>
```

 Repeat this pattern for each referenced file in your predeployment or post-deployment scripts.

Write idempotent reference data scripts

Post-deployment scripts run on every deployment, not just the first one. If you use plain `INSERT` statements, the second deployment fails with duplicate key violations. Use `MERGE` statements instead to make the scripts safe to run repeatedly:

```
MERGE INTO [dbo].[OrderStatuses] AS target
USING (VALUES
    (1, N'Pending'),
    (2, N'Processing'),
    (3, N'Shipped'),
    (4, N'Delivered'),
    (5, N'Cancelled')
) AS source ([StatusID], [StatusName])
ON target.[StatusID] = source.[StatusID]
WHEN MATCHED THEN
    UPDATE SET [StatusName] = source.[StatusName]
WHEN NOT MATCHED THEN
    INSERT ([StatusID], [StatusName])
    VALUES (source.[StatusID], source.[StatusName]);
```

Tip

You can validate the predeployment and post-deployment scripts after a project build by changing the `.dacpac` file extension to `.zip` and extracting it. A single `.sql` file for each script type contains the combined T-SQL content from all referenced files.

Key takeaways

Initialize a Git repository at the solution level and use a `.gitignore` file to exclude build outputs like `bin/`, `obj/`, and `.dacpac` files. Organize schema objects into folders that mirror the database structure, with one file per object. Place reference data in post-deployment scripts and use SQLCMD `:r` includes to keep each script focused on a single table. To prevent data scripts from being compiled as schema objects, use `Build Remove` and `None Include` in the `.sqlproj` file. Write `MERGE` statements instead of plain `INSERT` statements so reference data scripts are idempotent and safe to run on every deployment. Next, you put this repository to work with branching and pull request workflows.

4. Manage branching, pull requests, and conflict resolution

Manage branching, pull requests, and conflict resolution

Completed

Two developers add columns to the same table on the same day. One pushes to production, the other follows an hour later and overwrites the first change. Sound familiar? Branching, pull requests, and conflict resolution exist to prevent exactly this kind of collision.

Use feature branches for database changes

A simple branching strategy for SQL database projects follows three principles:

- Creates a **feature branch** for every change, whether it's new tables, bug fixes, or stored procedure updates.
- Merges feature branches into **main** through pull requests.
- Always keep main in a deployable state.

When you start a database change, branch off `main`. Your work is isolated from everything else in progress. You modify the relevant `.sql` files, and because each object has its own file, a commit that adds a column to `Customers` touches only `Tables/Customers.sql` and nothing else.

```
git checkout -b feature/add-customer-email
```

After completing the changes, push the branch to the remote repository:

```
git add .
git commit -m "Add Email column to Customers table"
git push origin feature/add-customer-email
```

Review changes with pull requests

A pull request signals that your changes are ready for a second pair of eyes. Because SQL database projects store each object in a separate file, the diff shows the precise T-SQL that changed. Separate files make it far easier to review than a 500-line migration script that mixes unrelated modifications.

Pull requests for database changes should address questions like:

- Does the schema change break existing queries or stored procedures?

- Are the naming conventions consistent with the rest of the project?
- Does the change require a corresponding update to a post-deployment script?
- Does the change cause data loss or require data migration?

In Azure DevOps, create a pull request from a feature branch to `main` in the **Repos** section. In GitHub, use the **Pull requests** tab.

Connect pull requests to CI builds

A pull request with a green build gives reviewers confidence. Configure the PR to trigger a CI build that compiles the SQL database project. If the build fails because of a broken reference, syntax error, or unresolved dependency, the problem gets caught before it reaches main.

In Azure DevOps, you configure a CI build trigger through a **Build Validation** branch policy. In GitHub, you set up a workflow that triggers on `pull_request` events targeting the `main` branch.

Resolve merge conflicts in SQL database projects

Conflicts happen when two branches modify the same file in ways Git can't reconcile on its own. In database projects, a common scenario is two developers editing the same `CREATE TABLE` statement. One adds a `Phone` column, the other adds `Address`, and both touch the same region of the file.

To resolve the conflict:

1. Pull the latest changes from `main` into your feature branch:

```
git checkout feature/add-customer-phone
git pull origin main
```

2. Git marks the conflicting sections in the file. Open the file and review both versions.
3. Edit the file to include both changes in the correct syntax.
4. Mark the conflict as resolved and commit:

```
git add Tables/Customers.sql
git commit -m "Resolve merge conflict: include both Phone and Address columns"
```

Tip

Reduce the frequency of merge conflicts by keeping feature branches short-lived. Merge them back to `main` as soon as the change is complete and reviewed. Long-running branches that diverge significantly from `main` are harder to merge.

Validate after resolving conflicts

A clean text merge doesn't guarantee a valid schema. Suppose one branch renames a column while another branch adds a stored procedure that references the old column name. Git merges the files without complaint, but the project is broken. Always rebuild after resolving conflicts:

```
dotnet build MyDatabaseProject.sqlproj
```

 A successful build confirms that object references are intact and the T-SQL syntax is correct after the merge.

Key takeaways

Use short-lived feature branches to isolate database changes, and merge them back to `main` through pull requests. Configure CI builds as pull request checks so a successful `dotnet build` validates every proposed change before merging. Merge conflicts in `.sql` files are typically straightforward because each file contains a single object declaration. Always rebuild the project after resolving merge conflicts to verify that all object references are intact. Next, you learn how to detect when the live database falls out of sync with your project.

5. Detect and resolve schema drift

<https://learn.microsoft.com/en-us/training/modules/implement-cicd-sql-database-projects/05-detect-resolve-schema-drift>

Detect and resolve schema drift

Completed

Your SQL database project says the `Customers` table has 12 columns. Production has 13. Someone added a `LoyaltyTier` column directly through SQL Server Management Studio (SSMS) last Thursday during an incident. Your next deployment will quietly drop that column because the project doesn't know it exists. This type of situation is known as schema drift, and it's one of the ways CI/CD pipelines break production without warning.

Understand schema drift

Schema drift is the gap between what your project defines and what actually exists in the live database. Common causes include:

- **Manual changes:** someone opens a query window and runs an `ALTER TABLE` outside the normal workflow.

- **Emergency hotfixes:** a production issue gets patched at 2 AM, and the fix never makes it back into the project.
- **Third-party tools:** monitoring agents or Object-Relational Mapping (ORM) frameworks that create or modify objects behind the scenes.

The danger isn't the drift itself. It's what happens next. Your pipeline deploys the `.dacpac`, `SqlPackage` calculates the diff, and every object the project doesn't know about gets dropped. Detecting drift before that deployment runs is critical.

Detect drift with schema comparison

Schema comparison tools let you hold two database definitions side by side. You can compare any combination of a live database, a SQL database project, or a `.dacpac` file. When you compare a live database against your project, the results surface every difference, grouped by action type (`Create`, `Alter`, `Delete`) with the source and target definitions shown for each object.

In Visual Studio or Visual Studio Code, launch schema compare from the SQL Server menu or the Database Projects view. Point the source at the live database and the target at the project. The comparison grid shows what's different and what would change if you brought them into alignment.

Note

The direction is reversible. Compare database-to-project to find drift. Compare project-to-database to preview what a deployment would change. Same tool, opposite perspective.

Schema comparison options

Not every difference matters. You can tune the comparison to focus on what's meaningful:

- **Ignore whitespace** to skip formatting-only differences.
- **Ignore column order** when column position is irrelevant to your application.
- **Block on possible data loss** to flag destructive operations like dropping columns that contain data.

Save your comparison settings as an `.scmp` file and commit it to your repository. It stores the source, target, comparison options, and excluded object types, making drift checks repeatable across the team.

Import changes from the database into the project

Once you spotted a drift, you need to decide: does the live database have the right version, or does the project? If production's hotfix was correct, bring it into the project so your source of truth stays accurate.

Use schema compare to apply changes

In the comparison results, select the specific differences you want to import and apply them. The graphical interface in Visual Studio and Visual Studio Code lets you cherry-pick. Accept the hotfix index but ignore the monitoring agent's temp table, for example.

Automate with SqlPackage Extract

For CI/CD scenarios or scheduled drift checks, use SqlPackage `Extract` to pull the live schema into files:

```
sqlpackage /Action:Extract /SourceConnectionString:"{connection string}" /TargetFile
```

 This extracts database objects into files organized by schema and object type. With the project folder under Git, running `git status` after the extract reveals exactly which files changed. That's your automated drift report.

Three commands give you a drift count:

```
rm -rf MyDatabaseProject
sqlpackage /Action:Extract /SourceConnectionString:"{connection string}" /TargetFile
git status --porcelain | wc -l
```

 The output is the number of files that changed, giving you an automated drift count.

Review deployment changes before applying them

You saw the `Script` and `DeployReport` actions in the earlier unit on building and deploying. In a drift-detection context, these same commands help you verify what SqlPackage would do before you run it against production.

Generate a deployment script

The `Script` action produces the exact T-SQL that would execute, without actually running it:

```
dotnet build MyProject.sqlproj
sqlpackage /Action:Script /SourceFile:bin/Debug/MyProject.dacpac /TargetConnection
```

Review `Deployment.sql` to see every `ALTER`, `CREATE`, and `DROP` that the deployment would execute. Store the script as a pipeline artifact for approval before running it against production.

Generate a deployment report

The `DeployReport` action produces an XML summary of planned operations:

