

Domain 3: Implement AI capabilities in database solutions

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 \(Transact-SQL\)](#)

Design and implement intelligent search with SQL

1. Introduction

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

Introduction

Completed

Finding relevant information in a database often requires more than matching keywords. Users search for concepts, not just words. A customer looking for "lightweight hiking backpack" might find nothing if the product is listed as "ultralight trail bag." SQL Server and Azure SQL Database provide multiple search approaches to handle these challenges: full-text search for keyword matching, vector search for semantic similarity, and hybrid search that combines both.

Full-text search excels when users know the specific terms they want. Vector search uses embeddings to find content based on meaning even when the exact words differ. Hybrid search runs both approaches and merges the results, giving users the best of both worlds. Choosing the right approach depends on how your users search and what trade-offs you can accept between precision, recall, and performance.

Imagine a retail team building a product search feature. They have embeddings stored for their product descriptions and support articles. Now they need search capabilities that help customers find products and answers quickly. Some customers type exact product names. Others describe what they need in natural language. The team decides to implement hybrid search so their application handles both patterns effectively, using full-text indexes for keyword matching and vector search for semantic similarity.

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

- To select the right approach for different query types, evaluate full-text search, vector search, and hybrid search.
- Implement full-text search using full-text indexes and query predicates.
- Prepare SQL databases for vector search by storing embeddings and choosing distance metrics.

- Write vector search queries using VECTOR_DISTANCE and VECTOR_SEARCH functions.
- Combine full-text and vector search results using Reciprocal Rank Fusion (RRF).

2. Choose an intelligent search approach

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/02-choose-intelligent-search-approach>

Choose an intelligent search approach

Completed

SQL Server and Azure SQL Database provide you with three ways to search text data, full-text search, vector search, and hybrid search. Each works differently, and choosing the right one depends on what your users need to find.

To see why the right search approach matters, imagine a customer searching your product catalog. They type "lightweight hiking backpack" but your best-selling item is listed as "ultralight trail bag." A traditional keyword search finds nothing. A smarter search option understands that both descriptions mean the same thing. The search approach you choose determines whether your users find what they need or not.

Understand the three search approaches

Let's look at how each approach works and when you'd use it.

Full-text search looks for words and phrases. When a customer searches for "mountain bike helmet," full-text search finds products containing those exact terms. It understands language rules, so searching for "ride" can also find "riding" or "rode." Use full-text search when users know the specific words they want.

Semantic vector search looks for meaning. Instead of matching words, it compares the mathematical representation of concepts. When a customer searches for "something to keep me visible on evening rides," vector search can find products described as "reflective cycling vest" or "LED bike lights" even though those words don't appear in the search. This approach works well when users describe what they need rather than name it.

Hybrid search uses both. It runs a full-text search and a vector search, then combines the results. A customer searching for "comfortable seat for long tours" gets products matching those keywords and products like "touring saddle" or "gel bike seat" that vector search identifies as semantically related.

Match the approach to the query intent

Your users express different intentions when they search. Some know exactly what they want. Others are exploring.

Consider these two searches against a product database:

- "Model XR-500" - The user wants that specific product. Full-text search handles this search perfectly.
- "something to keep drinks cold on a hike" - The user describes a need. Vector search understands this means "insulated water bottle" or "portable cooler."

When you can't predict how users search, hybrid search covers both cases. The user who types "XR-500 portable cooler" gets results matching the model number and results matching the concept.

Evaluate the trade-offs

Choosing a search approach involves trade-offs. Consider these factors as you design your solution.

Precision versus recall - Full-text search returns fewer results, but they closely match the query terms. Vector search returns more results, including items that are conceptually related but use different words. If your application needs exact matches, lean toward full-text. If it needs discovery, lean toward vector search.

Data requirements - Full-text search needs a full-text index on your text columns. Vector search needs embeddings stored in vector columns. You might need to prepare your data differently depending on which approach you choose.

Performance characteristics - Full-text indexes are optimized for fast keyword lookup. Vector search performance depends on how many vectors you're comparing and whether you use approximate or exact methods. Hybrid search runs both, so it takes longer than either one alone.

These trade-offs aren't about which approach is better. The right choice depends on which approach fits your scenario.

Combine approaches with hybrid search

Full-text or vector search alone doesn't handle every situation. Full-text search misses documents that express the same idea with different words. Vector search might return conceptually related items that don't contain important terms the user specified.

Hybrid search solves this issue by running both approaches and merging the results. A user searching for "Azure SQL backup strategies" gets documents containing those keywords and documents about "database recovery planning" that vector search identifies as related.

The challenge is combining two ranked lists into one. How do you decide which result should appear first when full-text search ranks it #3 and vector search ranks it #7?

Merge results with Reciprocal Rank Fusion

Reciprocal Rank Fusion (RRF) is an algorithm for combining ranked results from different searches. Instead of comparing raw scores, which use different scales, RRF calculates a score based on each document's position (the **rank**) using the formula $1/(\text{rank} + k)$. The constant **k** is 60, a value established by the original RRF research and used by Azure AI Search. Documents appearing in multiple result sets have their scores summed, which is where RRF really shines.

Here's how it works. Suppose a customer searches for "comfortable bike seat" and you run both full-text and vector search:

Rank	Full-text results	Vector search results
1	Bike Seat Comfort Pro	Ergonomic Touring Saddle
2	Comfortable Handlebar Grips	Gel Cushion Saddle
3	Racing Bike Seat	Bike Seat Comfort Pro

Full-text search finds products containing the words *comfortable*, *bike*, and *seat*. Vector search finds semantically similar products, including saddles that don't use the word *seat* at all.

Notice that "Bike Seat Comfort Pro" appears in both lists: rank 1 in full-text and rank 3 in vector search. RRF calculates its combined score by summing:

- Full-text rank 1: $1/(60+1) = 0.0164$
- Vector rank 3: $1/(60+3) = 0.0159$
- **Combined: 0.0323**

Compare that to "Ergonomic Touring Saddle," which only appears in vector search at rank 1:

- Vector rank 1: $1/(60+1) = 0.0164$
- **Combined: 0.0164**

Even though "Ergonomic Touring Saddle" ranked #1 in vector search, "Bike Seat Comfort Pro" wins the combined ranking because appearing in both lists doubled its score. The merged results:

Rank	Combined results	RRF Score
1	Bike Seat Comfort Pro	0.0323
2	Ergonomic Touring Saddle	0.0164
3	Comfortable Handlebar Grips	0.0161
4	Gel Cushion Saddle	0.0161
5	Racing Bike Seat	0.0159

Notice that ranks 3 and 4 have identical scores. When RRF produces ties, the ordering between those results depends on the implementation and might be arbitrary or alphabetical. In practice, tied

results are equally relevant, so the exact order between them matters less than ensuring they all appear above results with lower scores.

RRF rewards documents that perform well across multiple search methods, surfacing results that match both keywords and meaning.

Key takeaways

Full-text search, vector search, and hybrid search each serve different purposes. Full-text search excels at finding exact keywords and phrases, vector search captures semantic meaning across different phrasings, and hybrid search combines both to maximize relevance. Your choice depends on how your users search and what trade-offs you can accept between precision, recall, and performance. When you implement hybrid search, Reciprocal Rank Fusion (RRF) is a powerful technique for merging ranked results from full-text and vector searches, ensuring that items relevant to both methods rise to the top of the results list.

3. Implement full-text search

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/03-implement-full-text-search>

Implement full-text search

Completed

Full-text search lets you run linguistic searches against text data in SQL Server and Azure SQL Database. Unlike the `LIKE` operator, which only matches character patterns, full-text search understands words, phrases, and language rules.

When a customer searches your product catalog for "ride," they expect to find results that say "riding," "rides," or "rode." Full-text search handles these scenarios by working with language-aware indexes that understand word forms. It doesn't automatically match synonyms or abbreviations (like "MTB" for "mountain bike") unless you configure a thesaurus, but it handles inflections out of the box.

