ChatGPT

AI-Powered Codebase Understanding with SQL Server 2025

SQL Server 2025 introduces new AI features that enable retrieval-augmented generation (RAG) on your own data, making it possible to build a local AI assistant for code comprehension and refactoring. This report explains how to ingest an enterprise codebase into SQL Server 2025 and leverage its semantic search, vector embeddings, and model integration features to create a system that behaves like a "senior engineer." We focus on fully on-premises solutions – using local data and open-source models – to avoid sending any code to cloud services.

1. Ingesting the Codebase into SQL Server 2025

Storing code in the database: Begin by importing your code files (VB.NET, VB6, T-SQL scripts, SSRS report files, C#, C++ source, etc.) into SQL Server as text data. A common approach is to create a table (e.g., `CodeFiles`) with columns for file identifier, file path or module name, language/type, and the file's contents (as `NVARCHAR(MAX)`). This provides a centralized repository of all source code that SQL Server can index and query. You can use custom scripts or ETL tools to read files from the repository and insert them as rows in this table. Ensure to store any relevant metadata (such as language or project) as columns so you can filter searches by those attributes later.

Chunking files for RAG: Large files should be broken into smaller *"chunks"* of text for effective retrieval. SQL Server 2025 offers a built-in table-valued function `AI_GENERATE_CHUNKS` to split text into fragments of a specified size [1]. This function can produce fixed-size chunks (with optional overlap) from each source text, which is ideal for preparing code snippets for embedding. For example, you might chunk each file into segments of, say, 300-500 characters (or tokens) with a small overlap (to avoid losing context between chunks). Each chunk will then serve as a unit of retrieval. You can automate this by storing chunks in a separate table (e.g., `CodeChunks`) linked to the original file via an ID. The `AI_GENERATE_CHUNKS` function makes it easy to chunk multiple rows at once – for instance, by using a `CROSS APPLY` on the `CodeFiles` table [1] [2]. Be sure to enable the preview feature flag and set the database compatibility level to 170 or higher to use this function (since it's a preview feature in SQL 2025).

Why chunking? Splitting files ensures that each piece can be semantically indexed and retrieved individually. Current LLMs have context length limits, so it's not feasible to feed a whole large file into a prompt. By indexing smaller code segments, the RAG system can fetch only the most relevant pieces of code to answer a question or suggest an improvement. This also improves semantic search granularity – a search query can match a small function or paragraph of code rather than an entire file.

Indexing and organizing: Once chunks are created, consider adding full-text indexes or using SQL 2025's text indexing to allow keyword-based searches in tandem with semantic search. For example, you might maintain a full-text index on the `CodeChunks` table for traditional exact keyword matches (like finding all occurrences of a function name), while also using vector-based semantic search for conceptual queries. Basic metadata filtering (by language, module, date, etc.) can be done with normal SQL `WHERE` clauses. By

organizing code this way, SQL Server becomes a **code knowledge base** that can be queried in natural language.

2. Semantic Search with Built-in Vector Embeddings

SQL Server 2025 introduces **native vector search capabilities** that enable semantic querying of text data [3] [4]. In practice, this means you can ask natural language questions or describe a code concept, and the database can retrieve code snippets that *"semantically"* match the query – even if they don't share keywords. This is powered by **embeddings**: numerical representations of text generated by AI models. Key features include:

- **Vector data type and indexes:** SQL 2025 adds a new `VECTOR` column type to store embedding arrays (e.g., 768-dimensional float vectors) [5]. You can add a `VECTOR(...)` column to the `CodeChunks` table to hold each chunk's embedding. The database also supports indexing these vectors using the DiskANN algorithm (an approximate nearest neighbors index) for fast similarity search [6]. Creating a DiskANN index on the vector column lets you efficiently find the *k* nearest vectors to a query vector, even in a large table. (Note: DiskANN indexes in the preview are currently read-only; you'd rebuild or recreate the index if embeddings change [7].)

- **Embedding generation in T-SQL:** SQL 2025 can generate embeddings for your data *within* the database by calling out to AI models. The function `AI_GENERATE_EMBEDDINGS` produces a vector embedding from a given input string using a specified external AI model [8]. In practice, you would first register an embedding model (see Section 3) and then invoke it to vectorize each code chunk. For example, after defining an external embedding model, you could run an `UPDATE` like:

  ```
  UPDATE CodeChunks
  SET EmbedVec = AI_GENERATE_EMBEDDINGS(Content USING MODEL MyEmbedModel);
  ```

  This will call the model for each chunk's text and store the resulting vector in the `EmbedVec` column [8]. The initial embedding generation can be time-consuming (as it involves processing a lot of text through the model). In one demo, vectorizing ~300 short texts took ~30 seconds [8], while very large corpora might take hours or days. However, it's a one-time (or infrequent) process; you can batch it or update incrementally for new code. The end result is that each code fragment now has a machine-understandable semantic representation.

