

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/2008" >
  <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](#)