

Design and implement models and embeddings with SQL

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/design-implement-models-embeddings-with-sql/01-introduction>

Introduction

Completed

Building intelligent applications with Azure SQL Database involves more than storing and querying relational data. Developers need to integrate AI models, generate embeddings from text, and perform vector searches to enable features like semantic search and Retrieval Augmented Generation (RAG). Azure SQL Database and Fabric SQL database support these capabilities through external models, the vector data type, and built-in AI functions.

Together, external models and embedding workflows allow developers to add AI capabilities directly inside the database. This approach keeps AI processing close to the data and reduces the need to move information between systems for tasks like similarity search or content retrieval.

Imagine a retail development team that builds applications using Azure SQL Database to manage product catalogs, customer reviews, and support documentation. Their work includes generating embeddings from product descriptions, enabling semantic search across support articles, and keeping embeddings current as content changes. By using external models and built-in AI functions in Azure SQL Database, the team can design, generate, and maintain embeddings while working entirely within Transact-SQL.

After completing this module, you'll be able to:

- Evaluate AI models for SQL database workloads based on capabilities and performance requirements.
- Create and manage external models to reference AI endpoints from Transact-SQL.
- Design embeddings with appropriate chunking strategies.
- Generate and store embeddings using built-in SQL AI functions.
- Choose maintenance approaches to keep embeddings aligned with source data.

2. Understand and evaluate models for SQL database workloads

<https://learn.microsoft.com/en-us/training/modules/design-implement-models-embeddings-with-sql/02-understand-and-evaluate-models-sql-database-workloads>

Understand and evaluate models for SQL database workloads

Completed

Large language models (LLMs) enable applications to generate responses, summarize information, and reason over user input. Their usefulness increases when they can access application data stored in a database.

Azure SQL Database and Fabric SQL database support building intelligent applications by integrating AI capabilities such as embeddings, vector data types, and vector search. These features allow models to work directly with relational data, enabling common patterns like semantic search and Retrieval Augmented Generation (RAG).

Before integrating a model into a SQL-based solution, it's important to understand how different models behave and how their characteristics affect application design.

Identify model characteristics for SQL database workloads

Models differ in their capabilities, performance characteristics, and supported input and output formats. When evaluating models for use with Azure SQL Database or Fabric SQL database, consider the following factors.

Modalities

Some models process only text, while others support other inputs such as images or structured data. The required modality depends on the type of data stored in the database and the intended application scenario.

Language support

Multilingual support is important when applications serve users across regions or when stored content spans multiple languages.

Model size and capacity

Larger models typically provide stronger reasoning and more nuanced output, but they also consume more tokens and can introduce higher latency and cost. Smaller models might be more suitable for focused tasks such as embedding generation.

Structured output

Models that can produce structured output, such as JSON, are easier to integrate into SQL-based workflows where responses must be processed programmatically.

These characteristics influence whether a model is a good fit for generating embeddings, supporting RAG patterns, or enabling conversational access to database content.

Describe how models interact with Azure SQL data

Azure SQL Database and Fabric SQL database support intelligent application patterns by combining relational storage with AI features such as vector data types and vector functions.

A common pattern is Retrieval Augmented Generation (RAG), where relevant data is retrieved from the database and supplied to a model as context before generating a response. This step allows responses to be grounded in application data rather than relying only on a model's pretrained knowledge.

Several concepts affect how models interact with database data:

- **Tokens**, which are the units models use to process text
- **Embeddings**, which represent data as vectors
- **Vector search**, which compares embeddings to identify semantic similarity

Because vectors live alongside relational data in Azure SQL Database, you can combine vector similarity search with any standard SQL capability in a single query. For example, you might narrow vector search results with a `WHERE` clause, join them to related tables, or blend vector cosine rankings with full-text BM25 scores. This combination of vector search with regular SQL operations is known as **hybrid search**. Instead of sending requests to a separate search service and reconciling the results, you query one database that handles both semantic similarity and relational filtering together.

Understanding these concepts helps you design SQL-based applications that effectively use AI capabilities while managing performance and cost.

Explain how tokens affect cost and design

Models don't process text as raw characters. Instead, they break text into tokens. Tokens are small chunks that might be words, parts of words, or punctuation. For example, the word "hamburger" might become three tokens: "ham", "bur", and "ger", while a common word like "the" is typically a single token.