When to use full-text search

Full-text search works best when your users search using specific words or phrases. Consider these scenarios where full-text search is the right choice:

- A customer types a product name or part number

- A support agent searches for tickets containing specific error messages
- A user looks for documents with exact technical terms

If users instead describe what they need without using specific terms, vector search might be more appropriate. But when users know the words they want, full-text search delivers fast, precise results.

Understand full-text indexes and predicates

Full-text search requires two things: a full-text index on your text columns, and predicates to query that index.

A full-text index stores information about words and their locations within your text columns. SQL Server breaks down the text into individual tokens, removes common words like "the" and "is" (called stopwords), and builds an inverted index that maps words to the rows containing them. This structure allows fast lookups without scanning every row.

To query a full-text index, you use predicates in your `WHERE` clause. SQL Server provides two main predicates:

- **CONTAINS** searches for exact matches of words and phrases. Use it when you need precise control over what matches.
- **FREETEXT** searches for meaning rather than exact words. It automatically looks for inflectional forms of your search terms.

SQL Server also provides table-valued functions **CONTAINSTABLE** and **FREETEXTTABLE** that return ranked results with relevance scores, which is useful when you want to sort results by how well they match.

Query a full-text index

The following example shows how to search a product catalog using both `CONTAINS` and `FREETEXT`. Assume the `Production.Product` table has a full-text index on the `Name` column.

Use `CONTAINS` when you need exact word matching:

```
SELECT ProductID, Name, ListPrice
FROM Production.Product
WHERE CONTAINS(Name, 'mountain')
      AND ListPrice < 500;
```

This query returns products with "mountain" in the name under \$500. The `CONTAINS` predicate checks the full-text index while the `ListPrice` filter uses the regular column.

Use `FREETEXT` when you want SQL Server to find related word forms automatically:

```
SELECT ProductID, Name
FROM Production.Product
```

```
WHERE FREETEXT(Name, 'riding bikes');
```

This query finds products related to "riding bikes," including matches for "ride," "rides," and "biking." `FREETEXT` handles inflections automatically without requiring you to specify them.

When you need ranked results, use `CONTAINSTABLE` to get full-text relevance scores. Full-text search assigns a rank to each result based on how well it matches the query. This rank is useful for sorting results by relevance:

```
SELECT p.ProductID, p.Name, ft.RANK
FROM Production.Product AS p
INNER JOIN CONTAINSTABLE(Production.Product, Name,
    'NEAR((mountain, bike))') AS ft
    ON p.ProductID = ft.[KEY]
ORDER BY ft.RANK DESC;
```

This query finds products where "mountain" and "bike" appear near each other, sorted by full-text relevance. The `RANK` column indicates how well each row matches the search criteria. Later, when you combine full-text search with vector search, you use these full-text ranks alongside vector similarity scores to produce merged results.

Common query patterns

Full-text search supports several query patterns. Each addresses a different search need.

Term search finds rows containing a specific word. A search for `CONTAINS(Description, 'aluminum')` returns all products with "aluminum" in the description.

Phrase search finds rows containing words in a specific order. Wrapping the search in quotes like `CONTAINS(Description, '"mountain bike"')` matches only that exact phrase, not rows where "mountain" and "bike" appear separately.

Prefix search finds words that start with specific characters. Searching for `CONTAINS(Description, '"light*")` matches "light," "lights," "lighter," "lightweight," and any other word beginning with "light."

Inflectional search finds different forms of a word. Using `CONTAINS(Description, 'FORMSOF(INFLECTIONAL, "ride")')` matches "ride," "rides," "riding," and "rode."

Proximity search finds words that appear near each other. Searching for `CONTAINS(Description, 'NEAR((light, aluminum))')` finds products where "light" and "aluminum" appear close together, which often indicates a lightweight aluminum product.

Evaluate full-text search results

After implementing full-text search, evaluate whether it meets your users' needs. Watch for these signals:

Precision problems occur when results include irrelevant items. If searching for "brake" returns results about "brake-resistant" coatings when users want bicycle brakes, your search is too broad. Consider using phrase searches or more specific terms.

Noise happens when common words dilute results. Full-text search uses stoplists to filter out words like "the" and "and," but domain-specific noise might require custom stoplists. If searching for "the mountain" returns too many results because "mountain" appears everywhere, you might need to adjust your approach.

Query intent mismatch appears when users search for concepts rather than words. If customers search for "something for rainy commutes" and expect to find waterproof gear, full-text search doesn't help because it matches words, not meaning. This signal suggests you might need vector search or hybrid search instead.

Key takeaways

Full-text search excels when users search with specific words and phrases. It uses full-text indexes to enable fast linguistic searches and provides predicates like `CONTAINS` and `FREETEXT` to query those indexes. Different query patterns (term, phrase, prefix, inflectional, and proximity) address different search needs. When full-text search returns too many irrelevant results, or when users search for concepts instead of specific words, the next search method to consider is vector search.

4. Prepare SQL for vector search

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/04-prepare-sql-vector-search>

Prepare SQL for vector search

Completed

Before you can run vector searches, you need to store vectors in your database and decide how those searches execute. This unit covers the design decisions you make before writing your first vector query.

Vector search finds rows based on mathematical similarity rather than exact matches. To make this work, you store embeddings as vectors, choose how to measure similarity, and decide whether to search all vectors exactly or use an index for faster approximate results.

Store vectors with the vector data type

SQL Server and Azure SQL Database provide a native **vector** data type designed for storing embeddings. Each vector is an array of floating-point numbers, stored in an optimized binary format but exposed as JSON arrays for convenience.

When you define a vector column, you specify the number of dimensions:

```
CREATE TABLE dbo.Products
(
    ProductID INT PRIMARY KEY,
    Name NVARCHAR(100),
    Description NVARCHAR(MAX),
    DescriptionVector VECTOR(1536) NOT NULL
);
```

The number in parentheses matches the dimensions your embedding model produces. For example, OpenAI's text-embedding-3-small model produces 1,536 dimensions, so your column would be `VECTOR(1536)`. The maximum supported is 1,998 dimensions.

Each element is stored as a single-precision (4-byte) float. A 1,536-dimension vector uses about 6 KB per row. When designing your table, consider how this column affects storage and memory as your data grows.

Understand distance metrics

Vector search works by calculating the distance between vectors. Vectors that are closer together represent more similar concepts. SQL Server supports three distance metrics:

Cosine distance measures the angle between vectors, ignoring their magnitude. Two vectors pointing in the same direction have a cosine distance of 0, regardless of their length. This metric works well when you care about the direction of meaning, not the intensity. Cosine distance ranges from 0 (identical) to 2 (opposite).

Euclidean distance measures the straight-line distance between two points in vector space. It considers both direction and magnitude. Euclidean distance ranges from 0 (identical) to infinity.

Dot product calculates the sum of element-wise products. SQL Server returns the negative dot product so that smaller values indicate more similar vectors, consistent with the other metrics.

Most embedding models are optimized for cosine similarity, making cosine distance the common choice. However, the best metric depends on how your embeddings were trained.

Choose between exact and approximate search

When you search for similar vectors, you have two options: exact nearest neighbor (ENN) search and approximate nearest neighbor (ANN) search.

Exact nearest neighbor search compares your query vector against every vector in the table. It guarantees the most accurate results but requires calculating the distance to every row. Use the `VECTOR_DISTANCE` function for exact search:

```
DECLARE @query VECTOR(1536) = '[0.1, 0.2, ...]';

SELECT TOP 10 ProductID, Name,
       VECTOR_DISTANCE('cosine', @query, DescriptionVector) AS Distance
FROM dbo.Products
ORDER BY Distance;
```

This query calculates the cosine distance between `@query` and every row's `DescriptionVector`, then returns the 10 closest matches. For small tables (under 50,000 vectors as a general guideline), exact search performs well.

Approximate nearest neighbor search uses a vector index to find similar vectors without scanning every row. It trades perfect accuracy for speed, returning results that are very close to exact but not guaranteed to be the absolute nearest neighbors.

