

Agentic RAG Chatbot — Architecture Plan

.NET Core · SQL Server / Postgres · ChatGPT · Self-hosted vector search

August 2026

1. Goal

An agentic RAG chatbot, private on our own servers, with only the LLM call going to an external provider (ChatGPT / Azure OpenAI). The agent decides, per user message, whether to pull information from our relational database (existing product data) or from a vector database (uploaded PDFs and documents), and can use both in the same turn. Every request, response, tool call, and cost is logged for observability.

2. Recommended stack

Layer	Choice	Why
LLM	ChatGPT via Azure OpenAI (not the public OpenAI API)	Data-processing agreement, no training on our data, regional hosting — the right fit for a “private except the LLM” requirement
Orchestrator	Semantic Kernel (.NET)	Native .NET, built-in function-calling / agent loop that maps directly to tool routing
Vector DB	Postgres + pgvector	One engine to operate for new products, SQL-familiar, HNSW indexing is good enough for most RAG workloads
Structured DB	Existing SQL Server for current products, Postgres for new ones	No migration needed on legacy systems; run side by side
Observability	OpenTelemetry (.NET SDK) + Langfuse (self-hostable)	OTel gives distributed tracing across every layer; Langfuse adds LLM-specific views — prompts, tokens, cost, latency

3. System architecture

Client → .NET Core API (gateway, request logging) → Orchestrator agent (Semantic Kernel) → either:

- an **external call** to the LLM (the only component that leaves our infrastructure), or
- an internal **tool call** — the SQL query tool (existing SQL Server / Postgres) or the vector search tool (Postgres + pgvector) — after which the result goes back to the LLM to be woven into the final answer

Observability instrumentation wraps the API, the orchestrator, and every tool call, all correlated under one trace ID per user turn.

4. Ingestion pipeline (documents → vector DB)

1. Upload endpoint accepts PDF/DOCX, stores the raw file in blob storage
2. A background worker (.NET hosted service, queued via RabbitMQ / Azure Service Bus) picks it up
3. Text extraction (PdfPig / iText for PDFs, DocumentFormat.OpenXml for DOCX)
4. Chunking — 500–1000 tokens per chunk, ~10–15% overlap, split on semantic boundaries where possible
5. Metadata tagging per chunk — source document, page number, owner/ department, access-control tags
6. Embedding — see the privacy decision in section 6
7. Store vector + chunk text + metadata in Postgres/pgvector, indexed with **HNSW**
8. Log ingestion status and any extraction failures to observability

5. Chat request flow

1. User sends a message; the API receives it and starts a trace
2. The orchestrator sends the message plus the registered tool schemas to the LLM
3. The LLM decides: call a tool, call multiple tools, or answer directly — based entirely on how clearly each tool is described (see section 7)
4. If a tool is called, the orchestrator executes it under the safety constraints in section 8, and returns the result to the LLM
5. The LLM either calls another tool or produces a final answer; steps 3–4 repeat until it has enough information
6. The final answer is returned to the user; the full trace, every tool call, and token cost are logged

6. How the vector search tool works

- **Schema:** a `documents` table (title, source, owner, access tags) and a `chunks` table (chunk text, embedding vector, page number, access tags)
- **Query flow:** embed the user's (rewritten, standalone) query → pre-filter by access-control tags in SQL, not after retrieval → run vector similarity and keyword search in parallel (hybrid search) → merge and rerank with a small cross-encoder → return top 3-5 chunks with source and page number for citation
- **Embedding model — privacy decision:** sending document text to an embedding API is the same exposure as sending it to the chat model. Two options: accept it under the same Azure OpenAI data agreement as the chat model, or self-host an open-source embedding model (e.g. `bge-m3`) behind a small inference service so raw text and vectors never leave our network. This must be decided before ingestion is built, since it determines whether a Python inference sidecar is needed.

7. How the orchestrator decides DB vs. vector DB

This is standard LLM **function calling**, not a hand-written classifier. Each tool is registered with a name, a natural-language description, and a typed parameter schema; the LLM reads the user's question against those descriptions and decides which tool (if any) applies.

The tool description is the actual decision boundary — vague descriptions cause misrouting. Concretely:

- SQL tool description: *"Retrieves specific business records — order status, customer info, inventory, pricing — for questions referencing a specific ID, order number, or account."*
- Vector tool description: *"Searches uploaded manuals, policies, and documentation for conceptual or how-to questions — not for looking up a specific transaction."*

Questions needing both (e.g. an order-specific refund policy) trigger sequential or parallel tool calls in the same turn — the loop in section 5 keeps running until the model has what it needs. Add a few-shot routing example in the system prompt, let the model ask a clarifying question when genuinely ambiguous, and cap tool calls per turn (4-5) so a confused model can't loop indefinitely.