```
sqlpackage /Action:DeployReport /SourceFile:bin/Debug/MyProject.dacpac /TargetConnec
```

The report lists operations like `Create`, `Alter`, `Drop`, and `TableDataMotion`. Parse the XML in your pipeline to trigger alerts on high-impact events like data motion, clustered index rebuilds, or column drops:

```
<DeploymentReport xmlns="http://schemas.microsoft.com/sqlserver/dac/DeployReport/2016/05/01" >
  <Alerts />
  <Operations>
    <Operation Name="Create">
      <Item Value="[dbo].[Products].[IX_Products_CategorySlug]" Type="SqlIndex" />
    </Operation>
    <Operation Name="Alter">
      <Item Value="[dbo].[Customers]" Type="SqlTable" />
    </Operation>
  </Operations>
</DeploymentReport>
```

Verify .dacpac consistency with DacpacVerify

DacpacVerify compares two `.dacpac` files and reports differences between them, including predeployment scripts, post-deployment scripts, SQLCMD variables, database references, properties, and database objects. This comparison is useful when converting from an original SQL Server Data Tools (SSDT) project to SDK-style, or when validating that a pipeline produces the expected artifact.

Install it as a .NET global tool and run it against two `.dacpac` files:

```
dotnet tool install -g microsoft.dacpacverify
dacpacverify before.dacpac after.dacpac
```

Key takeaways

Schema drift occurs when the live database diverges from the SQL project, often caused by hotfixes, manual changes, or direct production edits. Schema comparison tools detect differences between a database and a project, letting you choose which changes to pull into the project or push to the database. Save schema comparison settings in `.scmp` files so the team uses consistent options every time. Automate drift detection by running `SqlPackage /Action:Extract` on a schedule and comparing the extracted project against the repository. Use `SqlPackage /Action:DeployReport` to preview every planned `CREATE`, `ALTER`, and `DROP` before applying changes. The goal isn't to prevent drift entirely, but to detect it early and resolve it before your next deployment turns a hotfix into a rollback. Next, you build the CI/CD pipelines that automate the entire build-and-deploy cycle.

6. Implement CI/CD pipelines

Implement CI/CD pipelines

Completed

Up to this point, every step in the process was manual. Build the project, run `SqlPackage`, check the deployment report. A CI/CD pipeline turns that sequence into something that happens automatically on every commit, with the same steps, in the same order, every time. No forgotten flags, no typos in connection strings, no "it worked on my machine."

Design the pipeline structure

Most database pipelines follow a two-stage pattern:

1. **Build:** compile the SQL database project and produce the `.dacpac` artifact.
2. **Deploy:** publish that `.dacpac` to target databases, moving through environments (development, staging, production).

Building once and deploying the same artifact everywhere eliminates the "works in dev, breaks in prod" problem. The `.dacpac` that passed validation in staging is the same file that gets deployed to production.

Implement a CI/CD pipeline with GitHub Actions

GitHub Actions workflows live in `.github/workflows/` and define your pipeline as YAML. The `azure/sql-action` action handles `.dacpac` deployment to Azure SQL Database.

Here's a workflow that builds on every push to `main` and deploys to production:

```
# .github/workflows/sql-deploy.yml
name: Build and Deploy SQL Project
on:
  push:
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Build SQL project
        run: dotnet build ./Database.sqlproj -o ./output
```

```

- name: Upload dacpac artifact
  uses: actions/upload-artifact@v4
  with:
    name: dacpac
    path: ./output/Database.dacpac

deploy:
  needs: build
  runs-on: ubuntu-latest
  environment: production
  steps:
    - name: Download dacpac artifact
      uses: actions/download-artifact@v4
      with:
        name: dacpac

    - name: Install SqlPackage
      run: dotnet tool install -g microsoft.sqlpackage

    - uses: azure/login@v2
      with:
        client-id: ${ secrets.AZURE_CLIENT_ID }
        tenant-id: ${ secrets.AZURE_TENANT_ID }
        subscription-id: ${ secrets.AZURE_SUBSCRIPTION_ID }

    - uses: azure/sql-action@v2.3
      with:
        connection-string: ${ secrets.AZURE_SQL_CONNECTION_STRING }
        path: './Database.dacpac'
        action: 'publish'

```

 The `azure/sql-action` supports `.dacpac`, `.sqlproj`, and `.sql` scripts. It works with SQL authentication, Microsoft Entra ID, and service principal authentication.

If your Azure SQL Database has firewall rules enabled, the GitHub Actions runner's IP needs access. When you pair `azure/login` with `azure/sql-action`, the action adds a temporary firewall rule for the runner's IP during deployment and removes it afterward.

Implement a CI/CD pipeline with Azure DevOps

Azure DevOps provides the `SqlAzureDacpacDeployment` task for `.dacpac` deployment. Here's the equivalent pipeline:

```

# azure-pipelines.yml
trigger:
  - main

pool:
  vmImage: 'windows-latest'

steps:

```

```

- task: DotNetCoreCLI@2
  inputs:
    command: 'build'
    projects: './Database.sqlproj'
    arguments: '-o $(Build.ArtifactStagingDirectory)'

- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: '$(Build.ArtifactStagingDirectory)/Database.dacpac'
    ArtifactName: 'dacpac'

- task: SqlAzureDacpacDeployment@1
  inputs:
    azureSubscription: 'your-service-connection'
    AuthenticationType: 'server'
    ServerName: 'your-server.database.windows.net'
    DatabaseName: 'your-database'
    SqlUsername: '$(SqlUser)'
    SqlPassword: '$(SqlPassword)'
    deployType: 'DacpacTask'
    DeploymentAction: 'Publish'
    DacpacFile: '$(Build.ArtifactStagingDirectory)/Database.dacpac'

```

For Linux agents or more control over the process, install and use SqlPackage directly:

```

steps:
- script: dotnet tool install --global microsoft.sqlpackage
  displayName: 'Install SqlPackage'

- script: |
  sqlpackage /Action:Publish \
    /SourceFile:$(Build.ArtifactStagingDirectory)/Database.dacpac \
    /TargetConnectionString:"$(ConnectionString)"
  displayName: 'Deploy database'

```

Whether you use GitHub Actions or Azure DevOps, both pipelines need credentials to connect to your database. The next step is making sure those credentials stay out of your YAML files.

A connection string hardcoded in a YAML file is a breach waiting to happen. Both GitHub Actions and Azure DevOps provide mechanisms to store and inject secrets at runtime without exposing them in source control.

GitHub repository and environment secrets

Store sensitive values as **repository secrets** or **environment secrets**. In your GitHub repository on github.com:

1. Select **Settings > Secrets and variables > Actions**.
2. Select **New repository secret**.
3. Add a name (like `AZURE_SQL_CONNECTION_STRING`) and the value.

Reference secrets in your workflow with `${{ secrets.SECRET_NAME }}`. Environment secrets are scoped to specific deployment environments, so production credentials are accessible only to jobs targeting production.

Service principal and OpenID Connect authentication

Better yet, skip passwords entirely. **OpenID Connect (OIDC)** with `azure/login` authenticates using federated credentials, with no client secret stored anywhere. You need three values:

- `AZURE_CLIENT_ID`: The application (client) ID of the service principal.
- `AZURE_TENANT_ID`: Your Microsoft Entra ID tenant ID.
- `AZURE_SUBSCRIPTION_ID`: Your Azure subscription ID.

The service principal authenticates through federated credentials, so there's no password to rotate or leak.

Azure Key Vault integration

For centralized secrets management, store connection strings and credentials in **Azure Key Vault** and have your pipeline pull them at deployment time. In Azure DevOps, the **Azure Key Vault** task fetches secrets into pipeline variables. In GitHub Actions, use `azure/login` followed by Azure CLI commands to read from the vault.

Important

Rotate database credentials regularly. Azure Key Vault can integrate with Azure Functions to automate secret rotation, reducing the risk of compromised credentials.

Implement deployment pipeline controls

A pipeline that deploys to production on every push with no review step is risky for database changes. You need controls that prevent unreviewed changes from reaching production.

Environment protection rules

Both GitHub and Azure DevOps support **environments** with protection rules that gate deployments:

- **Required reviewers**: one or more team members must approve before the deploy job runs. In GitHub, configure this setting under **Settings** > **Environments** > **Protection rules**.
- **Wait timers**: add a delay between approval and execution, giving the team a window to reconsider.
- **Deployment branches**: restrict which branches can target an environment. For example, only `main` deploys to production.

Branch policies

Branch policies protect your main branch and serve as the first-line of defense. In Azure DevOps, configure:

- **Minimum number of reviewers** for pull requests.
- **Build validation**, which runs the SQL project build as a PR check before merging.
- **Comment resolution**, requiring all review comments to be addressed.
- **Automatically included reviewers**, so specific people are added to PRs that touch database files.

GitHub provides similar protections through **branch protection rules** or **rulesets**.

Code owners

A `CODEOWNERS` file (GitHub) or automatically included reviewers policy (Azure DevOps) makes sure the right people review database changes. No SQL file gets merged without the database team seeing it:

```
# .github/CODEOWNERS
/Database/ @db-team
*.sql @db-team @dba-lead
```

This rule requires members of the `db-team` to review any pull request that modifies SQL files or the database project folder.

Branch control checks

In Azure DevOps, a **branch control check** on service connections locks down which pipelines can access production credentials. Only pipelines running in the context of `main` get access. Even if someone modifies a pipeline on a feature branch to target production, the service connection refuses the request.

Key takeaways

Use `azure/sql-action` (GitHub Actions) or `SqlAzureDacpacDeployment` (Azure DevOps) to deploy `.dacpac` files from your CI/CD pipeline. Store connection strings and credentials as repository secrets, environment secrets, or in Azure Key Vault, and never hardcode them in YAML files. To authenticate without storing passwords, use OpenID Connect (OIDC) with federated credentials. Gate production deployments with environment protection rules, required reviewers, and deployment branch restrictions. Use `CODEOWNERS` files or automatically included reviewers to ensure that the right people review the database changes.

7. Design and implement a testing strategy

Design and implement a testing strategy

Completed

A SQL database project that builds successfully doesn't mean the stored procedures inside it return the right results. A procedure can compile with no errors and still calculate the wrong totals or skip a validation check. Testing catches these problems before they reach production.

Understand database testing levels

Database testing works in layers, each catching a different class of problem:

- **Build validation** is the `dotnet build` step you already have. It catches syntax errors and broken object references. Fast, but it only proves the schema is structurally valid.
- **Unit tests** verify that individual stored procedures and functions produce the right results for a given set of inputs. They catch logic errors.
- **Integration tests** exercise end-to-end scenarios against a deployed database. They prove that objects work together correctly, but they require a running instance and take the most time.

Each layer is slower and more expensive than the one before it, but also catches problems the previous layer can't.

Create SQL Server unit tests

SQL Server Data Tools (SSDT) in Visual Studio includes a built-in framework for database unit tests. Each test executes T-SQL against a live database and validates the results using test conditions.

Structure of a unit test

Each test has three sections that follow a setup-execute-cleanup pattern:

- **Pre-test:** set up the data the test needs. Insert customer records, clear leftover data from previous runs.
- **Test:** execute the operation you're testing. Call the stored procedure and query the view.
- **Post-test:** clean up after the test so it doesn't contaminate the next run.

Test conditions

After the test T-SQL runs, **test conditions** validate what came back. The most commonly used conditions are:

- **Row Count:** verifies that the result set contains the expected number of rows.
- **Scalar Value:** verifies that a specific cell in the result set contains the expected value.
- **Expected Schema:** verifies that the result set has the expected column names and data types.

Other built-in conditions include **Data Checksum**, **Empty ResultSet**, **Not Empty ResultSet**, and **Execution Time**.

Create a unit test project

Setting up SQL Server unit tests in Visual Studio takes a few steps:

1. Open your SQL database project.
2. In **SQL Server Object Explorer**, find the stored procedure or function you want to test.
3. Right-click the object and select **Create Unit Tests**.
4. Choose or create a C# test project.
5. Set the test connection to your development database.
6. Select **Automatically deploy the database project before unit tests are run**. This option keeps the test database in sync with your latest project changes.

The designer opens with a T-SQL template where you write your test logic and attach test conditions.

Write effective database unit tests

A good unit test answers a specific question: "Does this procedure do what it supposed to for a known input?" Here's an example that tests `uspPlaceNewOrder`, verifying that placing an order correctly updates the customer's year-to-date total:

```
-- Pre-test: Set up customer and clear previous data
DECLARE @CustomerID INT;
INSERT INTO [Sales].[Customer] (CustomerName) VALUES (N'Test Customer');
SET @CustomerID = SCOPE_IDENTITY();
DELETE FROM [Sales].[Orders] WHERE [CustomerID] = @CustomerID;

-- Test: Place an order and verify YTDOrders updated correctly
DECLARE @RC INT;
EXECUTE @RC = [Sales].[uspPlaceNewOrder] @CustomerID, 100, GETDATE(), '0';

SELECT [YTDOrders]
FROM [Sales].[Customer]
WHERE [CustomerID] = @CustomerID;
```

 Pair this test T-SQL with a **Scalar Value** test condition that expects the `YTDOrders` value to be `100`.

Negative tests

You also need to verify that stored procedures fail correctly when they receive invalid inputs. For example, canceling an order that was already shipped should raise an error, not succeed silently. A **negative test** confirms that the expected error occurs.

When you select **Create Unit Tests** in step 3 of the previous section, Visual Studio generates a C# test project in **Solution Explorer** with a `.cs` file containing a test for each stored procedure you selected. For example, if you created a test for `uspCancelOrder`, the generated file includes a section named `Sales_uspCancelOrder_Test`. To mark that test as a negative test, open the `.cs` file in **Solution Explorer** and add the `ExpectedSqlException` attribute directly above it and save the file:

```
[TestMethod()]
[ExpectedSqlException(Severity = 16, MatchFirstError = false, State = 1)]
public void Sales_uspCancelOrder_FilledOrder_Test()
```

The test passes only when the stored procedure raises an error matching the specified severity and state. If the procedure succeeds silently, the test fails, which is exactly what you want.

Design an integration testing approach

Unit tests isolate individual objects. Integration tests go further and test scenarios that span multiple operations. They answer questions like:

- Does a sequence of stored procedure calls leave the database in the expected state?
- Do triggers fire correctly when data arrives through the application layer?
- Do views return accurate results after a series of related table changes?

Integration tests need a dedicated database. Deploy the SQL database project to a test instance before each run using the **Automatically deploy the database project before unit tests are run** setting. This setting keeps the test schema current.

Considerations for test databases

Both unit tests and integration tests run T-SQL against a live database, so the test environment needs careful setup to produce reliable results.

- **Isolate tests from production.** Use a separate instance or dedicated test database. Running tests against production is never recommended.
- **Reset to a known state** before each run. Post-deployment scripts or test cleanup scripts handle this issue.
- **Externalize connection strings** in configuration files so local development and CI pipelines each point to the right database without code changes.

Integrate tests into CI/CD pipelines

Add a test step after build and deployment so tests run automatically on every commit. In Azure DevOps, use the `VSTest` task:

```
- task: VSTest@2
  inputs:
```

```
testAssembly: '**\*Tests.dll'  
searchFolder: '$(System.DefaultWorkingDirectory)'
```

In GitHub Actions, run the tests using `dotnet test`:

```
- name: Run database unit tests  
  run: dotnet test ./DatabaseTests/DatabaseTests.csproj
```

Tip

Configure the test project to automatically deploy the database project before tests run. This setting ensures that the test database schema matches the project at the time of testing.

When a test fails, the pipeline stops and the change doesn't reach staging or production. This delay gives the team time to fix the issue before it affects any environment.

Key takeaways

Database testing spans three levels: build validation for structural correctness, unit tests for logic verification, and integration tests for end-to-end workflows. SQL Server unit tests follow a three-phase structure of test preactions, test actions, and test post-actions. To verify stored procedure and function behavior, use test conditions like `Scalar Value`, `Row Count`, and `Expected Schema`. To test methods that should validate error handling paths, add the `ExpectedSqlException` attribute. Wire test projects into your CI/CD pipeline so a failing test blocks the deployment before changes reach staging or production. Together, these three layers form a safety net that lets your team deploy database changes with confidence instead of anxiety.

8. Exercise: Implement CI/CD by using SQL Database Projects

<https://learn.microsoft.com/en-us/training/modules/implement-cicd-sql-database-projects/08-exercise-implement-cicd-sql-database-projects>

Exercise: Implement CI/CD by using SQL Database Projects

Completed

In this exercise, you create a SQL database project, build it locally, push it to a GitHub repository, and configure a GitHub Actions pipeline that automatically builds and deploys schema changes to Azure

SQL Database.

Note

To complete this exercise, you need an [Azure subscription](#) and a [GitHub account](#).

Launch the exercise and follow the instructions.

[Launch Exercise](#)

9. Knowledge check

<https://learn.microsoft.com/en-us/training/modules/implement-cicd-sql-database-projects/09-knowledge-check>

Knowledge check

Completed

10. Summary

<https://learn.microsoft.com/en-us/training/modules/implement-cicd-sql-database-projects/10-summary>

Summary

Completed

CI/CD for SQL Database Projects brings the same automation, consistency, and safety nets to database development that application teams rely on for their code.

In this module, you learned how to:

- **Create, build, and validate SQL Database Projects:** Define database objects in declarative T-SQL files, build them into `.dacpac` artifacts, and validate references and syntax against a target platform using the SDK-style `Microsoft.Build.Sql` project format.
- **Configure source control and manage reference data:** Place SQL database projects in Git, organize files by object type, and use predeployment and post-deployment scripts with

SQLCMD :r includes to manage reference data alongside the schema.

- **Manage branching, pull requests, and conflict resolution:** Use feature branches for database changes, review T-SQL diffs in pull requests, resolve merge conflicts at the object level, and validate merged results with a project build.
- **Detect and resolve schema drift:** Compare live databases against SQL database projects using schema comparison tools, automate drift detection with SqlPackage Extract, and review planned changes with deployment reports and scripts.
- **Implement CI/CD pipelines with deployment controls:** Build and deploy .dacpac files with GitHub Actions (azure/sql-action) and Azure DevOps (SqlAzureDacpacDeployment), manage secrets through repository secrets and Azure Key Vault, and protect production with environment approvals, branch policies, and code owners.
- **Design and implement a testing strategy:** Create SQL Server unit tests with test conditions (Row Count, Scalar Value, Expected Schema), write negative tests for error handling, and integrate tests into CI/CD pipelines to catch logic errors before deployment.

Learn more

- [What are SQL database projects?](#)
- [Get started with SQL database projects](#)
- [SQL Server Data Tools, SDK-style \(preview\)](#)
- [Predeployment and post-deployment scripts](#)
- [Schema comparison overview](#)
- [Compare a database and a project](#)
- [SQL projects automation](#)
- [About branches and branch policies](#)
- [Verify database code by using SQL Server unit tests](#)
- [Azure SQL Deploy action \(GitHub\)](#)
- [SqlAzureDacpacDeployment task reference](#)

Integrate SQL solutions with Azure services

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/1-introduction>

Introduction

Completed

Modern applications rarely interact with databases through direct SQL connections. Instead, they consume data through REST and GraphQL APIs that provide flexibility, security, and standardization. Building these API layers traditionally requires significant backend development effort—writing controllers, mapping data models, handling authentication, and managing deployment infrastructure.

Data API Builder solves this challenge by automatically generating REST and GraphQL endpoints from your database schema. With a single configuration file, you can expose tables, views, and stored procedures through secure, scalable APIs without writing backend code.

In this module, you learn how to:

- Create and configure Data API Builder configuration files for SQL databases
- Define entities with field mappings, caching, pagination, and filtering
- Configure REST and GraphQL endpoints for different client needs
- Expose views, stored procedures, and define GraphQL relationships
- Explore deployment options for Data API Builder, including Azure Container Apps, App Service, and Static Web Apps
- Set up Azure Monitor with Application Insights for API observability
- Handle database changes using Change Data Capture, Azure Functions, and Change Event Streaming

Prerequisites

- Experience with SQL Server, Azure SQL Database, or SQL databases in Microsoft Fabric
- Familiarity with T-SQL for creating tables, views, and stored procedures
- Basic understanding of REST APIs and JSON
- Access to an Azure subscription for deployment exercises
- Docker Desktop installed for local development (optional)

2. Create configuration files for Data API Builder

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/2-create-configuration-files-data-api-builder>

Create configuration files for Data API Builder

Completed

Data API Builder (DAB) is a cross-platform, open-source engine that creates modern REST and GraphQL endpoints for your database without requiring you to write custom code. With a single configuration file, you can expose your database objects through secure, scalable APIs that support authentication, authorization, and caching.

If you work with Azure SQL or SQL databases in Microsoft Fabric, you've likely faced requests to make your data accessible to application developers. Building custom API layers takes time and adds maintenance overhead. Data API Builder eliminates that work by generating APIs directly from your database schema.

Understand the Data API Builder configuration file

Everything in Data API Builder starts with a configuration file. This JSON file tells DAB how to connect to your database, which objects to expose, how to handle authentication, and how to behave at runtime. When DAB starts, it reads this configuration and generates your REST and GraphQL endpoints automatically.

A typical configuration file has four main sections:

- **\$schema** - References the JSON schema for validation and IntelliSense support
- **data-source** - Defines the database connection string and type
- **runtime** - Controls runtime behavior like host settings, authentication, and caching
- **entities** - Specifies which database objects to expose and how to configure them

Here's a basic configuration file structure:

```
{
  "$schema": "https://github.com/Azure/data-api-builder/releases/download/v1.1.7/dab-config-schema.json",
  "data-source": {
    "database-type": "mssql",
```

```

    "connection-string": "@env('DATABASE_CONNECTION_STRING')"
  },
  "runtime": {
    "rest": {
      "enabled": true,
      "path": "/api"
    },
    "graphql": {
      "enabled": true,
      "path": "/graphql"
    },
    "host": {
      "cors": {
        "origins": ["http://localhost:3000"],
        "allow-credentials": false
      },
      "authentication": {
        "provider": "StaticWebApps"
      }
    }
  },
  "entities": {}
}

```

Notice the `$schema` property at the top? That gives your editor autocompletion and validation as you type. Always use the schema version that matches your DAB installation.

Configure the data source connection

The `data-source` section tells DAB how to connect to your database. For SQL Server, Azure SQL Database, and SQL databases in Microsoft Fabric, you use `mssql` as the database type.

```

"data-source": {
  "database-type": "mssql",
  "connection-string": "@env('DATABASE_CONNECTION_STRING')",
  "options": {
    "set-session-context": true
  }
}

```

You could put your connection string directly in the configuration file, but don't. Use the `@env()` syntax instead to reference environment variables. This keeps credentials out of your configuration files and source control.

The `set-session-context` option is worth noting. When enabled, DAB passes user identity information to SQL Server through session context. Your stored procedures and functions can then read this information using `SESSION_CONTEXT()`, which is helpful for implementing row-level security or audit logging.

Tip

For local development, you can use the `dab init` command with the `--connection-string` parameter to set up your initial configuration file quickly.

Set up runtime options

The `runtime` section controls how DAB operates when serving requests. Here you configure REST and GraphQL endpoints separately, along with global host settings like CORS and caching.

```
"runtime": {
  "rest": {
    "enabled": true,
    "path": "/api",
    "request-body-strict": true
  },
  "graphql": {
    "enabled": true,
    "path": "/graphql",
    "allow-introspection": true
  },
  "host": {
    "cors": {
      "origins": ["https://myapp.azurewebsites.net"],
      "allow-credentials": true
    },
    "mode": "production"
  },
  "cache": {
    "enabled": true,
    "ttl-seconds": 5
  }
}
```

The `path` properties define the base URL path for each API type. REST endpoints appear under `/api/EntityName`, while GraphQL queries go to `/graphql`. The `allow-introspection` setting for GraphQL determines whether clients can query the schema itself, which is helpful during development but should often be disabled in production for security.

CORS (Cross-Origin Resource Sharing) settings in the `host` section specify which domains can call your API from browser-based applications. The `mode` property switches between `development` and `production` behavior, affecting features like detailed error messages.

Use the DAB CLI for configuration management

The [Data API Builder CLI](#) provides commands for creating and modifying configuration files without manual JSON editing. This approach reduces errors and ensures valid configuration syntax.

To create a new configuration file:

```
dab init --database-type mssql --connection-string "@env('DATABASE_CONNECTION_STRING')"
```

This command creates a `dab-config.json` file in your current directory with baseline settings. From there, you can add entities using CLI commands or edit the file directly.

To add an entity for a table:

```
dab add Product --source dbo.Products --permissions "anonymous:read"
```

The CLI handles JSON formatting and validates your configuration against the schema automatically. If you work in a team, using CLI commands also makes pull request reviews easier since the formatting stays consistent.

Important

Always store your configuration files in source control, but never commit actual connection strings. Use environment variable references and manage credentials through secure mechanisms like Azure Key Vault or deployment pipeline secrets.

Organize configuration for multiple environments

Most projects need different settings for development, staging, and production. You might want introspection enabled during development but disabled in production, or different CORS origins for each environment.

DAB supports this through configuration merging. Start with a base `dab-config.json` file containing shared settings, then create environment-specific overrides:

- `dab-config.development.json` - Development-specific settings
- `dab-config.staging.json` - Staging environment settings
- `dab-config.production.json` - Production settings

When you start DAB with the `--config` parameter, it merges that file's settings with your base configuration:

```
dab start --config dab-config.production.json
```

Environment-specific files only need to contain settings that differ from the base. DAB handles the merge automatically, with environment-specific values taking precedence over base settings.

3. Define entities for REST and GraphQL

Define entities for REST and GraphQL

Completed

Entities are the core building blocks of Data API Builder. Each entity maps to a database object and defines how that object appears through your REST and GraphQL APIs. When you define entities properly, you control what data clients can access, what operations they can perform, and how the data appears in API responses.

Building on the configuration file you created, the `entities` section is where you specify the tables, views, and other database objects that DAB exposes. You configure each entity with source mappings, field customizations, and relationship definitions that shape your API surface.

The examples throughout this unit use the following sample database tables. Keep these definitions in mind as you work through each configuration:

```
CREATE TABLE dbo.Categories (  
    CategoryID INT PRIMARY KEY,  
    CategoryName NVARCHAR(50) NOT NULL  
);  
  
CREATE TABLE dbo.Products (  
    ProductID INT PRIMARY KEY,  
    ProductName NVARCHAR(100) NOT NULL,  
    UnitPrice DECIMAL(10, 2) NOT NULL,  
    UnitsInStock INT NOT NULL,  
    CategoryID INT FOREIGN KEY REFERENCES dbo.Categories(CategoryID)  
);
```

Map database objects to entities

An entity definition starts with a name and a source reference. The entity name becomes the identifier used in API endpoints and GraphQL queries, while the source points to the actual database object.

```
"entities": {  
  "Product": {  
    "source": {  
      "object": "dbo.Products",  
      "type": "table"  
    },  
    "permissions": [  
      {
```

```

        "role": "anonymous",
        "actions": ["read"]
      }
    ]
  }
}

```

This configuration exposes the `dbo.Products` table as a `Product` entity. REST clients access it at `/api/Product`, while GraphQL clients query it as `product` (singular) or `products` (plural). DAB automatically generates these naming conventions based on your entity name.

The `type` property specifies the database object type: `table`, `view`, or `stored-procedure`. Each type has different capabilities. Tables support full CRUD operations (create, read, update, delete), views typically support read operations, and stored procedures execute custom logic.

Configure field mappings and aliases

Sometimes your database column names don't match the naming conventions you want in your API. Field mappings let you rename columns and control which fields appear in API responses.