- **Semantic querying:** With embeddings in place, you can perform semantic similarity searches directly in SQL. For instance, to find code related to "sending email notifications," you could take that query string, embed it (using the same model) into a vector, and then use the `VECTOR_DISTANCE` or `VECTOR_SEARCH` function to measure similarity against the stored vectors [9] [10]. A query might look like:

  ```
  DECLARE @q nvarchar(100) = N'Where do we send email notifications?';
  DECLARE @qVec vector(768) = AI_GENERATE_EMBEDDINGS(@q USING MODEL
  MyEmbedModel);
  ```

```
SELECT TOP 5 FilePath, ChunkText
FROM CodeChunks
ORDER BY VECTOR_DISTANCE('cosine', @qVec, EmbedVec);
```

This ranks code chunks by how *closely related* they are to the query's meaning, rather than by exact keywords [9]. Chunks with high cosine similarity (low distance) to the query vector are likely to contain logic about email notifications (e.g., code calling an SMTP API or constructing email content), even if they don't explicitly use the same words as the question. For faster performance on large tables, you would use `VECTOR_SEARCH` with a DiskANN index (which does approximate search) – this can dramatically speed up finding the nearest neighbors by searching the vector index graph instead of scanning every vector [6].

- **Combining semantic and symbolic filters:** A powerful aspect of SQL 2025's approach is that you can combine semantic search with traditional SQL filtering. For example, you could query "how is database encryption handled" and restrict results to only T-SQL code chunks, by adding a condition like `WHERE Language = 'SQL'` alongside the vector similarity clause. You can also leverage **full-text search** or pattern matching if needed. The vector search operates alongside full-text search and normal predicates in the same query [3]. This means you might first narrow down candidates with a keyword filter (e.g., only chunks containing the word "encrypt"), and then use `VECTOR_DISTANCE` to sort those by semantic relevance. This hybrid approach ensures precision: it uses contextual meaning to rank results, but also allows deterministic filtering when you have specific criteria (such as file type or project).

By ingesting code into SQL and augmenting it with embeddings, you essentially create a **semantic index** of your entire codebase. You can pose high-level questions ("Find all code that deals with user authentication") and get pointers to relevant snippets across different languages or modules, even if they use different terminology. This lays the groundwork for an AI assistant to reason about the code. In summary, SQL Server 2025 provides the *"data layer"* for RAG: it stores and indexes your code in a way that's friendly to LLMs, supporting the retrieval of context that the LLM will later use [11].

3. Integrating Local LLMs with SQL Server 2025

With the code and embeddings in place, the next step is to bring in the **"generation"** part of RAG – the large language model itself. SQL Server 2025 is designed to integrate with AI models through a flexible mechanism that works for both cloud-based and local models [12]. The key capability here is the ability to call external AI endpoints (via REST APIs) directly from T-SQL, which has been introduced in this version. Here's how a developer can leverage it for a local setup:

- **External model definitions:** SQL 2025 allows you to register *external AI models* in the database. An external model is basically a reference to a REST API endpoint that hosts an AI model. The new T-SQL syntax `CREATE EXTERNAL MODEL` is used to define these [13] [14]. When creating a model, you specify the endpoint URL, the API protocol/format, and the type of model (e.g., embeddings model). The engine supports multiple providers out-of-the-box via the `API_FORMAT` setting – for example, you can specify Azure OpenAI or OpenAI's API, but importantly also `Ollama` (for local Llama models) or `ONNX Runtime` for local models [15]. This means you can integrate *any* model that exposes a compatible REST interface, including one running on your own machine or server. The

model definitions are stored in the database metadata, and you can grant/restrict permissions on them as needed [16].

- **Local REST endpoint approach:** If you have a local LLM running as a service, you can point SQL Server to it. For example, one popular setup is using **Ollama**, an open-source tool that runs LLaMA-family models on your machine and provides an HTTP API. In the SQL 2025 preview, developers demonstrated connecting to a local Ollama instance hosting a model [13] [17]. The external model was created with `LOCATION = 'https://localhost:11434/api'` (for example) and `API_FORMAT = 'Ollama'`, telling SQL that this is an Ollama-compatible endpoint [18] [19]. SQL Server will communicate with that endpoint using REST calls under the hood. **Important:** SQL Server requires the endpoint to use HTTPS (TLS encryption) even if it's local [20] [21]. In practice, that means you might need to run a local reverse proxy (like Nginx or Caddy) with a self-signed certificate to expose your model on `https://127.0.0.1:<port>` and trust that cert on the SQL host [22] [23]. This is a one-time setup to ensure secure, encrypted communication between the DB engine and the model service (even though it's all behind your firewall).

