ChatGPT

Knowledge Graph RAG System Architecture for Semiconductor Documents

Introduction and Context

Building a retrieval-augmented generation (RAG) system with an integrated knowledge graph requires a robust architecture that can handle unstructured technical documents and structured domain knowledge. In our case, we aim to ingest semiconductor design documents (e.g. JEDEC standards, specifications, internal PPT/DOC files), extract their content into a structured form, and provide an **AI chatbot interface** for engineers to query this knowledge. The solution must be **on-premises** for confidentiality, serve ~200–300 internal users, and handle on the order of thousands of queries per day. The key challenge is to combine **unstructured text** (hundreds of pages of technical PDFs with figures, tables, etc.) with a **structured knowledge graph (KG)** of domain concepts, so that the chatbot can accurately answer complex technical questions with up-to-date and trustworthy information [1] [2]. In short, we need an end-to-end pipeline: **document ingestion → information extraction → knowledge graph construction → semantic indexing → Q&A chatbot.**

Requirements & Constraints: We are dealing with highly technical and sensitive data, so **accuracy, security, and explainability** are paramount. Documents may be ~600 pages each with complex formatting, which means the extraction process must reliably capture text, tables, and references (figures might have to be handled via captions or cross-references). The system should support both *broad corpus searches* (e.g. "Find all mentions of feature X across standards") and *targeted queries* (e.g. "What does parameter Y mean in standard Z?"). Given the data is confidential, all components (databases, AI models) should be **self-hosted on-prem**. There is no strict tech stack mandate, but any proposed solution must justify the use of newer technologies (like NoSQL databases or graph databases) to more traditionally-minded architects. We also anticipate relatively **low data velocity** – new documents (e.g. updated standards) arrive periodically, not in a high-volume stream – allowing for batch processing and manual curation steps (like ontology updates by domain experts). With this context in mind, we can now outline the architecture and discuss potential pain points, failure modes, and design decisions.

High-Level Architecture Overview

System Architecture: At a high level, the solution comprises distinct components for document processing, knowledge graph management, vector-based semantic search, and the Q&A chatbot interface. The workflow can be visualized as a pipeline starting from raw documents and ending with an answer to a user's query. **Figure 1** below illustrates the end-to-end flow, from ingesting unstructured files to delivering answers via the RAG chatbot.

Figure 1: End-to-end pipeline for transforming unstructured documents into a queryable knowledge graph and supporting a RAG chatbot. Documents are ingested and split into chunks, embeddings are generated for semantic search, and entities/relations are extracted to populate the knowledge graph [3] [4].

As shown in Figure 1, the architecture can be broken into **two primary phases**: (1) *Knowledge Base Construction* – processing documents into a unified knowledge repository (combining a graph database and a vector index), and (2) *Query Handling* – using that repository to answer user questions via an LLM-based chatbot. Below, we discuss each component in detail, along with the design decisions (e.g. database choices, communication formats) and potential pitfalls.

1. Document Ingestion & Extraction Pipeline

Purpose: This component ingests raw documents (PDFs, Word, PPT, etc.) and extracts structured information as an intermediate JSON graph or similar format. The goal is to pull out text content, and where possible, identify key entities and their relationships in the text.

Process: Documents are fed into an **Extraction Pipeline** which may involve several NLP steps: text parsing, cleaning, **chunking** (splitting large texts into smaller sections), Named Entity Recognition (NER), and relationship extraction. Using a pipeline ensures unstructured text is systematically transformed into data we can work with [5] [6]. For example, we might split a 600-page PDF into paragraph or section chunks and run NER to find technical terms (like component names, parameters, units) and relation extraction to detect statements like "*X is part of Y*" or "*X depends on Y*" in the text.

- **Chunking:** We break documents into chunks (e.g. per section or a few hundred words) to manage large files. This not only makes downstream processing (embeddings, LLM context) easier, but also helps preserve locality of information. Each chunk can be tagged with metadata (document ID, section heading, page number) for traceability [7] [8]. Chunking is crucial to ensure that embeddings capture specific meaning and that search results map to specific document parts (too large chunks become vague) [9].