Create a vector index using `CREATE VECTOR INDEX`:

```
CREATE VECTOR INDEX idx_Products_DescriptionVector
ON dbo.Products(DescriptionVector)
WITH (METRIC = 'cosine', TYPE = 'DiskANN');
```

SQL Server uses the DiskANN algorithm, which builds a graph structure that allows fast navigation to nearby vectors. Once the index exists, use the `VECTOR_SEARCH` function for approximate search:

```
DECLARE @query VECTOR(1536) = '[0.1, 0.2, ...]';

SELECT t.ProductID, t.Name, s.distance
FROM VECTOR_SEARCH(
    TABLE = dbo.Products AS t,
    COLUMN = DescriptionVector,
    SIMILAR_TO = @query,
    METRIC = 'cosine',
    TOP_N = 10
) AS s
ORDER BY s.distance;
```

Decide when to use each approach

The choice between exact and approximate search depends on your dataset size and accuracy requirements.

Use exact search (`VECTOR_DISTANCE`) when:

- Your table has fewer than 50,000 vectors

- Your query filters reduce the candidate set to a few rows
- You need guaranteed accurate results and can accept longer query times

Use approximate search (VECTOR_SEARCH with an index) when:

- Your table has hundreds of thousands or millions of vectors
- Query speed matters more than perfect accuracy
- A recall close to 1 (meaning most true nearest neighbors are found) is acceptable

Recall measures how many of the true nearest neighbors the approximate search returns compared to an exact search. DiskANN typically achieves high recall, meaning the results are very close to what exact search would return.

Consider index limitations

Vector indexes in SQL Server have specific requirements to be aware of:

- The table must have a single-column integer primary key with a clustered index
- Tables with vector indexes become read-only while the index exists—you must drop the index to modify data, then recreate it
- Vector indexes can't be partitioned

Note

Vector indexes are currently in preview. In Azure SQL Database and SQL database in Microsoft Fabric, you can set the `ALLOW_STALE_VECTOR_INDEX` database scoped configuration to `ON` to allow writes, but the index doesn't reflect new data until you rebuild it. This option isn't currently available in SQL Server 2025. Check the current documentation for the latest on these limitations.

These limitations affect how you design your tables and maintenance workflows. For tables that change frequently, you might use exact search until the data stabilizes, then add a vector index.

Key takeaways

Preparing SQL for vector search means making three decisions: what data type and dimensions to use, which distance metric matches your embedding model, and whether exact or approximate search fits your dataset size. Exact search with `VECTOR_DISTANCE` works well for smaller datasets or filtered queries, while approximate search with `VECTOR_SEARCH` and a DiskANN index handles larger datasets efficiently. In the next unit, you'll learn the specific functions and query patterns for running vector searches.

5. Implement vector search query patterns

Implement vector search query patterns

Completed

With vectors stored and a search strategy chosen, you're ready to write queries. This unit covers the T-SQL functions that power vector search and shows you how to combine them effectively.

Vector search queries follow a common pattern: generate or retrieve a query vector, calculate distances to stored vectors, and return the closest matches. SQL Server provides four functions that work together to support this pattern: `VECTOR_DISTANCE`, `VECTOR_SEARCH`, `VECTOR_NORMIMIZE`, and `VECTORPROPERTY`.

Calculate distances with `VECTOR_DISTANCE`

The `VECTOR_DISTANCE` function calculates the distance between two vectors using the metric you specify. This function performs exact nearest neighbor search, computing the distance to every qualifying row. In practical terms, it compares your search embedding against every stored embedding to find which ones are most similar, like checking every product in a catalog to find the best matches.

```
VECTOR_DISTANCE(metric, vector1, vector2)
```

The metric must be `'cosine'`, `'euclidean'`, or `'dot'`. Both vectors must have the same number of dimensions.

Here's a typical pattern for finding similar products based on a description embedding:

```
-- Generate an embedding for the user's search phrase
DECLARE @searchVector VECTOR(1536);
SELECT @searchVector = AI_GENERATE_EMBEDDINGS('lightweight hiking boots' USE MODEL

SELECT TOP 10
    ProductID,
    Name,
    VECTOR_DISTANCE('cosine', @searchVector, DescriptionVector) AS Distance
FROM dbo.Products
ORDER BY Distance;
```

This query works in two steps. First, it converts the search phrase "lightweight hiking boots" into an embedding using the same model that created the stored `DescriptionVector` values. Second, it calculates the cosine distance between `@searchVector` and every row's `DescriptionVector`. The

`ORDER BY Distance` clause sorts the results from smallest distance (most similar) to largest, and `TOP 10` returns only the first 10 rows. Since smaller distances mean more similar vectors, this combination gives you the 10 products most semantically related to "lightweight hiking boots."

You can also use `VECTOR_DISTANCE` in a `WHERE` clause to find all vectors within a specific distance threshold:

```
SELECT ProductID, Name
FROM dbo.Products
WHERE VECTOR_DISTANCE('cosine', @searchVector, DescriptionVector) < 0.3
ORDER BY VECTOR_DISTANCE('cosine', @searchVector, DescriptionVector);
```

This approach finds all products within a similarity threshold rather than returning a fixed count. The 0.3 value here's just an example. Cosine distance ranges from 0 (identical vectors) to 2 (completely opposite), but what counts as "similar enough" depends entirely on your data and embedding model. To find the right threshold, run test queries, examine the distance values in your results, and identify where relevant results stop and the noise begins. Use this pattern when you want everything meeting a quality bar rather than a fixed number of results.

Use `VECTOR_SEARCH` for approximate retrieval

When your table grows beyond tens of thousands of rows, calculating distances to every row becomes slow. The `VECTOR_SEARCH` function uses a vector index to find approximate nearest neighbors quickly.

```
VECTOR_SEARCH(
    TABLE = object AS alias,
    COLUMN = vector_column,
    SIMILAR_TO = query_vector,
    METRIC = 'cosine' | 'euclidean' | 'dot',
    TOP_N = k
)
```

This function returns a result set containing all columns from the source table plus a `distance` column. Here's how you'd use it:

```
DECLARE @searchVector VECTOR(1536);
SELECT @searchVector = AI_GENERATE_EMBEDDINGS('lightweight hiking boots' USE MODEL

SELECT t.ProductID, t.Name, s.distance
FROM VECTOR_SEARCH(
    TABLE = dbo.Products AS t,
    COLUMN = DescriptionVector,
    SIMILAR_TO = @searchVector,
    METRIC = 'cosine',
    TOP_N = 10
) AS s
ORDER BY s.distance;
```

The function returns the approximate 10 nearest neighbors. If a matching vector index exists on `DescriptionVector` with the same metric, `VECTOR_SEARCH` uses it for fast retrieval. Without an index, it falls back to exact search and raises a warning.

Handle post-filtering carefully

`VECTOR_SEARCH` applies any WHERE clause conditions *after* finding the nearest neighbors, not before. Consider this query:

```
SELECT t.ProductID, t.Name, s.distance
FROM VECTOR_SEARCH(
    TABLE = dbo.Products AS t,
    COLUMN = DescriptionVector,
    SIMILAR_TO = @searchVector,
    METRIC = 'cosine',
    TOP_N = 10
) AS s
WHERE t.CategoryID = 5
ORDER BY s.distance;
```

This query first finds the 10 nearest vectors across all categories, then filters to only category 5. If none of the 10 nearest products are in category 5, you get no results. To work around this issue, request more candidates than you need:

```
SELECT TOP 10 t.ProductID, t.Name, s.distance
FROM VECTOR_SEARCH(
    TABLE = dbo.Products AS t,
    COLUMN = DescriptionVector,
    SIMILAR_TO = @searchVector,
    METRIC = 'cosine',
    TOP_N = 50
) AS s
WHERE t.CategoryID = 5
ORDER BY s.distance;
```

By requesting 50 candidates from `VECTOR_SEARCH` and then taking the top 10 after filtering, you're more likely to have enough results in the target category.