Token counts matter for two reasons. First, models have input limits. A model might accept a maximum of 8,000 or 128,000 tokens in a single request. This limit constrains how much database content you can include as context in a RAG pattern. Second, model providers typically charge based on tokens processed. More tokens mean higher costs, so efficient text handling directly affects operating expenses.

When you're designing SQL-based AI solutions, token limits influence how you chunk content for embeddings and how much context you can pass to a model during generation.

Because models impose token limits and differ in how embeddings are generated, these characteristics affect schema design, chunking strategies, and query behavior.

Explore models with Microsoft Foundry

Microsoft Foundry Models provides a catalog of AI models that can be used with Azure services. The catalog includes models that support tasks such as text processing, embedding generation, reasoning, and multimodal input.

For SQL database workloads, Foundry helps you evaluate which models are appropriate for integration with Azure SQL Database or Fabric SQL database. Supported input types, language coverage, and deployment options all influence which model fits your scenario.

The model catalog exposes information such as model capabilities, benchmarks, version details, and lifecycle status. This information helps developers understand performance and operational constraints before connecting a model to database workflows.

Using Foundry during design helps ensure that the selected model aligns with SQL-based application requirements and can be integrated predictably with relational data.

Select a model for your solution

Selecting a model is a design decision that affects performance, cost, and maintainability. When choosing a model for use with Azure SQL Database or Fabric SQL database, consider:

- The type and format of data stored in the database
- Performance and scalability requirements
- Language or modality requirements
- Deployment and lifecycle considerations

Understanding these trade-offs helps ensure that AI capabilities are integrated into SQL database solutions in a way that's predictable, scalable, and aligned with application goals.

Key takeaways

Models differ in modalities, language support, size, and structured output capabilities, and these differences affect how they integrate with SQL database workloads. RAG retrieves relevant database content and supplies it to a model as context, while tokens determine how input and output text is processed. Embeddings represent data as vectors that enable semantic similarity comparisons. Microsoft Foundry Models provides a catalog for evaluating and selecting models that align with your SQL-based application requirements.

3. Create and manage external models in SQL

<https://learn.microsoft.com/en-us/training/modules/design-implement-models-embeddings-with-sql/03-create-manage-external-model-sql>

Create and manage external models in SQL

Completed

External models provide a way to reference AI models from within SQL Server and Azure SQL using Transact-SQL. An external model represents a connection to a model endpoint and defines how SQL built-in AI functions invoke that model.

By creating external models, you make AI capabilities available directly inside the database engine without embedding model-specific logic in application code.

What is an external model

An external model is a database object that stores metadata about an AI model endpoint. This metadata includes information such as the endpoint URL, authentication details, and model-specific configuration.

External models aren't models that run inside SQL Server. Instead, they act as a managed reference that allows SQL Server to call an external AI service when executing AI functions.

Once created, external models can be reused across queries and workloads, providing a consistent and centrally managed way to access AI capabilities from SQL.

Create an external model

You create an external model by using the `CREATE EXTERNAL MODEL` statement. This statement defines the model name and associates it with an external endpoint.

Creating the external model doesn't invoke the model or generate output. It establishes a reusable definition that AI functions such as embedding generation or other AI-assisted operations can later reference.

Because external models are database objects, they can be managed using standard SQL practices, including naming conventions, permissions, and deployment through scripts.

Modify an external model

External models can change over time as endpoints, credentials, or configuration settings are updated. To reflect these changes, you can use the `ALTER EXTERNAL MODEL` statement.

Altering an external model updates its metadata without requiring dependent queries or functions to be rewritten. This update allows you to adjust model configuration centrally while keeping existing SQL logic intact.

Managing changes at the external model level helps reduce coupling between application logic and AI service configuration.

Use external models with AI functions

SQL Server provides built-in AI functions that reference an external model to perform AI-related operations. These functions handle tasks such as generating embeddings by calling the external model endpoint defined in the database.

AI functions use the external model definition to determine which endpoint to call and how to authenticate the request. This separation allows Transact-SQL code to focus on data processing rather than connection or credential details.