- **Entity & Relationship Extraction:** Using NLP techniques, we extract **entities** (domain-specific noun phrases like signal names, register fields, standard names, etc.) and link them to form **triples** or graph fragments. This can be done via rule-based parsing of known patterns (for example, spec documents often have tables of parameters and definitions) and/or ML models. We might leverage LLMs for this step – e.g. prompting a model to read a chunk and output a JSON of entities/relations – but this must be done with care due to potential errors. In practice, one might use a combination of domain-specific NER models and heuristic rules. **No model is 100% accurate**, so expect some errors that domain experts may need to correct [10]. For instance, NER might miss an acronym or mis-label a term, so the pipeline should log uncertainties or allow human-in-the-loop review.

- **Interim JSON Graph:** The output of extraction can be a JSON structure capturing the discovered entities and relationships from each document (hence "JSON graph object"). For example, if a JEDEC spec defines a memory module with certain timing parameters, the JSON might list the module as an entity node and each parameter as a related node with a "HAS_PARAMETER" relationship, including values or definitions as properties. This intermediate representation is basically a mini knowledge graph extracted from the document. We will store these intermediate results in a **raw data store** (more on this shortly) so that they can be reviewed and later transformed into the final knowledge graph.

Storage of Raw Extracted Data: It's wise to keep the raw extracted information before we formalize it into the knowledge graph. This could be a simple NoSQL document database where each document's parsed

JSON is stored, or even a set of JSON files in a file system. A **document-oriented database** is suitable here because the data is semi-structured (each document might produce a different schema of JSON) and we may want to query or update parts of it during curation [11] [12]. If our team is wary of NoSQL, an alternative is to use a relational database with JSON column support (e.g. PostgreSQL JSONB) – this can store the JSON blobs and even allow some querying. However, a document DB (like MongoDB or Couchbase) offers a more natural fit for flexible schemas and nested data, which is exactly our use case (each document's info might be nested by sections, etc.) [12] [13]. Given our volume is not massive, even a small MongoDB instance or PostgreSQL with JSON could handle it – the key is **flexibility** in storing whatever the extractor outputs without forcing it into rigid tables.

Potential Pain Points in Extraction: This stage is often where many issues can arise: parsing failures (e.g. if PDFs have unusual encoding or if a PPT has non-text diagrams), mis-identified entities, or relationships that don't map cleanly to the ontology. We should plan for some **error handling and fallbacks**: for example, if a parser fails on a figure or table, we might at least capture an image or figure caption reference rather than dropping it entirely. Because no extraction will be perfect, we will rely on **domain experts** to review and correct the output as needed. The pipeline can generate logs of low-confidence extractions for manual review. It's also important to maintain a link from each extracted piece back to its source document/page, so the chatbot can ultimately refer to the source and we preserve traceability (and user trust).

Communication between sub-components here can be relatively simple – since this is likely a batch or triggered process, we can have the extraction scripts output JSON and then call the next component. **Using JSON as the interchange format** is practical (it's human-readable, and our graph database or transformation scripts can consume it easily). We don't necessarily need a complex message bus unless we want to decouple processes; a straightforward approach is fine given the moderate processing volume. In summary, the ingestion pipeline **reads files → produces JSON data** (text chunks, entities, relations) and stores it in a raw database, ready for the knowledge graph construction.

2. Ontology & Knowledge Graph Construction

Ontology Design: A critical step (often underestimated) is defining the **ontology** – the schema or formal structure of our knowledge graph. The ontology specifies what entity types (node labels) we will have (e.g. *StandardDocument*, *Section*, *SpecificationItem*, *Concept*, *Parameter*, etc.) and what relationship types connect them (e.g. *HAS_SECTION*, *DEFINES*, *IS_PART_OF*, *RELATES_TO*, *VERSION_OF*). It may also include attributes/properties on those entities (for instance, a *Document* node might have properties like title, version, date; a *Parameter* node might have a data type or default value). Essentially, the ontology is the **"blueprint"** that ensures the knowledge graph isn't just a tangle of data, but a structured, consistent representation of the domain [14]. A well-designed ontology aligns with the questions we expect to answer. For example, if engineers often ask about differences between standard versions, we might include a relationship like *UPDATED_IN_VERSION* or a property for version numbers to enable such queries.