Normalize vectors for consistent comparisons

Different embedding models produce vectors with different lengths (magnitudes). When comparing vectors from multiple sources or when your model doesn't normalize outputs, you can use `VECTOR_NORMIMIZE` to scale vectors to unit length.

```
VECTOR_NORMIMIZE(vector, norm_type)
```

The `norm_type` can be:

- `'norm2'` : Euclidean norm (most common)
- `'norm1'` : Sum of absolute values
- `'norminf'` : Maximum absolute value

```
DECLARE @v VECTOR(3) = '[3, 4, 0]';
SELECT VECTOR_NORMALIZE(@v, 'norm2') AS NormalizedVector;
-- Returns: [0.6, 0.8, 0.0]
```

Normalizing ensures that cosine distance and dot product behave consistently. Most modern embedding models (like those from OpenAI) produce normalized vectors, so you often don't need this step. But if you're working with embeddings from models that don't normalize, applying `VECTOR_NORMALIZE` before storing or comparing ensures consistent similarity scores.

You can calculate a vector's magnitude using the related `VECTOR_NORM` function:

```
DECLARE @v VECTOR(3) = '[3, 4, 0]';
SELECT VECTOR_NORM(@v, 'norm2') AS Magnitude;
-- Returns: 5.0
```

Inspect vectors with VECTORPROPERTY

The `VECTORPROPERTY` function returns metadata about a vector. This metadata is useful for validating data or debugging dimension mismatches.

```
VECTORPROPERTY(vector, property)
```

Two properties are supported:

- `'Dimensions'` : Returns the number of dimensions as an integer
- `'BaseType'` : Returns the data type name (currently always `float`)

```
DECLARE @v VECTOR(1536) = '[0.1, 0.2, ...]';
SELECT VECTORPROPERTY(@v, 'Dimensions') AS Dims;
-- Returns: 1536
```

This function helps when troubleshooting queries where vectors might have mismatched dimensions—a common issue when different embedding models are used.

Choose the right function for your scenario

Each function serves a distinct purpose in vector search:

Scenario	Function	When to use
Small dataset or filtered queries	<code>VECTOR_DISTANCE</code>	Under 50,000 vectors, or when a WHERE clause significantly reduces candidates

Scenario	Function	When to use
Large dataset with vector index	<code>VECTOR_SEARCH</code>	Hundreds of thousands of vectors or more, when speed matters
Comparing vectors from different models	<code>VECTOR_NORMALIZE</code>	When embedding sources vary or model doesn't produce normalized vectors
Validating vector data	<code>VECTORPROPERTY</code>	Checking dimensions during troubleshooting or data validation

Key takeaways

Vector search queries in SQL follow a predictable pattern: prepare a query vector, find neighbors using either `VECTOR_DISTANCE` (exact) or `VECTOR_SEARCH` (approximate), and order by distance. `VECTOR_NORMALIZE` and `VECTORPROPERTY` support data quality and consistency. Remember that `VECTOR_SEARCH` applies filters after finding nearest neighbors, so request more candidates than you need when combining with WHERE clauses. In the next unit, you'll learn how to combine vector search with full-text search in hybrid queries.

6. Implement hybrid search and ranking

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/06-implement-hybrid-search-ranking>

Implement hybrid search and ranking

Completed

Full-text search and vector search each have strengths, but neither alone handles every query. Full-text search excels at finding exact keywords but misses documents that express the same idea differently. Vector search captures semantic meaning but might return conceptually related items that don't contain important terms the user specified. Hybrid search combines both approaches to maximize relevance.

In this unit, you learn how to implement hybrid search in SQL by running full-text and vector searches in parallel, then merging the results using Reciprocal Rank Fusion (RRF).

Understand the hybrid search pattern

A hybrid search query runs two searches against the same data:

1. **Full-text search** uses `FREETEXTTABLE` or `CONTAINSTABLE` to find documents matching keywords, returning results ranked by BM25 relevance.
2. **Vector search** uses `VECTOR_DISTANCE` or `VECTOR_SEARCH` to find documents with similar embeddings, returning results ranked by distance.

Each search produces a ranked list. The challenge is combining these two lists into a single result set that benefits from both ranking signals.

Consider a user searching for "Azure SQL backup strategies." Full-text search finds documents containing those exact words. Vector search finds documents about "database recovery planning" or "disaster recovery for SQL databases" that are semantically related but use different terms. A hybrid approach returns both, giving users the best of both worlds.

Merge results with Reciprocal Rank Fusion

Reciprocal Rank Fusion (RRF) is a technique for combining ranked lists from different sources. Instead of comparing raw scores, which might use different scales, RRF focuses on rank positions.

The formula for RRF is:

```
RRF_score = 1/(k + rank_from_source_1) + 1/(k + rank_from_source_2)
```

The constant `k` (typically 60) prevents high-ranked items from dominating. A document ranked #1 in both lists receives a higher combined score than one ranked #1 in only one list.

This approach has two key advantages:

- **Score normalization isn't needed.** Full-text BM25 scores and cosine distances use completely different scales. RRF sidesteps these differences by using ranks instead.
- **Neither source dominates.** If vector search consistently returns higher scores than full-text search, raw score fusion would favor vector results. RRF treats both sources equally.

Implement hybrid search in SQL

Here's a complete hybrid search implementation using Common Table Expressions (CTEs) to structure the query:

```
DECLARE @searchText NVARCHAR(1000) = 'lightweight hiking boots';
DECLARE @searchVector VECTOR(1536);
DECLARE @topN INT = 50;
DECLARE @rrfK INT = 60;

-- Generate embedding for the search phrase
SELECT @searchVector = AI_GENERATE_EMBEDDINGS(@searchText USE MODEL MyEmbeddingModel);

-- Run hybrid search with RRF
WITH keyword_search AS (
```

```

SELECT TOP(@topN)
    p.ProductID,
    RANK() OVER (ORDER BY ftt.[RANK] DESC) AS keyword_rank
FROM dbo.Products p
INNER JOIN FREETEXTTABLE(dbo.Products, Description, @searchText) AS ftt
    ON p.ProductID = ftt.[KEY]
),
vector_search AS (
    SELECT TOP(@topN)
        ProductID,
        RANK() OVER (ORDER BY distance) AS vector_rank
    FROM (
        SELECT
            ProductID,
            VECTOR_DISTANCE('cosine', @searchVector, DescriptionVector) AS distance
        FROM dbo.Products
    ) AS similar_products
),
combined AS (
    SELECT TOP(@topN)
        COALESCE(ks.ProductID, vs.ProductID) AS ProductID,
        ks.keyword_rank,
        vs.vector_rank,
        COALESCE(1.0 / (@rrfK + ks.keyword_rank), 0.0) +
        COALESCE(1.0 / (@rrfK + vs.vector_rank), 0.0) AS rrf_score
    FROM keyword_search ks
    FULL OUTER JOIN vector_search vs ON ks.ProductID = vs.ProductID
)
SELECT
    p.ProductID,
    p.Name,
    p.Description,
    c.keyword_rank,
    c.vector_rank,
    c.rrf_score
FROM combined c
INNER JOIN dbo.Products p ON c.ProductID = p.ProductID
ORDER BY c.rrf_score DESC;

```

Let's break down what each CTE does:

1. **keyword_search**: Runs a full-text search using `FREETEXTTABLE`, which returns BM25-ranked results. The `RANK()` function assigns position numbers.
2. **vector_search**: Calculates cosine distance for all products and assigns rank positions based on distance (smallest first).
3. **combined**: Joins the two result sets using `FULL OUTER JOIN` so products appearing in either list are included. The RRF formula calculates the combined score. `COALESCE` handles products that appear in only one list by treating the missing rank as contributing zero.
4. **Final SELECT**: Joins back to the Products table to retrieve the full product details, ordered by the combined RRF score.

Use VECTOR_SEARCH for large datasets

When your table has hundreds of thousands of rows, replace the exact `VECTOR_DISTANCE` calculation with `VECTOR_SEARCH` for better performance:

```
vector_search AS (  
    SELECT TOP(@topN)  
        t.ProductID,  
        RANK() OVER (ORDER BY s.distance) AS vector_rank  
    FROM VECTOR_SEARCH(  
        TABLE = dbo.Products AS t,  
        COLUMN = DescriptionVector,  
        SIMILAR_TO = @searchVector,  
        METRIC = 'cosine',  
        TOP_N = @topN  
    ) AS s  
)
```

This version uses the DiskANN index for approximate nearest neighbor search, which is faster for large datasets while still providing high-quality results.