```

"Product": {
  "source": {
    "object": "dbo.Products",
    "type": "table"
  },
  "mappings": {
    "ProductID": "id",
    "ProductName": "name",
    "UnitPrice": "price",
    "UnitsInStock": "stockQuantity"
  },
  "permissions": [
    {
      "role": "anonymous",
      "actions": ["read"]
    }
  ]
}

```

With these mappings, clients see `id`, `name`, `price`, and `stockQuantity` in their API responses instead of the original database column names. This approach lets you present a clean, consistent API while keeping your existing database schema unchanged.

Note

When you use field mappings, clients must use the mapped names in their queries and mutations. The original database column names become internal implementation details.

Enable caching for improved performance

[Data API Builder caching](#) reduces database load by storing query results temporarily. You can configure caching at both the global level (in the runtime section) and per-entity for fine-grained control.

```
"Product": {
  "source": {
    "object": "dbo.Products",
    "type": "table"
  },
  "rest": {
    "enabled": true
  },
  "graphql": {
    "enabled": true,
    "cache": {
      "enabled": true,
      "ttl-seconds": 60
    }
  },
  "permissions": [...]
}
```

The `ttl-seconds` property (time-to-live) determines how long cached data remains valid. For product catalogs that change infrequently, longer cache durations improve performance. For rapidly changing data like inventory counts, use shorter durations or disable caching.

Caching works differently for REST and GraphQL. REST caching considers the complete URL including query parameters, while GraphQL caching examines the query structure. Both respect the TTL you configure.

Implement pagination for large datasets

When entities contain many records, returning all of them in a single response creates performance and usability problems. Data API Builder provides built-in pagination support that you can customize per entity.

```
"Product": {
  "source": {
    "object": "dbo.Products",
    "type": "table"
  },
  "rest": {
    "enabled": true,
    "path": "/products"
  },
  "graphql": {
    "enabled": true,
```

```

    "type": {
      "singular": "product",
      "plural": "products"
    }
  },
  "permissions": [...]
}

```

REST clients use query parameters like `$first` and `$after` to navigate through pages:

```

GET /api/products?$first=10
GET /api/products?$first=10&$after=eyJQcm9kdWN0SUQiOjEwfQ==

```

The `$after` parameter contains a cursor that points to the last item from the previous page. DAB generates these cursors automatically based on your entity's primary key.

GraphQL clients use similar pagination through the `first` and `after` arguments in their queries:

```

query {
  products(first: 10, after: "eyJQcm9kdWN0SUQiOjEwfQ==") {
    items {
      id
      name
      price
    }
    hasNextPage
    endCursor
  }
}

```

The response includes `hasNextPage` and `endCursor` fields that clients use to determine if more data exists and how to fetch it.

Configure filtering and searching

Data API Builder generates filtering capabilities automatically based on your entity's fields. Clients can filter results using standard comparison operators.

For REST endpoints, filtering uses OData-style query syntax:

```

GET /api/products?$filter=price gt 100 and stockQuantity gt 0

```

For GraphQL, filtering uses typed input objects:

```

query {
  products(filter: { price: { gt: 100 }, stockQuantity: { gt: 0 } }) {
    items {
      id
      name
    }
  }
}

```

```

    price
  }
}

```

DAB generates filter types for each field based on its data type. Numeric fields support `gt`, `gte`, `lt`, `lte`, `eq`, and `neq`. String fields add `contains`, `startsWith`, and `endsWith`. You don't need to configure these capabilities explicitly; they're available automatically for all exposed fields.

Tip

For complex filtering scenarios that can't be expressed through standard operators, consider creating a view or stored procedure that encapsulates the filtering logic, then expose that as a separate entity.

Define entity relationships for GraphQL

GraphQL's strength lies in traversing relationships between entities in a single query. Data API Builder supports defining relationships that map to your database's foreign key relationships.

```

"entities": {
  "Product": {
    "source": { "object": "dbo.Products", "type": "table" },
    "relationships": {
      "category": {
        "cardinality": "one",
        "target.entity": "Category",
        "source.fields": ["CategoryID"],
        "target.fields": ["CategoryID"]
      }
    },
    "permissions": [...]
  },
  "Category": {
    "source": { "object": "dbo.Categories", "type": "table" },
    "relationships": {
      "products": {
        "cardinality": "many",
        "target.entity": "Product",
        "source.fields": ["CategoryID"],
        "target.fields": ["CategoryID"]
      }
    },
    "permissions": [...]
  }
}

```

With these relationships configured, GraphQL clients can query across entities:

```

query {
  products {
    items {
      name
      price
      category {
        name
      }
    }
  }
}

```

The `cardinality` property indicates whether the relationship returns one item or many. Use `one` for many-to-one relationships (product to category) and `many` for one-to-many relationships (category to products).

Understand REST endpoint generation

When you define an entity, DAB automatically creates REST endpoints following standard conventions. Each entity gets a base URL path, and HTTP verbs map to CRUD operations.

For an entity named `Product` with the default configuration:

HTTP Method	URL Pattern	Operation
GET	<code>/api/Product</code>	List all products (paginated)
GET	<code>/api/Product/id/123</code>	Get product with ID 123
POST	<code>/api/Product</code>	Create a new product
PUT	<code>/api/Product/id/123</code>	Update or create product 123
PATCH	<code>/api/Product/id/123</code>	Partially update product 123
DELETE	<code>/api/Product/id/123</code>	Delete product 123

The `/id/` segment in URLs indicates a primary key lookup. For composite keys, chain multiple key segments: `/api/OrderDetail/orderId/100/productId/50`.

Customize REST endpoint paths

You can modify the default URL patterns through entity configuration. The `rest` section controls REST-specific behavior:

```

"Product": {
  "source": { "object": "dbo.Products", "type": "table" },
  "rest": {
    "enabled": true,
    "path": "/products",

```

```

    "methods": {
      "get": true,
      "post": true,
      "put": false,
      "patch": true,
      "delete": true
    }
  },
  "permissions": [...]
}

```

The `path` property changes the URL from `/api/Product` to `/api/products`. This setting is useful when you want lowercase, plural endpoint names that differ from your entity names.

The `methods` object lets you enable or disable specific HTTP verbs. In this example, `PUT` is disabled, which prevents full record replacement while still allowing `PATCH` for partial updates. This configuration pattern protects against accidental data loss from clients that might send incomplete records.

Tip

Disabling methods at the endpoint level provides defense in depth. Even if permissions allow an action, disabling the method prevents it entirely.

Set up GraphQL endpoint configuration

GraphQL operates through a single endpoint, typically at `/graphql`. All queries, mutations, and subscriptions go through this endpoint, with the request body specifying the operation.

```

"runtime": {
  "graphql": {
    "enabled": true,
    "path": "/graphql",
    "allow-introspection": true,
    "depth-limit": 8,
    "multiple-mutations": {
      "create": { "enabled": true }
    }
  }
}

```

The `allow-introspection` setting controls whether clients can query the GraphQL schema itself. During development, introspection helps tools like GraphiQL and Apollo Studio understand your API. In production, consider disabling it to hide implementation details.

The `depth-limit` setting prevents overly complex nested queries that could strain your database. A depth of 8 allows reasonable relationship traversal while blocking potential abuse through deeply nested queries.

Configure GraphQL-specific entity settings

Each entity can have GraphQL-specific configuration that differs from REST:

```
"Product": {
  "source": { "object": "dbo.Products", "type": "table" },
  "graphql": {
    "enabled": true,
    "type": {
      "singular": "product",
      "plural": "products"
    },
    "operation": "query"
  },
  "rest": {
    "enabled": true
  },
  "permissions": [...]
}
```

The `type` property customizes the GraphQL type names. By default, DAB uses the entity name, but you can specify different singular and plural forms.

The `operation` property determines where mutations appear. Use `query` for read-only entities (like views) or `mutation` for entities that support write operations.

4. Expose database objects, stored procedures, and views

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/4-expose-database-objects-stored-procedures-views>

Expose database objects, stored procedures, and views

Completed

Beyond simple table mappings, Data API Builder can expose views and stored procedures as API endpoints. These database objects let you encapsulate complex business logic, aggregate data from multiple tables, or implement operations that go beyond standard CRUD patterns.

Views provide read-only access to computed or joined data, while stored procedures enable custom operations with input parameters and multiple result sets. GraphQL relationships add another dimension, allowing clients to traverse connections between entities in single queries.

Expose views as read-only entities

Database views aggregate and transform data without duplicating storage. When you expose a view through Data API Builder, clients access the computed results through familiar REST and GraphQL patterns.

Consider a view that combines product information with inventory status:

```
CREATE VIEW dbo.ProductInventory AS
SELECT
    p.ProductID,
    p.ProductName,
    p.UnitPrice,
    p.UnitsInStock,
    c.CategoryName,
    CASE
        WHEN p.UnitsInStock = 0 THEN 'Out of Stock'
        WHEN p.UnitsInStock < 10 THEN 'Low Stock'
        ELSE 'Available'
    END AS StockStatus
FROM dbo.Products p
INNER JOIN dbo.Categories c ON p.CategoryID = c.CategoryID;
```

Expose this view in your configuration:

```
"ProductInventory": {
  "source": {
    "object": "dbo.ProductInventory",
    "type": "view",
    "key-fields": ["ProductID"]
  },
  "rest": {
    "enabled": true,
    "path": "/inventory"
  },
  "graphql": {
    "enabled": true,
    "operation": "query"
  },
  "permissions": [
    {
      "role": "anonymous",
      "actions": ["read"]
    }
  ]
}
```

The `key-fields` property is required for views because DAB can't automatically detect primary keys. Specify the columns that uniquely identify each row.

Setting `operation` to `query` in the GraphQL section indicates this entity supports only read operations. DAB won't generate mutations for this entity.

Note

Views can be updatable in SQL Server if they meet certain criteria. However, exposing updatable views through DAB requires careful consideration of the underlying table permissions and constraints.

Configure stored procedures for custom operations

[Stored procedures](#) provide the most flexibility for custom database operations. They can accept parameters, perform complex logic, and return multiple result sets.