Before creating an external model, you need a database scoped credential that stores the authentication details for the AI endpoint. Azure OpenAI endpoints support two authentication options:

- **Managed Identity** (recommended for Azure SQL Database): Your database's managed identity authenticates directly with Azure OpenAI. Grant it the Cognitive Services OpenAI User role on the Azure OpenAI resource.

```
CREATE DATABASE SCOPED CREDENTIAL [https://<resource-name>.cognitiveservices.  
    WITH IDENTITY = 'Managed Identity',  
    SECRET = '{"resourceid":"https://cognitiveservices.azure.com"}';
```

- **API key**: Store the key in HTTP headers. This approach works for both Azure SQL Database and SQL Server 2025.

```
CREATE DATABASE SCOPED CREDENTIAL [https://<resource-name>.cognitiveservices.  
    WITH IDENTITY = 'HTTPEndpointHeaders',  
    SECRET = '{"api-key":"<your-api-key>}';
```

Avoid hardcoding API keys in your T-SQL code. Use managed identities when possible for better security and easier key rotation.

Important

Grant the `EXECUTE ANY EXTERNAL ENDPOINT` permission to users or roles that need to call external endpoints: `GRANT EXECUTE ANY EXTERNAL ENDPOINT TO [user_or_role];`

With the credential in place, you can create the external model. The following example shows a minimal workflow.

```
CREATE EXTERNAL MODEL my_external_model
WITH (
    LOCATION      = 'https://<resource-name>.cognitiveservices.azure.com/openai/deploy',
    API_FORMAT    = 'Azure OpenAI',
    MODEL_TYPE    = EMBEDDINGS,
    MODEL         = 'text-embedding-3-small',
    CREDENTIAL    = [https://<resource-name>.cognitiveservices.azure.com/],
    PARAMETERS    = '{"dimensions":<n>}'
);
```

Once the external model is created, AI functions can reference it in queries.

```
SELECT
    id,
    AI_GENERATE_EMBEDDINGS(
        description USE MODEL my_external_model
    ) AS embedding
FROM dbo.documents;
```

In this example, the external model defines how SQL Server connects to the AI service. The `AI_GENERATE_EMBEDDINGS` function uses that definition to generate embeddings for the `description` column without embedding endpoint or authentication details in the query.

When you combine external models with AI functions, SQL Server enables AI workflows that are defined, executed, and maintained entirely through Transact-SQL.

Managing external models as database objects

Because external models are database-scoped objects, they fit naturally into existing database management practices. You can control access through permissions, include them in deployment pipelines, and manage their lifecycle alongside other schema objects.

Treating external models as first-class database objects helps ensure consistency, maintainability, and predictable behavior when integrating AI capabilities into SQL-based solutions.

Key takeaways

External models are database objects that store metadata about AI model endpoints, allowing SQL Server to call external AI services without embedding connection details in application code. You create them with `CREATE EXTERNAL MODEL`, update them with `ALTER EXTERNAL MODEL`, and reference them in built-in AI functions like `AI_GENERATE_EMBEDDINGS`. Because they're database-scoped, you can manage permissions, lifecycle, and deployment alongside your other schema objects.

4. Design embeddings for SQL database workloads

<https://learn.microsoft.com/en-us/training/modules/design-implement-models-embeddings-with-sql/04-design-embeddings-sql-database-workloads>

Design embeddings for SQL database workloads

Completed

Embeddings represent data as vectors so that similarity between pieces of text can be compared. How you design embeddings affects relevance, performance, and cost when vectors are later generated and queried.

SQL Server provides built-in AI functions that support embedding workflows. Common vector search patterns help guide how text should be prepared before embeddings are generated.

Understand how vectors are created

But what does the term **vector** mean in the context of AI integration with a SQL database?

An AI model creates a vector, not SQL itself. The model was trained to read text and return a list of numbers that represents the meaning of that text.

When SQL sends text to an embedding model, the model returns a vector, which SQL stores for later comparison with other vectors.

For example, when the text "*lightweight hiking backpack*" is sent to an embedding model, the model might return:

```
[0.12, -0.87, 0.45, 0.31, ...]
```

A related sentence such as "*compact backpack for day hikes*" would produce a different list of numbers that looks similar:

```
[0.10, -0.85, 0.47, 0.29, ...]
```

Because the vectors look similar, SQL can treat the two pieces of text as related, even though the wording is different.

Identify data to include in embeddings

Embeddings work best when they represent text that carries semantic meaning, such as descriptions, titles, or other free-form text fields.

Columns that store identifiers, numeric values, or operational metadata usually don't add semantic value and should be excluded. Limiting embeddings to meaningful text reduces token usage and improves similarity results.

Design choices at this stage determine what information embeddings capture and how well they represent the underlying data.