Tune hybrid search parameters

Several factors affect hybrid search quality:

Result set size (@topN): The number of candidates from each search. Larger values give RRF more candidates to work with but increase query time. Start with 50 and adjust based on your relevance testing.

The RRF constant (@rrfK): The formula uses $1/(k + \text{rank})$. A larger k (like 60) smooths out differences between ranks. A smaller k amplifies the advantage of top-ranked items. The value 60 is standard and comes from the original RRF research.

Weighting: You can weight one source more than the other by multiplying its contribution:

```
-- Weight vector search 2x more than keyword search  
COALESCE(1.0 / (@rrfK + ks.keyword_rank), 0.0) +  
COALESCE(2.0 / (@rrfK + vs.vector_rank), 0.0) AS rrf_score
```

Filtering: Apply filters before the hybrid search when possible. Filtering after RRF might eliminate highly ranked results.

Evaluate search quality

Measuring search quality helps you tune parameters and choose between approaches. Common metrics include:

Precision: Of the results returned, what percentage are relevant? High precision means few irrelevant results.

Recall: Of all relevant documents in your database, what percentage did the search find? High recall means you're not missing good results.

Mean Reciprocal Rank (MRR): How high does the first relevant result appear? MRR rewards searches that put the best result at the top.

To evaluate, you need a test set of queries with known relevant documents. Run each approach (full-text only, vector only, hybrid) and compare metrics. Hybrid search typically improves recall compared to either approach alone, often with minimal precision loss.

Key takeaways

Hybrid search combines full-text and vector search to handle both keyword-focused and concept-focused queries. Reciprocal Rank Fusion merges the ranked results without requiring score normalization, treating both sources fairly. You can tune the result set size, RRF constant, and source weights to optimize for your data. Evaluating precision, recall, and MRR helps you measure whether hybrid search improves relevance for your specific use case.

7. Exercise - Implement intelligent search with full-text, vector, and hybrid queries

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/07-exercise-implement-intelligent-search>

Exercise - Implement intelligent search with full-text, vector, and hybrid queries

Completed

In this exercise, you implement different search approaches in an Azure SQL Database. You create full-text indexes, run vector searches using stored embeddings, and combine both techniques with hybrid search using Reciprocal Rank Fusion (RRF). You then compare how each search approach handles the same query and observe the differences in results and performance.

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

8. Knowledge check

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/08-knowledge-check>

Knowledge check

Completed

9. Summary

<https://learn.microsoft.com/en-us/training/modules/design-implement-intelligent-search-with-sql/09-summary>

Summary

Completed

This module covered designing and implementing intelligent search capabilities for SQL Server and Azure SQL Database. You learned how to evaluate full-text search, vector search, and hybrid search approaches for different query types. You also explored the T-SQL functions for vector operations and implemented hybrid search using Reciprocal Rank Fusion to combine ranked results from multiple sources.

After completing this module, you can:

- To select the right approach for different query types, evaluate full-text search, vector search, and hybrid search.
- Implement full-text search using full-text indexes and query predicates.

- Prepare SQL databases for vector search by storing embeddings and choosing distance metrics.
- Write vector search queries using VECTOR_DISTANCE and VECTOR_SEARCH functions.
- Combine full-text and vector search results using Reciprocal Rank Fusion (RRF).

Additional reading

- [Full-text search overview](#)
- [Vector data type](#)
- [VECTOR_DISTANCE function](#)
- [VECTOR_SEARCH function](#)
- [Reciprocal Rank Fusion in hybrid search](#)

Design and implement RAG with SQL

1. Introduction

<https://learn.microsoft.com/en-us/training/modules/design-implement-rag-with-sql/1-introduction>

Introduction

Completed

Large language models can answer questions, summarize content, and generate text. But they only know what they learned during training. Ask about your company's products, customer orders, or compatible components, and the model has no useful answer. Retrieval Augmented Generation (RAG) solves this problem by giving the model access to your data at query time. Instead of hoping the model knows your product catalog, you retrieve the relevant products from your database and include them in the prompt. The model then generates a response grounded in actual, current information.

RAG keeps AI processing inside the database layer, where your data already lives. This approach avoids moving data between systems and lets you use Transact-SQL to control what context the model receives. The result is an application that can answer specific questions using real, up-to-date information from your tables.

Imagine a retail team building a customer support application on top of a product database. A customer asks "Which gloves work best for cold weather cycling?" The application needs to search the product catalog, find matching accessories, and generate a helpful response using an LLM. This workflow is RAG in action: retrieve relevant data, augment the prompt with that data, and generate a grounded response. By building this workflow in SQL, the team keeps the retrieval step close to the data and avoids building a separate retrieval service.

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

- Identify when RAG is the right approach for your application.
- Convert SQL query results to JSON for Large Language Model (LLM) processing.
- Construct prompts that combine instructions with database context.
- Call LLM endpoints from SQL.
- Parse LLM responses and return answers to your application.

2. Identify RAG use cases and architecture

<https://learn.microsoft.com/en-us/training/modules/design-implement-rag-with-sql/2-identify-rag-use-cases-architecture>

Identify RAG use cases and architecture

Completed

Large Language Models (LLM) don't know your organization. Your products, customers, policies. None of that exists in their training data. When a customer asks "Which pedals fit the mountain bike I bought last month?", the model can only guess.

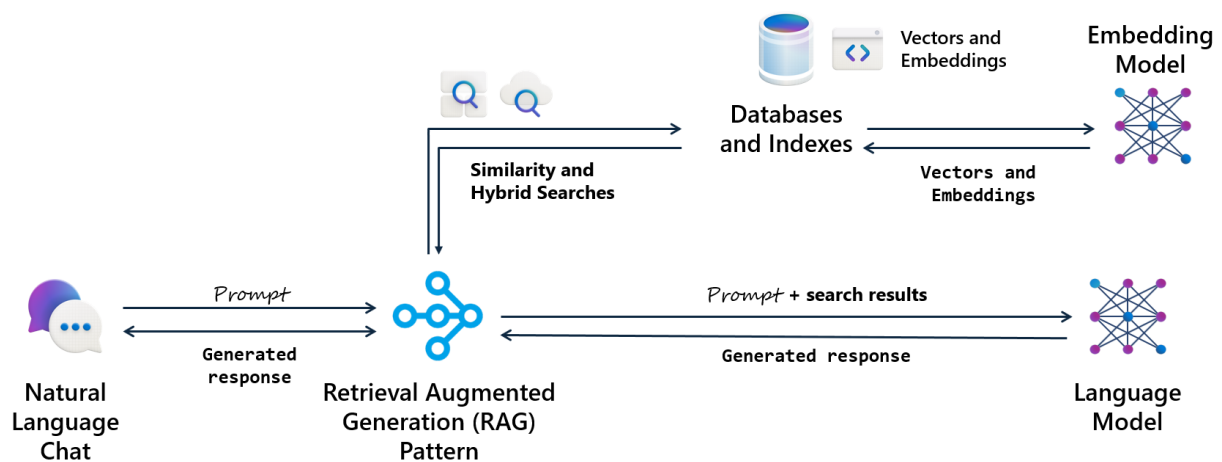
Retrieval Augmented Generation (RAG) fixes this issue by feeding the model your data when it needs to answer. You retrieve information from your database, add it to the prompt, and let the model respond based on what you provided. The model doesn't need to "know" your catalog because you give it what it needs for each question.

In previous modules, you learned to find relevant data using full-text, vector, and hybrid search. That step becomes the retrieval piece in RAG. Now you complete the pattern by augmenting prompts with retrieved data and generating grounded responses.

Understand how RAG works

RAG combines three steps. First, retrieve relevant data from your database. For the pedals question, find the customer's order, identify the bike model, and look up compatible components. Second, augment the prompt by adding that data as context. The model receives both the question and the information to answer it. Third, the model generates a response grounded in your data rather than its training.