8. Executing tool calls safely

The LLM decides *which* tool and *what parameters* — it never decides *how* the query runs. Concretely:

1. **Validate the call** — reject malformed tool names or parameter types before executing anything
2. **No raw SQL or raw filters from the LLM** — the SQL tool only accepts typed parameters mapped to a fixed stored procedure or view; the vector tool passes values into a fixed query builder
3. **Least-privilege execution** — a read-only DB role scoped to specific views/procs; vector queries always carry the requesting user's access-control filter, applied by our code, never overridable by the model
4. **Bounded execution** — per-call timeout, row/top-k limits, capped tool calls per turn
5. **Sanitize retrieved content** — uploaded documents are untrusted text and could contain a prompt-injection attempt; the system prompt keeps its own instructions authoritative over anything returned from a tool
6. **Log everything** — tool name, parameters, latency, row/chunk count, correlated to the turn's trace ID

9. How the SQL tool searches — and what the LLM knows about our schema

Two approaches, with very different risk profiles:

	Fixed tool-per-question (recommended)	Constrained text-to-SQL
What the LLM knows	Only tool names + business parameters (orderId, customerId)	A curated, deliberately limited schema description
Schema exposure	None — no table/column names ever reach the model	Partial — only what we choose to document
How search happens	Our code maps the call to a fixed stored proc/query	LLM generates SQL, parsed and validated before running
Safety	High	Lower — needs a SQL validator, read-only replica, caps
Flexibility	Limited to questions we've built a tool for	Handles ad hoc questions
Maintenance	Add a tool + proc per new question type	Keep the schema description current

Recommendation: start with the fixed-tool approach entirely. It covers the large majority of "search our DB" needs without exposing any schema to the model or opening SQL-injection surface. If lookups are needed (e.g. resolving a customer name to an ID before calling another tool), add small resolver tools — the orchestrator chains them automatically. Only consider constrained text-to-SQL later, for a narrow analytics use case, run against a read-only replica, if fixed tools genuinely can't cover a recurring need.

10. Observability — what to capture

- One trace ID per user turn, propagated through API → orchestrator → tools → LLM
- Per LLM call: prompt, response, model, token counts, cost, latency
- Per tool call: tool name, input parameters, execution time, rows/chunks returned
- Session view: full conversation, total cost, which tools fired each turn

11. Open items worth deciding early

- **Caching** — Redis in front of embeddings and frequent SQL lookups
 - **Reranking** — cross-encoder rerank step on vector search results
 - **Access control** — document-level tags enforced at query time, not relied on the model to withhold
 - **Evaluation** — RAGAS or similar to score retrieval relevance before/after changes to chunking or embeddings
 - **Fallback** — defined behavior when Azure OpenAI is rate-limited or down
-

Concerns raised during planning, and how they're addressed

“Everything should be private except the LLM — does that include embeddings?” No — by default, embedding calls to an API provider expose the same document text as a chat call would. Section 6 makes this an explicit decision: either accept it under the same data agreement as the chat model, or self-host an open-source embedding model so nothing but the final composed prompt ever leaves our network.

“I already have a SQL DB — what do I actually add for the vector DB?” A second, independent Postgres instance with the pgvector extension, running alongside SQL Server. No migration, no shared schema — Postgres only holds document chunks and embeddings; SQL Server stays the source of truth for product data. The .NET app gets a second connection (Npgsql + a separate DbContext), and the orchestrator's two tools each point at their own database. (Note: SQL Server 2025 added native vector support, which is a valid alternative if adding a second database engine is undesirable — but given the plan to standardize new products on Postgres, running it side by side is the more consistent path.)

“How does the orchestrator actually work, and how does the LLM decide DB vs. vector DB?” The orchestrator is our own .NET/Semantic Kernel code managing conversation state and a tool registry — it is not a separate AI. The routing decision is made entirely by the LLM via function calling, based on how each tool is described (section 7). Good, specific tool descriptions are what make routing reliable; vague ones cause misrouting.

“How does the orchestrator execute a requested tool call safely?” A six-step pipeline: validate the call, forbid raw SQL/filters from the LLM, execute under a least-privilege identity, bound execution with timeouts and row caps, sanitize retrieved content against prompt injection, and log everything to the trace (section 8).

“How does it search the DB — by a query? How does it know the tables and columns?” By design, it mostly doesn't need to. The recommended approach (section 9) keeps the LLM working with business-level tool parameters, not schema — table and column names never reach the model. A more flexible constrained-text-to-SQL approach is available for later if genuinely needed, but it requires exposing a curated schema subset and adding a SQL validator and read-only replica — meaningfully more engineering and risk for flexibility that most questions won't need.