- **Local ONNX runtime approach:** As an alternative to an HTTP API, SQL 2025 also offers an **embedded runtime** path for certain models via ONNX. You can deploy a model in ONNX format on the SQL server machine and have the database engine load and execute it locally [24] [25]. For example, Microsoft's docs show how to set up the *all-MiniLM-L6-v2* embedding model (an open-source MiniLM model) by copying the ONNX model and a tokenizer to a folder on the server and registering it with `API_FORMAT = 'ONNX Runtime'` [26] [27]. When you call `AI_GENERATE_EMBEDDINGS` using that external model, SQL Server will spin up an internal runtime host, load the ONNX model, and produce embeddings without any network calls [28]. This is a purely offline method – ideal if you have a smaller model that meets your needs. Currently, ONNX integration is focused on embedding models (MODEL_TYPE = EMBEDDINGS) [29] [30], so it's perfect for vector generation. In theory, you could also serve a small generative model via ONNX, but most code-oriented LLMs are too large/complex for ONNX execution inside SQL. For generation, the REST approach (local API) is more practical.

- **Using models from T-SQL:** Once an external model is set up, you can use new AI functions to invoke it. We already covered `AI_GENERATE_EMBEDDINGS(model, text)`. Although as of SQL 2025 preview there isn't a built-in `AI_GENERATE_TEXT` or chat function, you can still call a model for completions using the generic **REST invoke** procedure. SQL has a system stored procedure `sp_invoke_external_rest_endpoint` that lets you call any HTTP endpoint from T-SQL (this existed in Azure SQL and is now in SQL 2025) [31]. With it, you could call a local LLM's completion API or an OpenAI chat completion endpoint by constructing the JSON payload. However, managing multi-step conversations or large responses purely in T-SQL can be cumbersome. A more developer-friendly pattern is to use an external application or agent (see Section 5) to orchestrate calls to the database and the LLM. The key point is that **SQL Server can serve as the secure interface to your models** – whether they're on Azure or running on your own hardware. It decouples the model endpoint details from your application logic. For example, you might register a model named `LocalCodeAssistant` pointing to your own API (or to an Azure OpenAI deployment), and then your queries or app code simply references that model. If you later switch to a different model, you can update the `CREATE EXTERNAL MODEL` to the new endpoint, with no changes needed in the querying code [12]. This flexibility is great for experimenting with different LLMs or upgrading models.

In summary, **SQL 2025 can interface directly with local LLMs** by treating them as external services. It doesn't host the full model *inside* the SQL engine (that would be impractical for large models), but it provides a seamless bridge: you can invoke local AI with T-SQL commands, and get results back into queries. Microsoft explicitly designed it to work with popular AI endpoints *"anywhere from ground to cloud"*, including open-source models running on a laptop or on-prem server [32]. This means you can keep all data processing local: your code stays in your database, and your prompts and completions go only to a model running under your control (no cloud API calls).

For our RAG-based code assistant, you will typically use two kinds of models in tandem: - An **embedding model** (possibly a smaller model) to vectorize code and queries for semantic search. - A **generative model** (usually a larger LLM) to produce answers, explanations, or code changes using the retrieved context.

SQL Server 2025 supports integrating both. In a fully offline scenario, you might use an open embedding model like MiniLM or Instructor XL (via ONNX or local API) for vectors, and a larger model like Code Llama or StarCoder served locally for generation. We'll discuss model choices next.

4. Choosing Local LLMs for Code Comprehension and Refactoring

Selecting the right large language model is crucial for a code-focused AI assistant. We need models that are **knowledgeable in programming** (across multiple languages, including legacy ones) and that can run locally on available hardware. Fortunately, in the last couple of years, many high-quality open or community-release models have become available. Below are some recommendations:

- **Meta's Code Llama:** This is a family of code-specialized LLMs released by Meta AI in 2023, based on the LLaMA 2 model. Code Llama was trained on 500B tokens of code and code-related data and comes in sizes of 7B, 13B, and 34B parameters (and an uncited 70B research model) [33]. It has variants fine-tuned for Python and an *"Instruct"* variant tuned to follow natural language instructions [34]. Code Llama demonstrated state-of-the-art performance among open models on coding benchmarks – for example, the 34B model surpassed previous open models on tasks like HumanEval (solving >53% of coding problems correctly) [35]. In practical terms, Code Llama is strong at generating and completing code and explaining code in natural language. The **Instruct** version is preferred for our use-case since it's better at conversational Q&A and obeying prompts (e.g., "Explain what this function does" or "Refactor this code to improve performance"). These models can definitely handle C# and C++ (which were abundant in its training data), and likely have some exposure to VB.NET and SQL. VB6 code might be rarer in its training set, but the model can still attempt to read VB6 syntax since it's similar to VB.NET in structure. Running Code Llama locally requires significant memory – the 13B model can possibly run on a single high-end GPU (or on CPU with slower speed), while 34B may require multiple GPUs or quantization. But if you have the resources, Code Llama 34B-Instruct would be a top choice for an AI "engineer" due to its accuracy and context handling.