```
CREATE PROCEDURE dbo.GetProductsByPriceRange
    @MinPrice DECIMAL(10,2),
    @MaxPrice DECIMAL(10,2)
AS
BEGIN
    SELECT ProductID, ProductName, UnitPrice, UnitsInStock
    FROM dbo.Products
    WHERE UnitPrice BETWEEN @MinPrice AND @MaxPrice
    ORDER BY UnitPrice;
END;
```

Configure the stored procedure as an entity:

```
"GetProductsByPriceRange": {
  "source": {
    "object": "dbo.GetProductsByPriceRange",
    "type": "stored-procedure",
    "parameters": {
      "MinPrice": "minPrice",
      "MaxPrice": "maxPrice"
    }
  },
  "rest": {
    "enabled": true,
    "path": "/products/by-price-range",
    "methods": {
      "get": true,
      "post": true
    }
  },
  "graphql": {
    "enabled": true,
    "operation": "query"
  },
  "permissions": [
```

```
{
  "role": "authenticated",
  "actions": ["execute"]
}
```

The `parameters` object maps stored procedure parameters to API parameter names. REST clients call the procedure using query parameters or request body:

```
GET /api/products/by-price-range?minPrice=10&maxPrice=50
```

```
POST /api/products/by-price-range
Content-Type: application/json

{
  "minPrice": 10,
  "maxPrice": 50
}
```

GraphQL clients use the procedure through a generated query:

```
query {
  executeGetProductsByPriceRange(minPrice: 10, maxPrice: 50) {
    ProductID
    ProductName
    UnitPrice
  }
}
```

Expose stored procedures that modify data

For stored procedures that insert, update, or delete data, configure them with mutation operations:

```
CREATE PROCEDURE dbo.CreateOrder
  @CustomerID INT,
  @ProductID INT,
  @Quantity INT
AS
BEGIN
  DECLARE @OrderID INT;

  INSERT INTO dbo.Orders (CustomerID, OrderDate, Status)
  VALUES (@CustomerID, GETDATE(), 'Pending');

  SET @OrderID = SCOPE_IDENTITY();

  INSERT INTO dbo.OrderDetails (OrderID, ProductID, Quantity, UnitPrice)
  SELECT @OrderID, @ProductID, @Quantity, UnitPrice
  FROM dbo.Products
```

```

WHERE ProductID = @ProductID;

UPDATE dbo.Products
SET UnitsInStock = UnitsInStock - @Quantity
WHERE ProductID = @ProductID;

SELECT @OrderID AS OrderID;
END;

```

```

"CreateOrder": {
  "source": {
    "object": "dbo.CreateOrder",
    "type": "stored-procedure",
    "parameters": {
      "CustomerID": "customerId",
      "ProductID": "productId",
      "Quantity": "quantity"
    }
  },
  "rest": {
    "enabled": true,
    "methods": {
      "post": true
    }
  },
  "graphql": {
    "enabled": true,
    "operation": "mutation"
  },
  "permissions": [
    {
      "role": "authenticated",
      "actions": ["execute"]
    }
  ]
}

```

Setting `operation` to `mutation` places this procedure under GraphQL mutations rather than queries, indicating it modifies data.

Important

Stored procedures execute with the permissions of the database connection, not the calling user. Implement appropriate validation and authorization logic within the procedure or use session context for row-level security.

Define GraphQL relationships between entities

GraphQL relationships enable clients to traverse connections between entities without multiple round trips. DAB supports both one-to-one and one-to-many relationships.

```
"entities": {
  "Order": {
    "source": { "object": "dbo.Orders", "type": "table" },
    "relationships": {
      "customer": {
        "cardinality": "one",
        "target.entity": "Customer",
        "source.fields": ["CustomerID"],
        "target.fields": ["CustomerID"]
      },
      "orderDetails": {
        "cardinality": "many",
        "target.entity": "OrderDetail",
        "source.fields": ["OrderID"],
        "target.fields": ["OrderID"]
      }
    },
    "permissions": [...]
  },
  "OrderDetail": {
    "source": { "object": "dbo.OrderDetails", "type": "table" },
    "relationships": {
      "order": {
        "cardinality": "one",
        "target.entity": "Order",
        "source.fields": ["OrderID"],
        "target.fields": ["OrderID"]
      },
      "product": {
        "cardinality": "one",
        "target.entity": "Product",
        "source.fields": ["ProductID"],
        "target.fields": ["ProductID"]
      }
    },
    "permissions": [...]
  }
}
```

With these relationships, GraphQL clients can build rich queries:

```
query {
  orders(first: 10) {
    items {
      orderDate
      customer {
        companyName
        contactName
      }
      orderDetails {
```

```

    items {
      quantity
      product {
        productName
        unitPrice
      }
    }
  }
}

```

This single query retrieves orders with customer information and line items including product details. Without relationships, clients would need multiple separate requests.

Handle complex relationship scenarios

Some database designs use junction tables for many-to-many relationships. DAB handles these through explicit relationship chains.

For a products-to-suppliers many-to-many relationship through a `ProductSuppliers` junction table:

```

"Product": {
  "relationships": {
    "productSuppliers": {
      "cardinality": "many",
      "target.entity": "ProductSupplier",
      "source.fields": ["ProductID"],
      "target.fields": ["ProductID"]
    }
  }
},
"ProductSupplier": {
  "source": { "object": "dbo.ProductSuppliers", "type": "table" },
  "relationships": {
    "supplier": {
      "cardinality": "one",
      "target.entity": "Supplier",
      "source.fields": ["SupplierID"],
      "target.fields": ["SupplierID"]
    }
  }
}

```

Clients traverse the relationship chain: Product → ProductSupplier → Supplier. While this requires one extra level in queries, it accurately represents the database structure and maintains referential integrity.

5. Explore deployment options for Data API Builder

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/5-deploy-data-api-builder-azure-services>

Explore deployment options for Data API Builder

Completed

Moving Data API Builder from local development to Azure production environments requires choosing the right hosting platform and configuring deployment automation. Azure offers several options for running DAB, each with different characteristics for scalability, management overhead, and integration with other Azure services.

Your deployment strategy affects not only where DAB runs but also how you manage configuration, handle secrets, and scale to meet demand. Understanding these deployment options helps you select the approach that best fits your application architecture and operational requirements.

Deploy to Azure Container Apps

[Azure Container Apps](#) provides a serverless container platform that handles infrastructure management automatically. It's well-suited for Data API Builder because it can scale from zero when idle and automatically scale up based on request volume.

To deploy DAB to Container Apps, first create a container image with your configuration:

```
FROM mcr.microsoft.com/azure-databases/data-api-builder:latest

COPY dab-config.json /App/dab-config.json

ENV DATABASE_CONNECTION_STRING=""
ENV ASPNETCORE_URLS="http://+:5000"

EXPOSE 5000
```

This Dockerfile starts from the official DAB image and adds your configuration file. Environment variables handle the connection string and port configuration.

Deploy using the Azure CLI:

```
az containerapp create \
  --name dab-api \
  --resource-group myResourceGroup \
  --environment myContainerAppEnv \
  --image myregistry.azurecr.io/dab-api:v1 \
  --target-port 5000 \
  --ingress external \
  --secrets database-conn="<connection-string>" \
  --env-vars DATABASE_CONNECTION_STRING=secretref:database-conn
```

Container Apps integrates with Azure Key Vault for secret management. Reference secrets in your environment variables rather than storing credentials in configuration files or container images.

Tip

Enable Azure Container Apps managed identity and grant it access to your Azure SQL Database. This approach eliminates stored credentials entirely by using Microsoft Entra authentication.

Deploy to Azure App Service

[Azure App Service](#) offers a mature platform with built-in deployment slots, custom domains, and comprehensive monitoring. For teams already using App Service, adding DAB follows familiar patterns.

Configure App Service to run the DAB container:

```
az webapp create \
  --name dab-api \
  --resource-group myResourceGroup \
  --plan myAppServicePlan \
  --deployment-container-image-name mcr.microsoft.com/azure-databases/data-api-bui
```

Set environment variables through App Service configuration:

```
az webapp config appsettings set \
  --name dab-api \
  --resource-group myResourceGroup \
  --settings DATABASE_CONNECTION_STRING="@Microsoft.KeyVault(SecretUri=https://myva
```

The Key Vault reference syntax lets App Service pull secrets at runtime without exposing them in configuration. Enable system-assigned managed identity on the App Service and grant it access to Key Vault.

Deploy to Azure Static Web Apps

[Azure Static Web Apps](#) includes Data API Builder as a built-in feature called database connections. This integration provides the simplest deployment experience when your API complements a static frontend.

Link your database through the Azure portal or CLI:

```
az staticwebapp dbconnection create \  
  --name myStaticWebApp \  
  --resource-group myResourceGroup \  
  --db-type mssql \  
  --db-resource-id /subscriptions/.../servers/myserver/databases/mydb
```

The `staticwebapps.database.config.json` file in your repository defines entity configurations using the same structure as standalone DAB:

```
{  
  "data-source": {  
    "database-type": "mssql"  
  },  
  "entities": {  
    "Product": {  
      "source": "dbo.Products",  
      "permissions": [  
        { "role": "anonymous", "actions": ["read"] }  
      ]  
    }  
  }  
}
```

Static Web Apps automatically handles authentication integration with its built-in auth providers. API routes appear under `/.auth/` and `/data-api/` paths by default.

Note

Database connections in Static Web Apps have some limitations compared to standalone DAB, including restricted stored procedure support. Review the documentation for current feature availability.

Implement CI/CD deployment pipelines

Automated deployment pipelines ensure consistent configuration across environments and enable rapid iteration. GitHub Actions provides a straightforward approach for DAB deployments.

Create a workflow file at `.github/workflows/deploy-dab.yml`:

```
name: Deploy Data API Builder  
  
on:
```

```

push:
  branches: [main]
  paths:
    - 'dab-config*.json'
    - 'Dockerfile'

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Login to Azure Container Registry
        uses: azure/docker-login@v1
        with:
          login-server: ${ secrets.ACR_LOGIN_SERVER }
          username: ${ secrets.ACR_USERNAME }
          password: ${ secrets.ACR_PASSWORD }

      - name: Build and push container
        run: |
          docker build -t ${ secrets.ACR_LOGIN_SERVER }/dab-api:${ github.sha }
          docker push ${ secrets.ACR_LOGIN_SERVER }/dab-api:${ github.sha }

      - name: Deploy to Container Apps
        uses: azure/container-apps-deploy-action@v1
        with:
          resourceGroup: myResourceGroup
          containerAppName: dab-api
          imageToDeploy: ${ secrets.ACR_LOGIN_SERVER }/dab-api:${ github.sha }

```

This workflow triggers on changes to configuration files or the Dockerfile. It builds a new container image, pushes it to Azure Container Registry, and updates the Container App.

Configure environment-specific deployments

Production deployments require careful separation between environments. Use environment-specific configuration files combined with deployment variables.

Structure your repository with multiple configuration files:

```

/
├── dab-config.json           # Base configuration (shared settings)
├── dab-config.development.json
├── dab-config.staging.json
├── dab-config.production.json
└── Dockerfile

```

Modify your Dockerfile to accept a build argument:

```
FROM mcr.microsoft.com/azure-databases/data-api-builder:latest

ARG ENVIRONMENT=production
COPY dab-config.json /App/dab-config.json
COPY dab-config.${ENVIRONMENT}.json /App/dab-config.${ENVIRONMENT}.json

ENV DAB_ENVIRONMENT=${ENVIRONMENT}
```

Build environment-specific images:

```
docker build --build-arg ENVIRONMENT=production -t dab-api:production .
```

Important

Never include actual connection strings or secrets in configuration files committed to source control. Use environment variable references (`@env()`) and manage secrets through Azure Key Vault or your deployment pipeline's secret management.

Monitor deployment health

After deployment, configure health checks to verify DAB is operating correctly. Container Apps and App Service support health probes:

```
{
  "probes": [
    {
      "type": "liveness",
      "httpGet": {
        "path": "/",
        "port": 5000
      },
      "periodSeconds": 30
    },
    {
      "type": "readiness",
      "httpGet": {
        "path": "/api/Product?$first=1",
        "port": 5000
      },
      "periodSeconds": 10
    }
  ]
}
```

The liveness probe checks that DAB is responding. The readiness probe verifies database connectivity by making an actual API request. Configure your orchestration platform to restart unhealthy instances automatically.

6. Recommend Azure Monitor configurations

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/6-recommend-azure-monitor-configurations>

Recommend Azure Monitor configurations

Completed

Running Data API Builder in production requires monitoring to understand performance, diagnose issues, and optimize resource usage. [Azure Monitor](#) provides the comprehensive observability platform that integrates with DAB through Application Insights and Log Analytics.

Effective monitoring goes beyond simple uptime checks. You need visibility into request latencies, error rates, database query performance, and resource consumption patterns. This type of data helps you identify bottlenecks before they affect users and provides data for capacity planning.

Configure Application Insights integration

[Application Insights](#) captures detailed information from your Data API Builder instance, including request traces, dependency calls to your database, and exception details.

Enable Application Insights by setting the connection string as an environment variable:

```
APPLICATIONINSIGHTS_CONNECTION_STRING="InstrumentationKey=your-key;IngestionEndpoint=your-endpoint"
```

For Azure Container Apps or App Service deployments, configure this through application settings. DAB automatically detects the connection string and begins sending telemetry.

Once enabled, Application Insights tracks:

- **Request telemetry** - Every REST and GraphQL request with timing, status code, and response size
- **Dependency telemetry** - Database queries with execution time and success/failure status
- **Exception telemetry** - Unhandled errors with stack traces
- **Custom metrics** - Cache hit rates, connection pool usage

Tip

Set a sampling rate for high-volume production deployments to control costs while maintaining visibility. Application Insights adaptive sampling automatically adjusts based on traffic volume.

Create custom dashboards for API monitoring

Application Insights data powers custom Azure dashboards that display key performance indicators for your API.

Create a dashboard with these essential visualizations:

Request metrics:

```
requests
| where timestamp > ago(24h)
| summarize
    RequestCount = count(),
    AvgDuration = avg(duration),
    P95Duration = percentile(duration, 95),
    FailureRate = countif(success == false) * 100.0 / count()
| project RequestCount, AvgDuration, P95Duration, FailureRate
```

This query summarizes 24-hour request volume, average response time, 95th percentile latency, and failure rate. These metrics quickly indicate API health.

Top slow endpoints:

```
requests
| where timestamp > ago(1h)
| summarize
    AvgDuration = avg(duration),
    RequestCount = count()
    by name
| top 10 by AvgDuration desc
| project name, AvgDuration, RequestCount
```

Identifying slow endpoints helps prioritize optimization efforts. Slow queries might indicate missing indexes, inefficient entity configurations, or database contention.

Set up Log Analytics for detailed diagnostics

[Log Analytics](#) provides a central repository for logs from all Azure resources. Configure your DAB deployment to send logs to a Log Analytics workspace.

For Container Apps, enable logging through the environment configuration:

```
az containerapp env update \
  --name myContainerAppEnv \
  --resource-group myResourceGroup \
  --logs-workspace-id <workspace-id>
```

Query DAB container logs to investigate issues:

```
ContainerAppConsoleLogs_CL
| where ContainerName_s == "dab-api"
| where Log_s contains "error" or Log_s contains "exception"
| project TimeGenerated, Log_s
| order by TimeGenerated desc
```

Combine container logs with Application Insights telemetry for complete request traces. Correlate errors with specific requests using operation IDs that flow through both data sources.

Configure alert rules for proactive monitoring

Alert rules notify your team when API behavior deviates from normal patterns. Set up alerts for critical conditions that require immediate attention.

High error rate alert:

```
az monitor metrics alert create \
  --name "DAB-HighErrorRate" \
  --resource-group myResourceGroup \
  --scopes "/subscriptions/.../components/myAppInsights" \
  --condition "avg requests/failed > 5" \
  --window-size 5m \
  --evaluation-frequency 1m \
  --action-group myActionGroup
```

This alert triggers when the failure rate exceeds 5% over a 5-minute window. Action groups define notification channels like email, SMS, or webhooks to your incident management system.

Response time degradation alert:

```
requests
| where timestamp > ago(5m)
| summarize P95 = percentile(duration, 95) by bin(timestamp, 1m)
| where P95 > 2000
```

Create a log-based alert that fires when 95th percentile latency exceeds 2 seconds. This catches performance degradation even when requests technically succeed.

Important

Configure alerts thoughtfully to avoid alert fatigue. Start with high-severity conditions and refine thresholds based on normal operating patterns.

Monitor database query performance

Application Insights dependency tracking captures database queries issued by DAB. Use this telemetry to identify slow or frequently executed queries.