Control input size and structure

Embedding models operate on tokenized input and impose limits on how much text can be processed in a single request. A token is a small piece of text, such as a word or part of a word, that the model processes as a unit. Long text values often require division into smaller units.

SQL Server supports this pattern through built-in AI functions that help prepare text for embedding workflows. By controlling input size, you can keep text within model limits and ensure each embedding represents a clear, focused piece of content.

Well structured input also helps avoid embedding unrelated ideas together, which can reduce the quality of similarity results.

Design chunking strategies

Chunking defines how larger text values are divided into smaller segments. A chunking strategy balances context and precision.

Chunks that are too large may exceed token limits or dilute semantic focus. Chunks that are too small may lose important context. The goal is to preserve meaning while keeping chunks efficient to process.

In practice, chunking is defined by the rules you apply when splitting text in SQL. These rules typically control how much text goes into each chunk, such as a maximum number of characters, and where splits are allowed to occur.

```
AI_GENERATE_CHUNKS(SOURCE = description, CHUNK_TYPE = FIXED, CHUNK_SIZE = 500)
```

By defining chunking rules directly in SQL, near the source tables, you can change chunk size or splitting behavior by adjusting the query instead of modifying application code.

Apply embedding design with SQL

The following example shows a conceptual pattern that prepares text for embedding generation by splitting it into smaller units.

```
SELECT
    id,
    c.chunk
FROM dbo.documents
CROSS APPLY
    AI_GENERATE_CHUNKS(SOURCE = description, CHUNK_TYPE = FIXED, CHUNK_SIZE = 500)
```

In this example, the text in the `description` column is divided into chunks of up to 500 characters. Each chunk can later be passed to an embedding function, ensuring embeddings represent focused portions of the original content.

For example, if a row contains the following text in the `description` column:

```
"Lightweight backpack designed for long day hikes in warm weather."
```

The chunking function might produce multiple chunks such as:

- "Lightweight backpack designed for long day hikes"
- "in warm weather."

Each chunk is returned as a separate row by the query. These smaller pieces can then be passed individually to an embedding function so that each embedding represents a focused portion of the original text.

Tip

One design pattern is to store vectors in a separate table from the source text data. A dedicated embeddings table makes it easier to track how much space vectors consume and to rebuild embeddings without affecting the original data.

Key takeaways

Embedding quality depends on the decisions you make before any model runs. Choosing the right columns and chunking long text into focused segments with `AI_GENERATE_CHUNKS` determine how useful your vector search results are. Getting these design choices right early prevents costly rework when you move to embedding generation and storage.

5. Generate and maintain embeddings for SQL database workloads

Generate and maintain embeddings for SQL database workloads

Completed

After you design how embeddings represent your data, you need to generate them and keep them in sync as data changes.

SQL Server provides built-in AI functions to generate embeddings from text stored in database columns. Generating embeddings is typically not a one-time operation. When source data changes, embeddings may need to be regenerated so they continue to reflect the current state of the data.

Generate embeddings with SQL

SQL Server provides the `AI_GENERATE_EMBEDDINGS` function to generate embeddings directly from text stored in database columns. This function uses an external model to convert text into a vector that can be stored and later compared.

A common pattern is to generate embeddings during an initial load or as part of a batch process. The resulting vectors are stored alongside the source data or in a related table so they can be queried efficiently.

The following example shows a simple end-to-end pattern, from table definition to embedding generation.

First, create a table that stores both the source text and its embedding.

```
CREATE TABLE dbo.documents
(
    id INT PRIMARY KEY,
    description NVARCHAR(MAX),
    embedding VECTOR(1536)
);
```

Next, generate embeddings from the text and store them in the table.

```
UPDATE dbo.documents
SET embedding = AI_GENERATE_EMBEDDINGS(description USE MODEL my_embedding_model);
```

In this example, each row's `description` value is sent to the embedding model. The function returns a vector, which is stored in the `embedding` column. These stored vectors can later be queried or compared without regenerating them. You might want to include extra logic to handle chunking or filtering based on your embedding design.

Embedding generation determines how vectors are created. Maintenance strategies determine when those vectors need to be refreshed.

Understand embedding maintenance

Embedding maintenance keeps stored embeddings aligned with changes to the underlying data. When text values are inserted, updated, or deleted, the corresponding embeddings may no longer reflect the current content.

Different maintenance approaches can be used depending on how often data changes, how quickly embeddings must be updated, and where embedding generation runs.