- **BigCode's StarCoder (and StarCoder2):** StarCoder is another leading open model for code, released by the BigCode research project. The original StarCoder is a 15.5B parameter model with about an 8K token context window [36]. StarCoder was trained on **The Stack**, a dataset of over 1 trillion tokens of source code from GitHub (permissively licensed) spanning **over 600 programming languages and markup formats** [37]. This broad training means StarCoder is adept in many languages – not just popular ones like Python or Java, but also niche and legacy languages that appear on GitHub. It

is very likely to have knowledge of T-SQL (SQL scripts), classic Visual Basic or VB6 code, and various .NET languages, given the breadth of The Stack dataset. StarCoder was further fine-tuned on Python and on instruction following, making it capable of conversational assistance. BigCode has also released **StarCoder2** in 2024 with sizes 3B, 7B, and 15B, trained on an even larger corpus (Stack v2, ~4× bigger) [37] [38] . Notably, StarCoder2-15B can outperform the original 15B model despite the smaller size, thanks to the massive training data [38] . For our scenario, StarCoder (or StarCoder2) 15B is a compelling option because it balances quality with relatively manageable size (15B parameters can run on a single 24 GB GPU or a couple of 16 GB GPUs with optimization). It has a broad multilingual coding ability – for example, StarCoder's training included languages from C++ and C# to SQL, PHP, even COBOL and likely VB, so it can interpret a variety of file types. This broad knowledge is akin to having an engineer who has seen code bases in many languages. StarCoder's 8K context window is also helpful: it can take in several code chunks or a long piece of code for analysis in one go.

- **Other open models:** Beyond Code Llama and StarCoder, there are other models you might consider:

- **Phind-CodeLlama 34B** – an enhanced fine-tune of Code Llama by the Phind team, which achieved even better results on coding tasks (Phind tuned the 34B model on an internal dataset, pushing HumanEval scores into the high 60% range) [39] . This model might offer a boost in coding prowess, though it's basically an improved Code Llama and similarly heavy to run.
- **WizardCoder** – this is an instruction-tuned 15B model (based on Code Llama or similar) known in the community for strong performance on coding tasks. It uses advanced fine-tuning techniques (like Evol-Instruct). A **WizardCoder 34B** also exists, which is powerful but requires more memory. These could be viable if you find community checkpoints and have the compute to run them.
- **Mistral 7B (instruct)** – Mistral AI released a high-quality 7B model in late 2023 that, while not primarily a code model, has very strong general performance (it outperforms older 13B models on many tasks) [40] . It has been fine-tuned in instruction-following versions. A 7B model can run on a modest machine (even on CPU or a single 8–12 GB GPU). While it won't match the code expertise of a 34B Code Llama, it *can* be useful for smaller tasks or as a fallback due to its speed. Mistral or similar small models might not deeply understand VB6 quirks, but they could handle simpler Q&A and quick summarization with fine-tuning.
- **Legacy models** like GPT-J-6B, GPT-NeoX-20B, or CodeGen (by Salesforce) were early open models for code. However, the newer ones mentioned above generally surpass them in capability. For instance, a 13B Code Llama or 15B StarCoder will be much better than GPT-J (6B) at code tasks. So you would likely focus on the latest generation of open models for best results.

When choosing a model, consider the trade-off between size and accuracy. Larger models (with more parameters or trained on more data) usually understand code better and produce more coherent improvements, but they need more hardware. Because we prioritize *local* operation, you may need to quantize models or use techniques like 4-bit or 8-bit running to fit them on smaller GPUs. There's an active community around running large models on consumer hardware (e.g., projects like LLaMA.cpp, text-generation-webui, etc.), which you can leverage.

Crucially, **test the models on your codebase's languages**. For example, try prompting each candidate model with a VB6 function and ask for an explanation, to see if it grasps VB syntax. You might find one model handles SQL stored procedures more gracefully, while another is better at C# – this could even lead to using specialized models per language if needed (though that complicates the system). In general, Code Llama and StarCoder are solid multi-language choices and have the advantage of community support and

continued improvements. Both are under permissive licenses for use (Code Llama has a community license allowing commercial use with some conditions; StarCoder's license is OpenRAIL which allows broadly internal use). By running them on-prem, you ensure privacy since your proprietary code never leaves your servers [41].

One more consideration is **embedding models vs. generative models**: Often, the model that's best for generating code/comments might not be the fastest for embedding large amounts of text. You might use a smaller, efficient model to generate embeddings for tens of thousands of code chunks (for example, MiniLM or SentenceTransformer models tuned for coding text), while reserving the heavy LLM for actual question answering and code generation. This two-model setup is common in RAG architectures: it improves performance without sacrificing the answer quality. SQL Server 2025 supports multiple external models, so you could register one model (or ONNX) for `AI_GENERATE_EMBEDDINGS` and another for handling free-form questions via `sp_invoke_external_rest_endpoint`.

In summary, **preferred local LLM options include**: - *Code Llama* (7B/13B/34B) – excellent coding capabilities, with the instruct variant for conversational use [35]. - *StarCoder* (15B) or *StarCoder2* – broad language coverage (600+ languages) ideal for a heterogeneous codebase [37]. - Fine-tuned derivatives like *Phind-CodeLlama* or *WizardCoder* – for potentially higher accuracy if you have the resources. - For embeddings, smaller models like *MiniLM* (as used in examples) or *all-MiniLM-L6-v2* can be employed to vectorize text quickly [26] [25].