The following diagram shows a RAG architecture with SQL Database. A natural language chat sends a prompt to a RAG orchestration layer, which runs similarity and hybrid searches against the database. The database exchanges vectors and embeddings with an embedding model. The orchestration layer forwards the prompt plus search results to a language model, which returns a grounded response.



This approach differs from fine-tuning, which retrain a model on your data. Fine-tuning bakes knowledge in permanently. RAG keeps data in your database and supplies only what each request needs. When specs change or new components arrive, answers reflect those changes immediately.

Recognize when RAG fits your scenario

RAG works when data changes frequently. Fine-tuned models can't keep pace with daily inventory updates, but RAG retrieves current data every time. RAG also fits when you need to trace answers. You control retrieval, so you know which records informed each response.

Privacy matters too. Fine-tuning sends data to the model provider for training. RAG keeps data in your database and sends only the context needed per request. For sensitive customer data or proprietary knowledge, RAG provides a cleaner separation.

Consider whether users ask domain-specific questions. General models handle common knowledge, but struggle with your products, processes, or history. RAG bridges that gap by grounding responses in your information.

Explore common RAG scenarios

RAG applies wherever users need answers that depend on your specific data. Here are some scenarios where SQL-based RAG adds value:

- **Customer support:** A customer asks "Can I return the bike I ordered three weeks ago?" The system retrieves their order history, checks your return policy, and generates an answer that cites the specific order and applicable policy terms.
- **Product configuration:** A shopper says "Build me a complete mountain biking setup under \$2,000." RAG retrieves compatible frames, components, and accessories from your catalog, checks current inventory and pricing, and assembles a configuration that meets the budget.

- **Internal knowledge base:** An employee needs the expense reimbursement process for international travel. RAG searches policy documents stored in your database and summarizes the relevant steps with links to forms.
- **Sales enablement:** A sales rep preparing for a call says "Draft a follow-up email for this customer based on their recent purchases and open issues." RAG pulls order history and support tickets, then generates a personalized email grounded in the actual relationship context.
- **Technical documentation:** A developer asks "Generate a migration script to move our user table to the new schema." RAG searches your documentation and schema definitions, retrieves the relevant structures, and produces a script based on your actual table definitions.

Each scenario follows the same pattern: retrieve context from SQL, augment the prompt, and generate a grounded response. The difference is what you retrieve and how you structure it. There's an endless variety of use cases where RAG enhances LLM capabilities with your data.

Understand SQL's role in RAG architecture

Azure SQL Database and SQL Server 2025 participate in RAG without hosting models. Your database stores products, orders, documents, and embeddings. T-SQL handles retrieval using full-text, vector, or hybrid search. You format results as JSON, construct prompts, call the LLM via `sp_invoke_external_rest_endpoint`, and parse responses.

The LLM runs elsewhere, for example Azure OpenAI. Your T-SQL orchestrates the flow: retrieve data, build a prompt, call the model, extract the answer, return it. This flow keeps logic close to data and lets you add AI to existing apps by modifying T-SQL rather than rearchitecting.

Note

SQL Server 2025 and Azure SQL Database both support RAG. However, `sp_invoke_external_rest_endpoint` is enabled by default only in Azure SQL Database. In SQL Server 2025, enable it with `sp_configure`.

Key takeaways

RAG grounds model responses in your database by combining retrieval, augmentation, and generation. It fits when data changes often, when you need traceable answers, or when privacy requires keeping data in your control. SQL handles retrieval and orchestration; the model handles generation. Next, you prepare context as JSON, construct prompts, and call LLM endpoints.

3. Prepare retrieval context for augmentation

Prepare retrieval context for augmentation

Completed

You now understand what RAG is and when to use it. In the following units, you learn how to implement each step of the RAG workflow in T-SQL.

Consider the following scenario: A customer asks "Which pedals fit the mountain bike I bought last month?" You query the database, find their order, identify the bike model, and locate compatible components. That step represents the "R" in RAG: retrieval. Let's retrieve and prepare that data for the language model. There are many ways to retrieve data from SQL, but for RAG, the goal is to provide the model with structured context it can actually use.

Convert relational data to JSON

Language models process text, not relational structures. If you pass raw query results, the model has no way to interpret column names, data types, or relationships. JSON solves this issue by preserving structure in a text format. Field names stay attached to values. Nested objects represent relationships. The model can read "ProductName": "Mountain-500" and reference that specific product in its response.

JSON also keeps things predictable. When the model returns an answer, you know the answer came from fields you explicitly provided. If you dumped unstructured text instead, you'd have less control over what the model uses to formulate its response.

Format query results with FOR JSON

No need to build JSON strings by hand. Just add `FOR JSON` to the end of your `SELECT` and SQL does the work:

- `FOR JSON AUTO` formats output based on your query structure. Column names become field names automatically.

```
SELECT Name, ListPrice, Color
FROM Production.Product
WHERE ProductID = @ProductID
FOR JSON AUTO;
```

This query returns something like: `[{"Name": "Mountain-500 Black, 48", "ListPrice": 564.99, "Color": "Black"}]`

- `FOR JSON PATH` gives you explicit control over the JSON structure. You define the field names and nesting using column aliases.

```
SELECT
    Name AS 'product.name',
    ListPrice AS 'product.price',
    Size AS 'product.size'
FROM Production.Product
WHERE ProductID = @ProductID
FOR JSON PATH;
```

This query returns something like: [{"product":{"name":"Mountain-500 Black, 48", "price":564.99, "size":"48"}}]

Control JSON output options

A few options let you shape the output:

- `WITHOUT_ARRAY_WRAPPER` removes the square brackets when you're retrieving a single record. This option is useful for RAG because you often retrieve one customer, one product, or one order at a time.

```
SELECT Name, Description, Size, Weight
FROM Production.Product
WHERE ProductID = @ProductID
FOR JSON PATH, WITHOUT_ARRAY_WRAPPER;
```

- `INCLUDE_NULL_VALUES` keeps null fields in the output instead of omitting them. Use this parameter when the absence of a value is meaningful.
- `ROOT('name')` wraps the entire output in a named root element, which can help the model understand what kind of data it's receiving.

Choose what to include

Not every column belongs in your context. For example, internal IDs, audit timestamps, and warehouse codes don't help the model answer customer questions. They just consume tokens and add noise.

For a product question, include the product name, description, specifications, and pricing. Skip the rowguid, modified date, and discontinued flag unless they're directly relevant.

Token limits matter too. Every token you send costs money and counts against the model's context window. If you're retrieving multiple products, keep each one lean. The model performs better with focused context than with everything you could possibly include.

Combine multiple sources

RAG context often comes from multiple tables. A pedal compatibility question needs product details, specifications, and related components. You might start by finding relevant products with vector search, then join other tables to build out the full picture.

Here's what that looks like in practice. First, convert the user's question into an embedding. Then use `VECTOR_DISTANCE` to find the closest matches, join the related tables, and format everything as JSON:

```
DECLARE @userQuestion NVARCHAR(1000) = 'Which pedals are compatible with the Mounta
DECLARE @questionVector VECTOR(1536);
DECLARE @context NVARCHAR(MAX);

-- Generate embedding for the question
SELECT @questionVector = AI_GENERATE_EMBEDDINGS(@userQuestion USE MODEL my_embeddin

-- Find relevant products and format as JSON
SET @context = (
    SELECT TOP 3
        p.Name AS ProductName,
        p.Color,
        p.Size,
        pc.Name AS Category,
        pm.Name AS Model
    FROM Production.Product p
    INNER JOIN Production.ProductSubcategory ps ON p.ProductSubcategoryID = ps.Prod
    INNER JOIN Production.ProductCategory pc ON ps.ProductCategoryID = pc.ProductC
    INNER JOIN Production.ProductModel pm ON p.ProductModelID = pm.ProductModelID
    ORDER BY VECTOR_DISTANCE('cosine', p.DescriptionVector, @questionVector)
    FOR JSON PATH
);
```

The variable `@context` now holds a JSON string ready for the prompt. This JSON string is your retrieval result formatted for augmentation.