```
dependencies
| where timestamp > ago(1h)
| where type == "SQL"
| summarize
    CallCount = count(),
    AvgDuration = avg(duration),
    FailureCount = countif(success == false)
    by data
| top 20 by AvgDuration desc
```

The `data` field contains the SQL query text. Review slow queries to determine if indexes would help, if entity configurations need optimization, or if certain operations should use stored procedures instead.

Compare database metrics across time periods to identify trends:

```
dependencies
| where type == "SQL"
| summarize
    AvgDuration = avg(duration)
    by bin(timestamp, 1h)
| render timechart
```

Gradual increases in query duration might indicate growing data volumes, index fragmentation, or changing query patterns that need attention.

Implement distributed tracing

When DAB serves requests from multiple client applications, distributed tracing connects the complete request flow from client through API to database.

Enable correlation by including trace context headers in client requests:

```
GET /api/Product HTTP/1.1
traceparent: 00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-01
```

Application Insights correlates these traces, allowing you to see end-to-end latency breakdown in the Transaction Search view. This visibility is valuable for diagnosing intermittent issues that span multiple services.

For GraphQL queries that resolve multiple entities, traces show the timing of each database call, helping identify which relationships or entities contribute most to response time.

Review resource utilization metrics

Beyond application telemetry, monitor the underlying compute resources running DAB. Container Apps and App Service provide resource metrics through Azure Monitor.

Key metrics to track:

- **CPU percentage** - Sustained high CPU indicates compute-bound operations or need for scaling
- **Memory percentage** - Memory pressure can cause container restarts or degraded performance
- **Network bytes in/out** - Unusual network patterns might indicate large result sets or denial-of-service attempts
- **Replica count** - For scaled deployments, verify autoscaling responds appropriately to load

Set up autoscaling rules based on these metrics to maintain responsive APIs during traffic spikes while controlling costs during quiet periods.

7. Handle changes with event-driven patterns

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/7-handle-changes-event-driven-patterns>

Handle changes with event-driven patterns

Completed

Modern applications often need to respond to database changes in real time. When a customer places an order, updates their profile, or when inventory levels change, downstream systems need notification. SQL Server and Azure SQL provide several mechanisms for capturing and streaming these changes, each with different characteristics for latency, throughput, and implementation complexity.

Data API Builder integrates with these change mechanisms through Azure Functions SQL trigger bindings. Understanding when to use each approach helps you build responsive, event-driven architectures that react to data changes efficiently.

Understand change capture mechanisms

SQL platforms offer multiple approaches for tracking data changes, each suited to different scenarios:

- **Change Data Capture (CDC)** records insert, update, and delete operations to special change tables. It captures the complete before-and-after state of changed rows, making it ideal for audit requirements and data synchronization.

- [Change Tracking](#) provides a lighter-weight alternative that records which rows changed without capturing the actual values. Applications query change information to determine what data needs synchronization.
- [Change Event Streaming \(CES\)](#) is a newer feature that pushes changes to event streams without polling. It integrates directly with Azure Event Hubs for high-throughput scenarios.

Feature	CDC	Change Tracking	CES
Captures changed values	Yes	No	Yes
Polling required	Yes	Yes	No
Historical data	Yes	Limited	Stream only
Setup complexity	Moderate	Low	Moderate

Enable Change Data Capture for audit scenarios

CDC is appropriate when you need a complete history of changes for compliance or debugging. Enable it on specific tables:

```
-- Enable CDC at the database level
EXEC sys.sp_cdc_enable_db;

-- Enable CDC on a specific table
EXEC sys.sp_cdc_enable_table
    @source_schema = 'dbo',
    @source_name = 'Orders',
    @role_name = NULL,
    @capture_instance = 'dbo_Orders',
    @supports_net_changes = 1;
```

Once enabled, SQL Server creates change tables that store historical records. Query these tables to retrieve changes within a time range:

```
DECLARE @from_lsn binary(10), @to_lsn binary(10);
SET @from_lsn = sys.fn_cdc_get_min_lsn('dbo_Orders');
SET @to_lsn = sys.fn_cdc_get_max_lsn();

SELECT *
FROM cdc.fn_cdc_get_all_changes_dbo_Orders(@from_lsn, @to_lsn, 'all');
```

This query returns all changes between the minimum and maximum log sequence numbers, including the operation type (`__$operation`) indicating insert, update, or delete.

Note

CDC requires SQL Server Agent to be running. The capture and cleanup jobs run on schedules you can configure. Monitor job performance in high-transaction environments.

Configure Azure Functions SQL trigger binding

[Azure Functions SQL trigger binding](#) provides the most direct integration for responding to database changes. The trigger monitors tables using Change Tracking and invokes your function when rows change.

First, enable Change Tracking on your database and table:

```
-- Enable Change Tracking at database level
ALTER DATABASE [YourDatabase]
SET CHANGE_TRACKING = ON
(CHANGE_RETENTION = 2 DAYS, AUTO_CLEANUP = ON);

-- Enable Change Tracking on the table
ALTER TABLE dbo.Orders
ENABLE CHANGE_TRACKING;
```

Create an Azure Function with the SQL trigger:

```
[FunctionName("OrderChangedTrigger")]
public static void Run(
    [SqlTrigger("[dbo].[Orders]", "SqlConnectionString")]
    IReadOnlyList<SqlChange<Order>> changes,
    ILogger log)
{
    foreach (var change in changes)
    {
        log.LogInformation($"Change operation: {change.Operation}");
        log.LogInformation($"Order ID: {change.Item.OrderId}");

        if (change.Operation == SqlChangeOperation.Update)
        {
            // Process order update
            ProcessOrderUpdate(change.Item);
        }
    }
}

public class Order
{
    public int OrderId { get; set; }
    public int CustomerId { get; set; }
    public DateTime OrderDate { get; set; }
    public string Status { get; set; }
}
```

The `SqlChange<T>` type wraps each changed row with its operation type. Your function receives batches of changes, which helps efficiency when multiple rows change simultaneously.

Configure the connection string in your function app settings:

```
{
  "Values": {
    "SqlConnectionStrings": "@Microsoft.KeyVault(SecretUri=https://myvault.vault.azure.net/secrets/connection-string/secret-value)"
  }
}
```

Important

The SQL trigger binding requires Change Tracking, not CDC. If you need CDC's historical capabilities alongside real-time triggers, enable both features on your tables.

Stream changes with Change Event Streaming

For high-throughput scenarios where polling latency is unacceptable, Change Event Streaming pushes changes directly to Azure Event Hubs. This approach eliminates polling delays and scales to millions of events per second.

Configure CES on your database:

```
-- Create an event streaming group
CREATE EVENT STREAMING GROUP MyOrderStream
WITH (
  TARGET_TYPE = 'eventhub',
  TARGET_NAME = 'orders-eventhub',
  TARGET_CONNECTION = '<event-hub-connection-string>'
);

-- Add tables to the streaming group
ALTER EVENT STREAMING GROUP MyOrderStream
ADD TABLE dbo.Orders;
```

Changes flow to Event Hubs immediately as transactions commit. Downstream consumers process events using Azure Functions Event Hubs triggers, Stream Analytics, or custom applications.

The Event Hubs message contains:

- Operation type (insert, update, delete)
- Timestamp
- Full row data for inserts and updates
- Key values for deletes

Choose the right change pattern

Selecting the appropriate change mechanism depends on your requirements:

Use **CDC** when:

- You need complete before-and-after values
- Compliance requires historical change records
- Batch synchronization is acceptable

Use **change tracking** with Functions when:

- You need real-time response to changes
- You only need current values, not history
- You're building event-driven microservices

Use **change event streaming** when:

- You need sub-second latency
- High transaction volumes require streaming
- You're building real-time analytics pipelines

Consider combining approaches. Use CDC for audit and compliance while using Functions triggers for real-time notifications. This pattern provides both historical records and immediate responses.

8. Exercise - Configure Data API Builder for a product catalog

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/8-exercise-data-api-builder>

Exercise - Configure Data API Builder for a product catalog

Completed

Now it's your chance to create a Data API Builder configuration and expose a SQL database through modern APIs.

In this exercise, you create a Data API Builder configuration for a product catalog database. You configure entities for tables and views, set up field mappings and relationships, and test REST and GraphQL endpoints locally.

Note

To complete this exercise, you'll need an [Azure subscription](#).

Launch the exercise and follow the instructions.

[Launch Exercise](#)

9. Module assessment

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/9-knowledge-check>

Module assessment

Completed

10. Summary

<https://learn.microsoft.com/en-us/training/modules/integrate-sql-solutions-azure-services/10-summary>

Summary

Completed

In this module, you learned how to:

- Create Data API Builder configuration files with data source connections and runtime settings
- Define entities for tables with field mappings, caching, pagination, and filtering capabilities
- Configure REST and GraphQL endpoints with customized paths and enabled operations
- Expose views as read-only entities and stored procedures for custom operations
- Explore deployment options for Data API Builder, including Azure Container Apps, App Service, and Static Web Apps
- Set up Azure Monitor with Application Insights for request telemetry and database performance tracking
- Implement change capture patterns using CDC, Change Tracking, Azure Functions triggers, and Change Event Streaming

Key takeaways

- Data API Builder eliminates custom API code by generating REST and GraphQL endpoints from a single JSON configuration file
- Entity relationships enable GraphQL clients to traverse connections between tables in single queries, reducing round trips
- Azure Container Apps provides serverless hosting with automatic scaling and managed identity support for production deployments
- Application Insights integration gives visibility into request performance, database queries, and error rates
- Change capture mechanisms enable event-driven architectures that respond to database modifications in real time

Learn more

- [Data API Builder documentation](#)
- [Data API Builder configuration reference](#)
- [Deploy Data API Builder to Azure Container Apps](#)
- [Azure Functions SQL trigger binding](#)
- [Change Data Capture in SQL Server](#)
- [Application Insights overview](#)