All of these models can be run locally with the right setup, allowing you to keep the entire analysis pipeline offline.

5. From Insights to Action: Using AI to Improve the Code

Collecting code embeddings and retrieving relevant snippets is only half the battle – the goal is to have the AI not just *find* code, but also *understand and improve* it. This is where the generative capabilities of the LLM come in, turning retrieval results into useful insights or even code changes. We want the system to function like a seasoned engineer who can read through code and suggest refactorings, catch bugs, or outline design improvements. Here are strategies and tools for converting AI insights into actionable outputs:

- **Natural language Q&A for understanding:** Developers can ask the system questions about the codebase and get answers that synthesize information from various files. For example, *"How is data validation implemented in our order processing module?"* The RAG pipeline would:
- Take the question, generate an embedding for it, and perform a vector search on the `CodeChunks` table to find relevant code (e.g., code in that module, any validation logic) [11].
- Fetch the top-N chunks and construct a prompt for the LLM that includes those code snippets (or summaries of them) along with the developer's question.
- The local LLM then produces an answer, perhaps: "It looks like in *OrderProcessor.cs* we validate inputs by checking nulls and ranges, and for database fields we use stored procedure *usp_ValidateOrder* in SQL which ensures no negative quantities. Additionally, there's an XML schema validation for certain inputs (see *OrderSchema.xsd*)." The answer is derived from actual code, not just the model's training, which makes it accurate to your internal logic (and reduces hallucinations) [42].

This QA mode is straightforward and very useful for onboarding new developers or for quickly locating where certain logic is. It's effectively a **chatbot on your code**. Since SQL 2025 supports integration with

frameworks like LangChain and Semantic Kernel [43] [44], you can use those tools to simplify building a chat interface that uses the database as a retriever. For instance, the `langchain-sqlserver` connector can allow an LLM to query the SQL database for context [44], and Semantic Kernel can orchestrate the prompt assembly and model calls.

- **Code summarization and documentation:** Another immediate use-case is generating summaries or documentation for legacy code. For example, you can prompt the model with a retrieved code snippet and ask: "Explain what this function does and what its major steps are." The LLM can produce a concise description in plain English, effectively writing missing documentation. This can be done ad-hoc (one function at a time via a query in a tool), or you could even automate it repository-wide: e.g., iterate over all public methods and have the LLM generate XML doc comments or markdown docs. Because this would involve a lot of model calls, you'd likely script it in batches and use the local model to avoid API costs. The output can then be stored (maybe in a table or directly in code comments for review). The semantic search comes in handy when a function calls many others – the assistant could pull in relevant pieces of those to include in the explanation if needed.

- **Detecting issues and suggesting fixes:** You can turn the AI into a code reviewer. For instance, ask the assistant: "Analyze this code for potential null reference errors or resource leaks." Using prompt patterns, you can have the LLM list any problematic code patterns it sees and suggest corrections. Because the LLM has been trained on a lot of code and even some common flaws, it may catch things that static analysis might miss in context. However, combining the AI with traditional static analysis is wise: you could feed outputs of analyzers (like a list of warnings) into the prompt as well, making the LLM focus on those areas. The LLM's suggestions might be general ("check if `dbConnection` is null before using it"), but you can then drill down with RAG to find where exactly to apply changes.

- **Automated refactoring & code generation:** One of the ultimate goals is to have the AI not only point out improvements but actually *implement* them (or at least draft the changes). This is challenging but increasingly feasible with careful tool integration. Here's a possible pattern:

- **Generate a plan/design:** For a high-level request like "Modernize our data access layer in the VB.NET application," the assistant can first output a proposed approach: e.g., it might outline that the code uses old ADO, and suggest moving to Entity Framework or parameterized queries, etc. This is a non-trivial multi-step task, so the AI's role is to break it down (almost like a consultant). This plan can guide developers or feed into an automated workflow for each sub-task.
- **Code diff suggestions:** For more localized changes, you can prompt the LLM to produce a diff. For example, after identifying a chunk of code that needs change, you provide that code context and say "Refactor this to use a using-statement for disposing the object. Provide the diff." The model can then output something like a unified diff format:

```
- Dim conn As New SqlConnection(connStr)
- conn.Open()
- ' ... use conn ...
- conn.Close()
+ Using conn As New SqlConnection(connStr)
+    conn.Open()
```

```
+    ' ... use conn ...
+End Using
```

This textual diff can be parsed and applied to the codebase automatically. It's basically the AI writing a patch. Because the system is local, you could even have the AI agent directly write to a file (if you trust it enough and set proper permissions). More safely, you might have it output changes for review, then a script or developer approves and applies them.

