

# 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

---

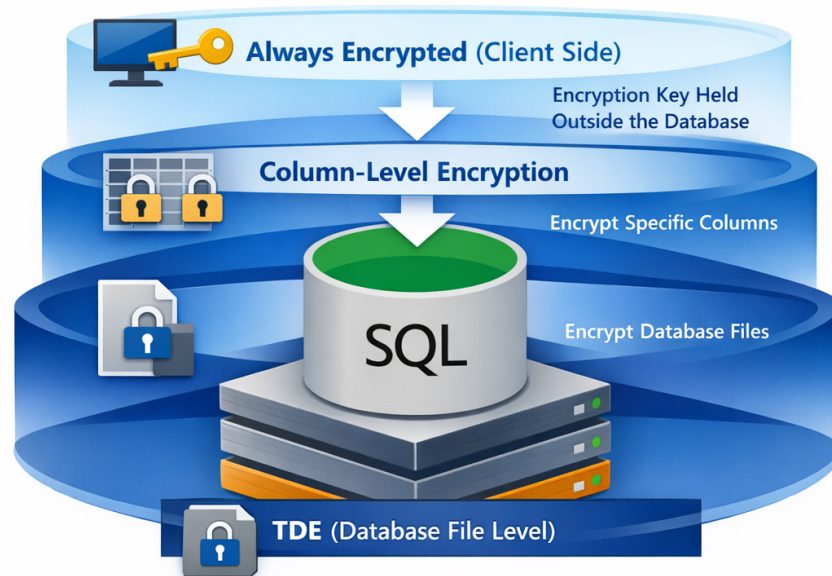
Completed

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.



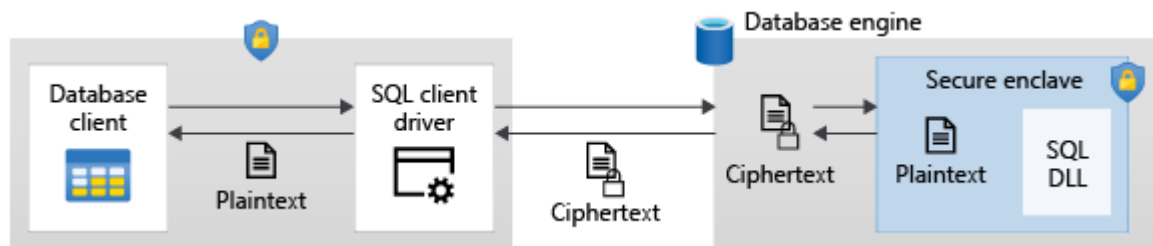
When you enable TDE, SQL Server automatically encrypts database files, transaction logs, and backups. Your applications don't need any code changes—encryption happens transparently behind the scenes. TDE uses a database encryption key protected by a certificate stored in the **master** database.