Choose an embedding maintenance method

Embeddings need to stay aligned with the source text as data changes. Several options can be used to detect changes and decide when embeddings should be regenerated. These options differ in where the work happens and how quickly changes are reflected.

- **Table triggers**
Triggers run automatically when rows are inserted or updated. For embedding maintenance, a trigger can mark rows that need new embeddings or initiate regeneration immediately. This approach reflects changes quickly but adds work to write operations.
- **Change Tracking**
Change Tracking records that a row changed since a given point in time. A background process can use this information to identify which rows need their embeddings refreshed and process them in batches. This approach balances latency and performance.
- **Change Data Capture (CDC)**
CDC records detailed information about data changes, including before and after values. Embedding maintenance can use CDC tables to identify which text values changed and regenerate embeddings asynchronously. This approach is suitable for high-volume workloads.
- **Azure Functions with SQL trigger binding**
Azure Functions can react to database changes using SQL trigger bindings. This feature allows embedding generation to run outside the database engine while still responding to data changes. This approach offloads work from the database and can scale independently.
- **Azure Logic Apps**
Logic Apps can orchestrate embedding updates as part of a workflow. For example, a Logic App might periodically check for changed rows and call an embedding service, coordinating

updates with other systems. This approach is low-code and integrates well with other Azure services.

- **Change Event Streaming**

Change event streaming (CES) streams DML changes directly into Azure Event Hubs in near real-time. Downstream systems can consume these events and regenerate embeddings as changes occur without adding work to database transactions. This approach decouples embedding generation from the database and supports multiple consumers processing the same change stream.

- **Microsoft Foundry**

Microsoft Foundry can be used to manage and evaluate the models that generate embeddings. In a maintenance workflow, Foundry typically supports model selection or hosting while another process handles change detection and database updates. This approach centralizes model management while embedding generation occurs in response to data changes.

Choose an appropriate maintenance approach

There's no single correct way to maintain embeddings. The right approach depends on factors such as data volume, update frequency, latency requirements, and where embedding generation fits within the overall solution.

Some solutions favor immediate updates, while others prioritize batching or external processing. Understanding these trade-offs helps you choose a maintenance strategy that fits your SQL application.

Key takeaways

Generating embeddings with `AI_GENERATE_EMBEDDINGS` is only the first step. As source data changes, stored vectors can fall out of sync, so you need a maintenance strategy. Options range from triggers and Change Tracking for tightly coupled updates to Change Data Capture, Change Event Streaming, and Azure Functions for asynchronous or decoupled approaches. The right choice depends on your data volume, latency requirements, and where embedding generation fits in your overall architecture.

6. Exercise - Generate and update embeddings in Azure SQL Database

<https://learn.microsoft.com/en-us/training/modules/design-implement-models-embeddings-with-sql/06-exercise-generate-update-embeddings-sql>

Exercise - Generate and update embeddings in Azure SQL Database

Completed

In this exercise, you create an external model, generate embeddings from text stored in an Azure SQL Database using built-in AI functions, and perform a vector search to find semantically similar content.

Note

To complete this exercise, you need an [Azure subscription](#) and be approved for Azure OpenAI access. If you need Azure OpenAI access, apply at the [Azure OpenAI limited access](#) page.

Launch the exercise and follow the instructions.

[Launch Exercise](#)

7. Knowledge check

<https://learn.microsoft.com/en-us/training/modules/design-implement-models-embeddings-with-sql/07-knowledge-check>

Knowledge check

Completed

8. Summary

Summary

Completed

This module covered designing and implementing models and embeddings for Azure SQL Database and Fabric SQL database. You learned how to evaluate AI models, create external models to reference AI endpoints from Transact-SQL, and design effective chunking strategies. You also explored approaches for generating and maintaining embeddings as data changes.

After completing this module, you can:

- Evaluate AI models for SQL database workloads based on capabilities and performance requirements.
- Create and manage external models to reference AI endpoints from Transact-SQL.
- Design embeddings with appropriate chunking strategies.
- Generate and store embeddings using built-in SQL AI functions.
- Choose maintenance approaches to keep embeddings aligned with source data.

Additional reading

- [Generative AI with Azure SQL Database](#)
- [Vector data type](#)
- [Azure AI Foundry model catalog](#)
- [CREATE EXTERNAL MODEL \(Transact-SQL\)](#)