- **Iterate and verify:** After applying changes, the AI (or separate test pipelines) can run unit tests to verify nothing broke. This is where an **agent** approach shines – an AI agent could in theory run tests, see failures, and adjust. This level of autonomous refactoring is cutting-edge and experimental, so in practice a human engineer would oversee it. But even without full autonomy, the AI can save time by generating the bulk of repetitive changes (like updating syntax or API usage across hundreds of files).

- **Create a pull request:** Once a set of changes are finalized, tools can package them into a PR. For instance, using Git automation, you can have a script that commits the changes on a new branch and even uses the LLM to generate a *commit message and PR description* summarizing what was done. Since the LLM has context of why changes were made (from the prompt), it can draft a rationale. This PR can then be reviewed by senior engineers as usual.

- **Architecture and tool support:** To accomplish the above, it's recommended to use an orchestration layer – essentially a controller outside of SQL that sequences these steps. Both **LangChain** and **Semantic Kernel** (open-source frameworks) can be extremely useful:

- *LangChain:* It provides abstractions to do RAG querying and to define custom "tools" that an LLM can use. For example, you can give the LLM a *tool* that queries the SQL database (to retrieve code) and a tool that executes shell commands (to run `git` or run tests). You could then prompt the LLM in an agent mode, where it decides: "I need to get relevant code from the DB" (uses the SQL tool), then "I should suggest a code edit" (uses its own reasoning to produce diff), then "I should run tests" (uses a command tool), etc. This is complex but demonstrates how an AI agent can automate coding tasks. LangChain has a SQL Database toolkit and can integrate with Python or other language runtimes to do such multi-step workflows.

- *Semantic Kernel:* Microsoft's Semantic Kernel is another option, which is well-suited for .NET developers. It's an SDK that lets you combine AI with **native code execution** and planning. SK has a connector for SQL Server and can work with Hugging Face or other local models [45]. One could implement "skills" in C# that do things like read a file, open a GitHub issue, or call an internal API. The LLM can be directed to invoke those skills as needed. Microsoft suggests using Semantic Kernel to build agents that *"answer questions and automate processes"* by calling existing code and tools [45]. In our scenario, that could mean an SK agent that, for example, on command "upgrade this project to .NET 6", will retrieve relevant project files, call a script to retarget frameworks, and use the LLM to adjust code syntax for the new version, etc., all orchestrated in one flow.

- **Human oversight and iterative refinement:** It's important to treat the AI's output as suggestions unless you have high confidence. Especially for critical enterprise code, you'll want a human in the loop to review any changes. The AI system can greatly accelerate understanding and even provide draft implementations, but a senior engineer (the real one) should verify that the changes make

sense, adhere to coding standards, and pass all tests. Over time, as trust builds, you might increase the automation level for certain low-risk tasks (like formatting or simple refactors). Always maintain good version control practices so that any AI-introduced changes can be tracked and reverted if needed.

By using the above approaches, you can transform retrievals into tangible improvements: - **Example:** The AI finds all places where an outdated encryption algorithm is used (via semantic search), then suggests a modern library replacement, and even generates the code changes to call the new API. The developers review these suggestions, maybe tweak a few, and accept them. The assistant then helps create a PR with all changes and a summary of what was done. This turns what could be a laborious manual code audit and update into a partially automated process guided by AI.

6. Architectural Recommendations for a Local "AI Engineer" System

Bringing everything together, here's an outline of an architecture that implements a local AI-assisted code engineer using SQL Server 2025:

- **Data Ingestion and Storage:** All source code is loaded into **SQL Server 2025** (on-premises). Use a **CodeFiles** table for raw file storage and a **CodeChunks** table for chunked segments of code, each with an embedding vector. Leverage `AI_GENERATE_CHUNKS` to preprocess text and `AI_GENERATE_EMBEDDINGS` to populate vectors [8] [1]. Build a DiskANN **vector index** on the embeddings for efficient similarity search [6]. Also create any needed relational links (e.g., map which chunks belong to which file, which file belongs to which application/module, etc., so you can filter or group results). Ensure this database is updated whenever the codebase changes (you might integrate it with your CI/CD or nightly jobs to re-index new code).

- **AI Model Hosting (Local):** Choose and deploy your **embedding model** and **LLM model** on local infrastructure:

- For embeddings, an example is running a small embedding model via **ONNX Runtime** inside SQL (as shown with all-MiniLM) [26] [25], or running a service like **Ollama** with an embedding model (Ollama can serve models like `nomic-embed` or `all-MiniLM` with an `/api/embed` endpoint) [46]. This model turns queries or any text into a 768-dim vector that matches your `VECTOR` size.

- For the main LLM, set up a **local API endpoint**. This could be:

    - Ollama with a larger model (e.g., `ollama pull codellama-13b` or any model you want to use) for text completion/chat.
    - Or a custom HTTP server using Hugging Face Transformers (for instance, using the HF **text-generation-inference** server or simply a Flask app that loads the model and serves requests).
    - Ensure this endpoint is HTTPS (use a reverse proxy + self-signed cert as needed) so that SQL can call it [23].
    - Register the model in SQL Server via `CREATE EXTERNAL MODEL`, giving it a name like `CodeAssistantModel`, with `LOCATION = 'https://yourhost:port/...'` and `API_FORMAT` appropriate to your server (could be "Ollama" or a custom format if using OpenAI-compatible API) [15] [19].