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

---

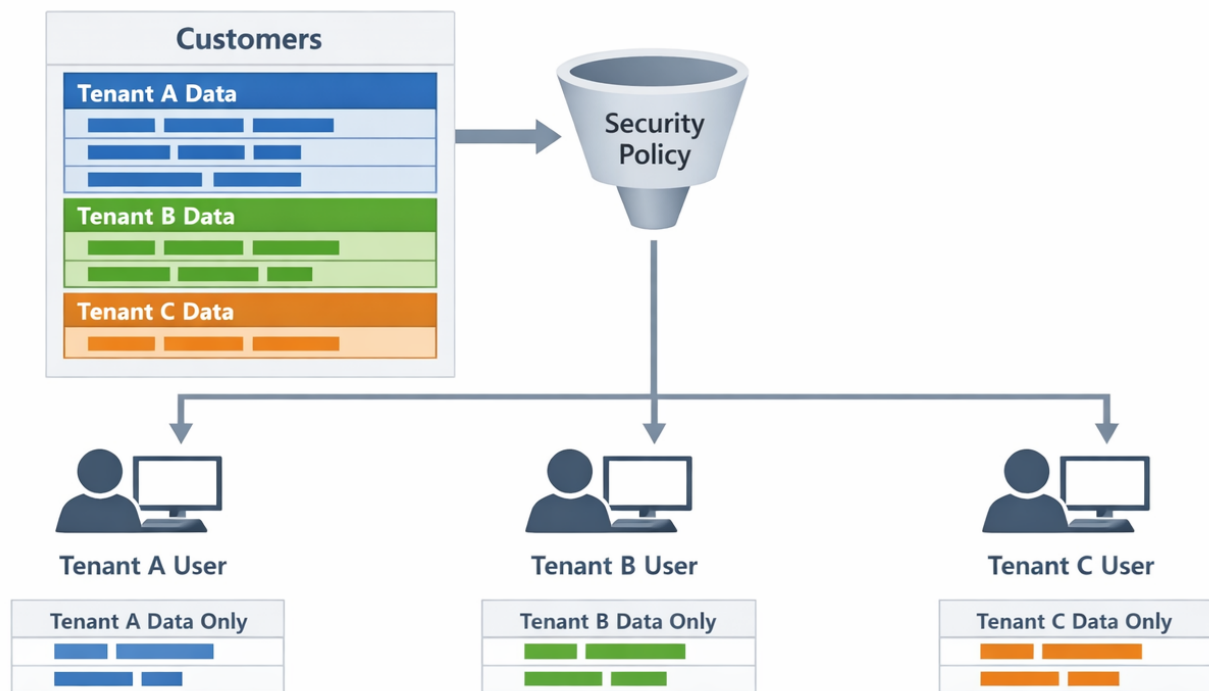
Completed

**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/completions?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.

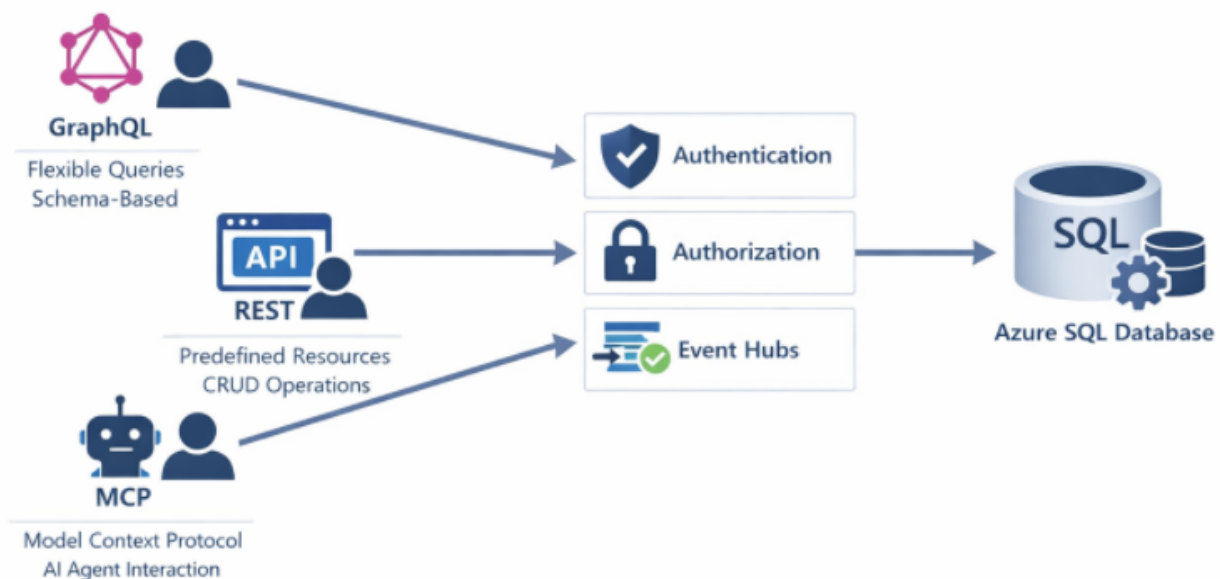
These endpoints often serve as the primary data access layer for web and mobile applications. Without appropriate security controls, they can expose sensitive data, allow unauthorized modifications, or become targets for denial-of-service attacks.

## 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](#)