Key takeaways

The goal is to give the language model structured context it can actually use. JSON preserves the meaning of your data while keeping it readable as text. Use `FOR JSON AUTO` when you want quick results and `FOR JSON PATH` when you need control over field names and nesting. Add `WITHOUT_ARRAY_WRAPPER` for single-record queries. Keep your JSON lean by including only the columns the model needs. Less noise means better answers and lower token costs.

4. Augment prompts with database context

Augment prompts with database context

Completed

Retrieval gets you the data. But a JSON blob on its own doesn't answer anyone's question. If a customer asks about bike pedal compatibility, you've got the product info from your database, but now you need to tell the language model what to do with it. That step represents the "A" in RAG: augmentation.

Understand the chat message structure

Different models have different APIs, but most chat-based models follow a similar pattern. We use Azure OpenAI as our example, but the concepts apply broadly.

Azure OpenAI chat models expect messages with roles. The **system** role defines how the assistant should behave. The **user** role contains the original question and the database context into which the model should ground its answer. Optionally, an **assistant** role can hold previous responses in a multi-turn conversation. Here's a simplified example of a prompt structure:

```
{
  "messages": [
    {
      "role": "system",
      "content": "You are a helpful assistant that answers questions about products",
    },
    {
      "role": "user",
      "content": "Context: {retrieved_data}\n\nQuestion: {user_question}"
    }
  ]
}
```

In this example, `{retrieved_data}` would be the JSON you built from your product query, and `{user_question}` would be something like "Which pedals are compatible with the Mountain-500?"

The **system** message sets ground rules. The **user** message combines your database context with the actual question.

Build prompts in T-SQL

You could build this JSON in your application code, but it's often convenient to keep it all in the database. Your retrieval query, your context formatting, and your prompt construction live together, and T-SQL's `JSON_OBJECT` and `JSON_ARRAY` functions handle the JSON formatting.

```
DECLARE @userQuestion NVARCHAR(1000) = 'Which pedals are compatible with the Mounta

-- @context contains retrieved product data as JSON from the retrieval step

DECLARE @systemMessage NVARCHAR(MAX) = 'You are a customer service assistant for Ac

DECLARE @userMessage NVARCHAR(MAX) = 'Product information: ' + @context + CHAR(10)

DECLARE @payload NVARCHAR(MAX) = JSON_OBJECT(
    'messages': JSON_ARRAY(
        JSON_OBJECT('role': 'system', 'content': @systemMessage),
        JSON_OBJECT('role': 'user', 'content': @userMessage)
    ),
    'max_tokens': 500,
    'temperature': 0.7
);
```

The `@payload` variable now contains a complete request body ready for the Azure OpenAI API.

Ground the model in your data

Grounding tells the model to use your data as the source of truth. Without it, the model might rely on its training data, which could be outdated or wrong for your domain. A customer asking about Adventure Works bike warranties shouldn't get generic warranty information from the internet.

Good grounding instructions set scope ("use only the provided product data"), encourage honesty ("if you don't have enough information, say so"), and can specify format ("keep responses under 100 words"). Here's an example for the *system* role:

```
DECLARE @systemMessage NVARCHAR(MAX) =
'You are an Adventure Works product expert. Follow these rules:
1. Answer only using the product information provided
2. Do not invent features or specifications
3. If information is missing, tell the customer you'll need to check
4. Keep responses under 100 words
5. Suggest related products when relevant';
```

These instructions help the model stay focused and provide useful, accurate answers.

Control model behavior

The request payload includes a couple of parameters that affect how the model generates responses:

- `max_tokens` limits response length. For detailed product answers, 500 to 1,000 works well.

- `temperature` controls creativity on a scale from 0 to 2. Lower values (0.3 to 0.5) produce more consistent, factual responses. Higher values let the model get more creative, which you usually don't want for RAG.

Controlling the tokens and temperature helps ensure the model behaves predictably when grounded in your data.

Construct the complete payload

Here's how you might build a RAG prompt for a product question using Adventure Works tables:

```
DECLARE @userQuestion NVARCHAR(1000) = 'Which pedals are compatible with the Mounta
DECLARE @questionVector VECTOR(1536);
DECLARE @context NVARCHAR(MAX);
DECLARE @payload NVARCHAR(MAX);

-- Generate embedding for the question
SELECT @questionVector = AI_GENERATE_EMBEDDINGS(@userQuestion USE MODEL my_embeddin

-- Get product context using vector search
SET @context = (
    SELECT TOP 3
        p.Name AS ProductName,
        p.Color,
        p.Size,
        pc.Name AS Category,
        pm.Name AS Model
    FROM Production.Product p
    INNER JOIN Production.ProductSubcategory ps ON p.ProductSubcategoryID = ps.Prod
    INNER JOIN Production.ProductCategory pc ON ps.ProductCategoryID = pc.ProductC
    INNER JOIN Production.ProductModel pm ON p.ProductModelID = pm.ProductModelID
    ORDER BY VECTOR_DISTANCE('cosine', p.DescriptionVector, @questionVector)
    FOR JSON PATH
);

-- Build the prompt
SET @payload = JSON_OBJECT(
    'messages': JSON_ARRAY(
        JSON_OBJECT(
            'role': 'system',
            'content': 'You are an Adventure Works product assistant. Use only the
        ),
        JSON_OBJECT(
            'role': 'user',
            'content': 'Product data: ' + @context + CHAR(10) + 'Question: ' + @use
        )
    ),
    'max_tokens': 500,
    'temperature': 0.5
);
```

The `@payload` variable now contains everything the model needs: your grounding instructions, the retrieved product data, and the customer's question. All you need to do is send it to the model endpoint and handle the response.

Key takeaways

The prompt is where retrieval becomes useful. Your JSON context means nothing unless you tell the model how to use it. Set clear grounding rules in the system message so the model sticks to your data. Keep temperature low for factual answers. Use `JSON_OBJECT` and `JSON_ARRAY` to build valid JSON payloads directly in T-SQL.

5. Generate and process RAG responses

<https://learn.microsoft.com/en-us/training/modules/design-implement-rag-with-sql/5-generate-process-rag-responses>

Generate and process RAG responses

Completed

A customer wants to know which pedals fit their Mountain-500 bike. You use vector search to find the relevant products, formatted them as JSON, and built a prompt with grounding instructions. The retrieval and augmentation steps are done. Now comes the "G" in RAG: generation. This step is where you send everything to a language model and get back an answer.

Think about it like this: you gathered all the ingredients (retrieved data), prepared the recipe (augmented prompt), and now you're putting it in the oven (calling the model) to bake the final dish (the response). The model uses the context you provided to generate a grounded answer.

Call the model from SQL

You might think this step requires leaving T-SQL and writing application code. But SQL Server and Azure SQL Database can call REST endpoints directly using `sp_invoke_external_rest_endpoint`. This stored procedure sends HTTPS requests to external services and returns the response. The stored procedure is enabled by default in Azure SQL Database and can be enabled in SQL Server 2025 using `sp_configure`.

Here's the basic pattern:

```
DECLARE @response NVARCHAR(MAX);  
DECLARE @returnValue INT;
```

```
EXECUTE @returnValue = sp_invoke_external_rest_endpoint
    @url = N'https://<endpoint>.openai.azure.com/openai/deployments/<model>/chat/completions?api-version=2023-03-15',
    @method = 'POST',
    @payload = @payload,
    @credential = [https://<endpoint>.openai.azure.com],
    @response = @response OUTPUT;
```

The `@url` parameter points to your Azure OpenAI deployment endpoint. The `@method` is usually 'POST' for generation requests. The `@payload` contains the JSON prompt you previously built. The `@credential` references a database scoped credential that holds your authentication details. The `@response` output parameter captures the model's response.

When you call the deployment endpoint of your model with your augmented prompt, the model processes it and returns the result. The stored procedure returns 0 when the HTTP call succeeds with a 2xx status code, or the actual HTTP status code when it fails.