    - Test that you can call it with `sp_invoke_external_rest_endpoint` or similar – e.g., a simple prompt to echo back something – to verify connectivity.

- **Application/Orchestration Layer:** Build an **AI Orchestrator** that sits between the user (developer) and the backend (SQL + LLM). This could be a command-line tool, a web app, or an IDE extension. The orchestrator's responsibilities:

- **Interpret user requests:** For example, the user might type a question or select a piece of code and ask for improvement suggestions.
- **Retrieve relevant context:** The orchestrator formulates a SQL query (or uses a parameterized stored procedure) to perform the vector search in SQL Server. Thanks to the semantic index, it gets back the top relevant code chunks (and maybe some metadata like file names).
- **Compose the LLM prompt:** The orchestrator inserts those retrieved code snippets (perhaps with some formatting or truncation if needed) into a prompt template. For instance: "You are an expert software engineer. The user's question is: <question>. Here are relevant code excerpts: `<code chunk>` ... Based on this, provide an answer...". If the task is to generate code changes, the prompt might be structured to ask for a diff or for updated code.
- **Call the LLM model:** It sends this prompt to the local LLM endpoint (either directly or by invoking the external model via SQL). The model will process it and return a response.
- **Post-process the response:** The orchestrator may need to format the LLM's output for display. If it's just an answer, it can show it to the developer. If it's a code diff or multiple suggestions, it might highlight them or even apply them to a sandbox for preview.
- **Tool execution (if automated):** If operating in an agent mode, the orchestrator can parse the LLM's output for actions. For example, if the LLM says "I recommend changing function X in file Y", the orchestrator could automatically open that file (or use a version control API to create a branch and apply changes). This requires careful design to avoid unintended edits. A safer mode is *confirmation-based*: the tool could present a preview of changes and ask the user to confirm each or all.

- **Iterative interaction:** Ideally, the system allows back-and-forth. The developer might ask a follow-up question ("Why do you suggest that change?") or refine the request ("Show me the changes in a unified diff format"). The orchestrator would then incorporate that into the next query/prompt. Maintaining conversational state can be done within the app or by using the LLM's memory (by resending prior Q&A as context, if the model and context window allow).

- **Integration in developer workflow:** To maximize usefulness, integrate this AI system where developers work. For example:

- In **Visual Studio Code** or Visual Studio, you could create an extension that uses the orchestrator. The user could select code and trigger a command like "Ask AI" or "Refactor with AI" – behind the scenes, it would run the steps above. Microsoft is in fact adding Copilot-style features to their tools (the SQL 2025 release notes mention *"integration of MSSQL extension for VS Code with GitHub Copilot"*, which is a cloud solution) [47] . But you can implement a similar interface with your local models.
- Alternatively, a **chatbot interface** (web-based or even as a Teams/Slack bot) that developers can query about the codebase could be built. They ask questions in natural language and get answers with code references attached.

- A **dashboard** could be created to track AI suggestions and actions. For instance, it might list "10 possible improvements identified by AI in the last scan" with buttons to apply or dismiss them.

- **Security and privacy:** Since everything is local – the code database and the models – you maintain full control over the data. Nevertheless, treat the AI's access with the same security as a developer would have. The external model calls are effectively like a user with read (and possibly write) access to the data you feed it. Ensure the SQL queries used for retrieval only expose the necessary code content to the model. It's wise to avoid sending extremely sensitive data to the LLM unless needed (even locally, to minimize any logging risk). If your model runs on a separate machine or container, secure that channel (which is why TLS is enforced) [20]. Use database permissions to ensure only authorized processes or users can create/external models and call them [16].

- **Incremental improvement and learning:** Start with a narrow scope (e.g., Q&A and small refactor suggestions in a subset of the codebase) and gradually extend trust and capabilities as the team grows comfortable. Monitor the quality of AI output. You can even log the questions and answers (perhaps storing them in a history table) to analyze how often it's correct or useful. This feedback can guide if you need a different model or further fine-tuning.

By following this architecture, you essentially deploy a local AI co-engineer that knows your entire codebase (via the vector index in SQL) and can reason about it using advanced language models. All components – storage, search, and model inference – are running on machines you control, meeting the requirement of not sending data over the network. **SQL Server 2025 acts as the backbone** that not only stores the data but also provides the "brain" with efficient access to relevant knowledge [48]. The result is a system that can answer complex questions about a sprawling enterprise codebase and even assist in modifying it – a huge boost to developer productivity and code quality.

