

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.

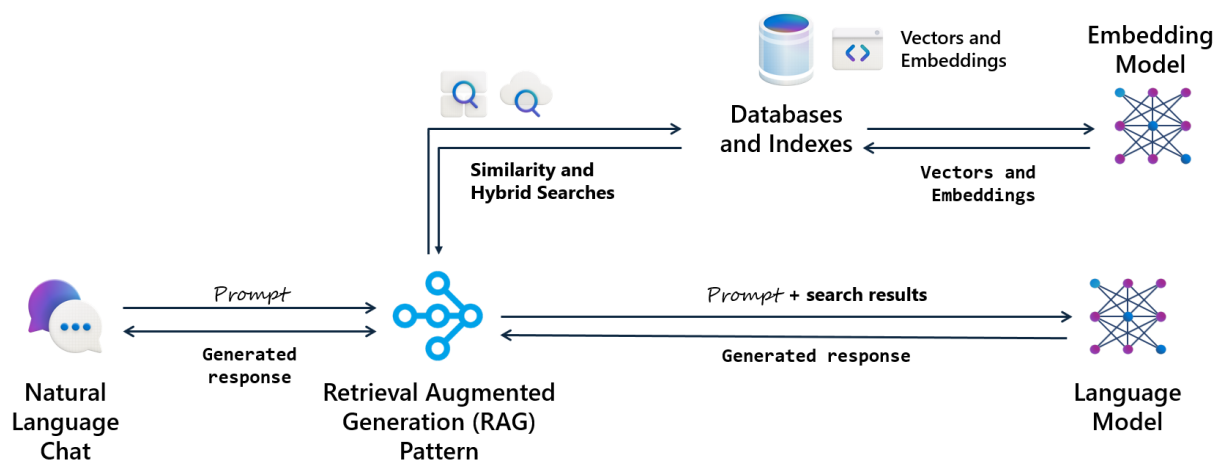
Retrieval Augmented Generation (RAG) fixes this issue by feeding the model your data when it needs to answer. You retrieve information from your database, add it to the prompt, and let the model respond based on what you provided. The model doesn't need to "know" your catalog because you give it what 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.ProductCa
    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=2023-03-01',
    @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](#)