Authenticate with the endpoint

The `@credential` parameter references a database scoped credential that holds your authentication details. You set up these credentials when you create an external model, using either managed identity or an API key. The same credential works for both external model calls and direct REST endpoint calls with `sp_invoke_external_rest_endpoint`.

Extract the answer from the response

When the model finishes processing your prompt, `sp_invoke_external_rest_endpoint` returns the result wrapped in a standard envelope. The stored procedure adds metadata about the HTTP transaction, then nests the actual API response inside a `result` property:

```
{
  "response": {
    "status": {
      "http": {
        "code": 200,
        "description": "OK"
      }
    }
  },
  "result": {
    "choices": [
      {
        "message": {
          "role": "assistant",
          "content": "The Mountain-500 is compatible with several pedal options..."
        }
      }
    ]
  }
}
```

```
}  
}
```

The answer you want lives at `$.result.choices[0].message.content`. To get there, use `JSON_VALUE`, which extracts scalar values from JSON:

```
IF @returnValue = 0  
BEGIN  
    DECLARE @answer NVARCHAR(MAX);  
    SET @answer = JSON_VALUE(@response, '$.result.choices[0].message.content');  
    SELECT @answer AS AssistantResponse;  
END  
ELSE  
BEGIN  
    SELECT  
        @returnValue AS HttpStatus,  
        JSON_VALUE(@response, '$.response.status.http.description') AS ErrorDescription;  
END
```

If you ever need to extract a JSON object or array rather than a single value, use `JSON_QUERY` instead. For most RAG scenarios, `JSON_VALUE` on the message content is all you need.

Manage errors and retries

Calling an external service from a database can introduce failure modes you don't encounter with local queries. The endpoint might be temporarily unavailable, your request might get rate-limited, or authentication could fail. Your SQL queries need to handle these conditions gracefully.

The return value from `sp_invoke_external_rest_endpoint` tells you what happened. A 0 means the HTTP call succeeded with a 2xx status. For failures, the return value is the HTTP status code itself. For example, a 429 status means the service is throttling your requests. A 401 or 403 points to credential problems:

```
IF @returnValue = 0  
    SET @answer = JSON_VALUE(@response, '$.result.choices[0].message.content');  
ELSE IF @returnValue = 429  
    RAISERROR('Service is busy. Try again later.', 16, 1);  
ELSE IF @returnValue = 401 OR @returnValue = 403  
    RAISERROR('Authentication failed. Check your credential configuration.', 16, 1);  
ELSE  
BEGIN  
    DECLARE @errorMsg NVARCHAR(500) = 'API call failed with status ' + CAST(@returnValue AS NVARCHAR(10));  
    RAISERROR(@errorMsg, 16, 1);  
END
```

For transient failures like timeouts or temporary service unavailability, the stored procedure can retry automatically. Add the `@retry_count` parameter and in this example, it attempts the call up to three times before giving up:

```
EXECUTE @returnValue = sp_invoke_external_rest_endpoint
    @url = @url,
    @payload = @payload,
    @credential = @credentialName,
    @retry_count = 3,
    @response = @response OUTPUT;
```

This parameter handles the common case where a request fails once but succeeds on the next attempt.

Build a complete RAG stored procedure

You now know each piece of the RAG puzzle. Let's put them together into a single stored procedure that a customer support application could call. The procedure accepts a natural language question and returns an answer grounded in your product data.

Here's what the procedure does:

1. Converts the question into an embedding so you can compare it against your product descriptions.
2. Finds the most relevant products using vector distance to compare the question embedding against each product's precomputed description embedding.
3. Builds the prompt with a system message that tells the model to stick to the provided data, and a user message that combines the retrieved products with the original question.
4. Sends everything to Azure OpenAI.
5. Extracts the answer from the response and return it to the caller.

```
CREATE PROCEDURE dbo.AskProductQuestion
    @Question NVARCHAR(1000),
    @Answer NVARCHAR(MAX) OUTPUT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @questionVector VECTOR(1536);
    DECLARE @context NVARCHAR(MAX);
    DECLARE @payload NVARCHAR(MAX);
    DECLARE @response NVARCHAR(MAX);
    DECLARE @returnValue INT;

    -- Step 1: Convert question to embedding
    SELECT @questionVector = AI_GENERATE_EMBEDDINGS(@Question USE MODEL my_embedding);

    -- Step 2: Retrieve relevant products using vector search
    SET @context = (
        SELECT TOP 3
            p.Name AS ProductName,
            p.Color,
            p.Size,
            pm.Name AS Model
```

```

FROM Production.Product p
INNER JOIN Production.ProductModel pm ON p.ProductModelID = pm.ProductModelID
ORDER BY VECTOR_DISTANCE('cosine', p.DescriptionVector, @questionVector)
FOR JSON PATH
);

-- Step 3: Build augmented prompt
SET @payload = JSON_OBJECT(
    'messages': JSON_ARRAY(
        JSON_OBJECT('role': 'system', 'content': 'You are an Adventure Works product expert'),
        JSON_OBJECT('role': 'user', 'content': 'Products: ' + @context + ' Question: ' + @question)
    ),
    'max_tokens': 500,
    'temperature': 0.5
);

-- Step 4: Call the model
EXECUTE @returnValue = sp_invoke_external_rest_endpoint
    @url = N'https://adventureworks-openai.openai.azure.com/openai/deployments/gpt-4o',
    @method = 'POST',
    @payload = @payload,
    @credential = [https://adventureworks-openai.openai.azure.com],
    @response = @response OUTPUT;

-- Extract and return the answer
IF @returnValue = 0
    SET @Answer = JSON_VALUE(@response, '$.result.choices[0].message.content');
ELSE
    SET @Answer = 'Unable to process your question. Please try again.';
END;

```

You completed the RAG pipeline in T-SQL. A customer asks "Which pedals work with the Mountain-500?" Your procedure converts that question to a vector, finds the most relevant products, sends them to the model with grounding instructions, and returns an answer based on your actual inventory. No middleware, no external application code, just SQL calling AI and returning results.

Key takeaways

The *generation* step is where your RAG pipeline delivers value. You send the augmented prompt to Azure OpenAI using `sp_invoke_external_rest_endpoint`, which handles the HTTP communication without leaving T-SQL. Authentication uses the same database scoped credentials you set up when creating external models. When the response comes back, `JSON_VALUE` pulls out the assistant's answer. Build in error handling because network calls fail in ways that local queries don't. With all three RAG steps running in T-SQL, your database becomes more than storage. It becomes an intelligent service that answers questions using your data.

6. Exercise: Implement a RAG solution

<https://learn.microsoft.com/en-us/training/modules/design-implement-rag-with-sql/6-exercise-implement-rag-solution>

Exercise: Implement a RAG solution

Completed

In this exercise, you implement a complete Retrieval Augmented Generation (RAG) solution using Azure SQL Database. You retrieve relevant data from your database, format it as JSON context, construct an augmented prompt, call a Large Language Model (LLM) endpoint, and extract the response.

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-rag-with-sql/7-knowledge-check>

Knowledge check

Completed

8. Summary

Summary

Completed

Retrieval Augmented Generation connects your database to the capabilities of large language models. Instead of relying on a model's training data, you provide current, relevant information from your own tables.

The entire RAG pattern executes in T-SQL. Your database orchestrates the flow: search, format, prompt, call, parse. You can add AI capabilities to existing applications by modifying stored procedures, without rearchitecting your application stack.

In this module, you learned how to:

- **Identify RAG use cases:** Recognize scenarios where grounding Large Language Model (LLM) responses in database content improves accuracy and relevance
- **Prepare context from SQL:** Use `FOR JSON` to convert query results into text that LLMs can process effectively
- **Construct augmented prompts:** Build request payloads that combine system instructions, retrieved context, and user questions
- **Execute the RAG pipeline:** Call Azure OpenAI endpoints using `sp_invoke_external_rest_endpoint` and parse the responses

Learn more

- [sp_invoke_external_rest_endpoint \(Transact-SQL\)](#)
- [Intelligent applications and AI](#)
- [Format query results as JSON with FOR JSON](#)
- [Intelligent applications and AI FAQ](#)