Conclusion

SQL Server 2025's AI integration features unlock new possibilities for understanding and improving complex codebases in-house. By ingesting your code into the database and using built-in semantic indexing, you create a private code search engine that goes far beyond simple text matching. Coupling that with local large language models gives you a powerful AI assistant who "reads" your code and offers insights or fixes on demand – all without exposing proprietary code to external services. We discussed how to set up the pieces: using SQL 2025's **vector storage and search** for Retrieval-Augmented Generation, leveraging **external model hooks** to connect to **local LLMs**, and selecting the right open-source models that are skilled in multiple programming languages. We also covered strategies to generate useful output (summaries, refactor suggestions, diffs) and even to apply changes through an orchestrated workflow.

This architecture, when implemented, functions like a diligent senior engineer who has the entire codebase in their head: you can ask any question and get an informed answer with source references, or request a code improvement and receive a plausible solution. It remains *"AI-assisted"* – meaning human developers stay in control, reviewing and guiding the changes – but it significantly augments their capabilities. Teams can expect faster onboarding (new devs can query the knowledge base instead of hunting through legacy code), quicker debugging and impact analysis (the AI can localize where a certain functionality lies), and help with modernizing code (by suggesting consistent improvements across the whole codebase). All of this can be done **locally**, ensuring compliance with security and privacy requirements.

In essence, SQL Server 2025 provides the foundation for building an AI co-pilot for your software development, directly within your existing data infrastructure [43]. By combining it with the right models and tools, you can achieve a near state-of-the-art RAG system that turns your own code into a source of

truth the AI can draw upon. This empowers your organization to intelligently analyze and evolve a complex enterprise codebase – keeping the knowledge in-house and the feedback loop tight. With careful implementation, the result is as if you've added a super-powered code analyst to the team, one who works 24/7 and helps your human engineers focus on creative and high-level design while routine understanding and heavy-lifting suggestions are handled by AI. The promise of an AI-assisted "senior engineer" working locally is within reach, enabled by the advancements in SQL Server 2025 and the flourishing ecosystem of open-source LLMs.

Sources:

- Microsoft SQL Server Blog – *Announcing SQL Server 2025 Preview: The AI-ready database* [49] [12] [43]
- RedmondMag – *SQL Server 2025 Brings AI-Powered Semantic Search* [13] [8]
- Microsoft Learn – *CREATE EXTERNAL MODEL (Transact-SQL)* [15] [19]
- Microsoft Learn – *AI_GENERATE_CHUNKS (Transact-SQL) [Preview]* [1]
- Microsoft Learn – *Vector search and indexes in SQL Database Engine* [50] [6]
- Cegal Tech Blog – *SQL Server 2025: 10 new features (AI integration)* [51] [4]
- E2E Networks Blog – *Top Open-Source LLMs for Coding* [35] [37]
- MSSQLTips – *SQL Server Semantic Search using New AI Features* [17] [23]
- Microsoft Learn – *Intelligent applications and AI – LangChain & Semantic Kernel* [44] [45]

---

[1] [2] AI_GENERATE_CHUNKS (Transact-SQL) - SQL Server | Microsoft Learn
https://learn.microsoft.com/en-us/sql/t-sql/functions/ai-generate-chunks-transact-sql?view=sql-server-ver17

[3] [12] [32] [43] [47] [49] Announcing SQL Server 2025 (preview): The AI-ready enterprise database from ground to cloud - Microsoft SQL Server Blog
https://www.microsoft.com/en-us/sql-server/blog/2025/05/19/announcing-sql-server-2025-preview-the-ai-ready-enterprise-database-from-ground-to-cloud/

[4] [48] [51] SQL Server 2025: 10 new features that can create value
https://www.cegal.com/en/resources/sql-server-2025-10-new-features-that-can-create-value

[5] [6] [7] [8] [9] [10] [13] [22] [31] [42] [50] SQL Server 2025 Brings AI-Powered Semantic Search to Local and Cloud Data -- Redmondmag.com
https://redmondmag.com/articles/2025/08/12/sql-server-2025-brings-ai-semantic-search-to-local-and-cloud-data.aspx

[11] From Snippets to Systems: Advanced Techniques for Repository-Aware Coding Assistants | by Colin Baird | Medium
https://medium.com/@colinbaird_51123/from-snippets-to-systems-advanced-techniques-for-repository-aware-coding-assistants-cf1a2086ab41

[14] [15] [16] [18] [19] [20] [24] [25] [26] [27] [28] [29] [30] CREATE EXTERNAL MODEL (Transact-SQL) - SQL Server | Microsoft Learn
https://learn.microsoft.com/en-us/sql/t-sql/statements/create-external-model-transact-sql?view=sql-server-ver17

[17] [21] [23] [46] SQL Server Semantic Search using New AI Features
https://www.mssqltips.com/sqlservertip/8285/sql-server-semantic-search-using-ai-features/

[33] [34] [35] [36] [37] [38] [39] [40] [41] Top Open-Source LLMs for Coding
https://www.e2enetworks.com/blog/top-8-open-source-llms-for-coding

44 45 Intelligent Applications and AI - SQL Server | Microsoft Learn
https://learn.microsoft.com/en-us/sql/sql-server/artificial-intelligence-intelligent-applications?view=sql-server-ver17