Since domain experts are involved, they will manually curate this ontology. Likely, they will enumerate key entity types from the documents (e.g. MemoryStandard, Register, Pin, TimingParameter) and relationships (e.g. *compliesWith*, *hasPin*, *timingSpecOf*). We should leverage their knowledge to make the ontology comprehensive yet not overly complex. **Consistency and clarity** are important – the ontology gives unique definitions to concepts so that, say, "DDR5" is recognized as the same entity across documents and is a subtype of "MemoryStandard" as defined by the ontology. This avoids ambiguity and data fragmentation in

the KG [15]. Ontologies also facilitate interoperability and integration: if later we integrate another data source, having a clear ontology will make mapping new data into the graph easier and less error-prone [16].

One approach is to use existing standards for ontology representation (like OWL/RDFS if we were going Semantic Web route), but we can also keep it informal and directly implement it in the graph database schema (i.e. define node labels and relationship types in code or config). Given this is an internal project, a pragmatic approach is fine – we don't necessarily need a full formal OWL ontology unless we see value in tooling (Protégé, reasoners, etc.). The key is that the ontology is documented and agreed upon by stakeholders. **Ontology evolution** should also be considered: as we learn from initial data or new document types appear, we may need to introduce new entity types or relationships. Our architecture should allow updating the ontology (which might involve data migration or re-tagging some nodes). It's worth noting that **multiple ontologies or taxonomy sources** might exist (JEDEC might have its own terminology). If we plan to incorporate any external ontology, we must align or merge them carefully to avoid semantic conflicts [17] – but initially, focusing on a single, custom ontology for our domain is likely easiest.

Graph Database Selection: To store and query the knowledge graph, a **graph database** is the natural choice. A graph DB is purpose-built to store entities (nodes) and relationships (edges) and query them efficiently, especially for multi-hop connections or patterns [13]. For our use case, a **property graph database** like Neo4j is a strong option – it's mature, supports ACID transactions (important for consistency when loading data), and has a flexible query language (Cypher) that will allow our engineers to query relationships easily. Neo4j also can be deployed on-prem and has tooling for data import and visualization. Another option could be an open-source graph DB like **JanusGraph** (which uses a distributed backend like Cassandra or Elasticsearch) or **ArangoDB** (multi-model including graph). However, introducing too many new technologies can be daunting – Neo4j is relatively self-contained and widely used in enterprise knowledge graphs. Importantly, **graph DBs are a subset of NoSQL** (they are non-relational), meaning we will need to justify this choice to any SQL-preferring colleagues. The justification is that representing and querying highly interlinked data (like "find all parameters defined in standard X that relate to feature Y") is *extremely cumbersome in a relational model*, whereas a graph DB can do it with straightforward path queries. It's certainly possible to emulate a graph in SQL (e.g. with tables for Nodes and Edges and recursive SQL queries), but performance and maintainability suffer beyond a certain complexity [18] [19]. By using a graph DB, we get out-of-the-box graph traversal algorithms, indexing on relationships, and simplified code. Moreover, **explainability** improves – we can easily trace why an answer was given by following relationships, something a purely vector approach lacks [20].

Graph Construction Pipeline: With the ontology defined and raw extracted data available, the next step is to map the JSON extraction output into the graph database. This can be seen as an ETL (Extract-Transform-Load) process:

- **Transform & Map:** For each document's JSON output, transform it according to the ontology. For example, if the JSON says DocumentA has a section "Electrical Specs" which mentions Parameter X = 1.2V, then we would: create a node for DocumentA (label *StandardDocument*), a node for "Electrical Specs" (label *Section*), a node for the parameter (label *Parameter* with name and value property), then create relationships: Section -*PART_OF*→ Document, Parameter -*DEFINED_IN*→ Section, etc. This transformation might be implemented as a script or using graph database batch import tools. Domain experts might need to define rules or mappings for complex cases (e.g. if something in text implies a relationship that isn't explicitly stated but should be in the KG).

- **Data Integration & Deduplication:** As we load data, we must ensure we're not duplicating entities. For instance, if "DDR5" appears in multiple documents, the ontology likely says DDR5 is an entity of type *MemoryStandard*. We should have one node representing DDR5 and link all relevant info to it, rather than separate DDR5 nodes per document. This may require checking if an entity already exists in the graph (by a key or name) before creating a new one. Using unique identifiers (perhaps the ontology can assign unique IDs for well-known standards or we use names) helps. Another consideration: if multiple documents have overlapping info, we might merge or annotate accordingly. The knowledge graph can be thought of as a **global integrated knowledge** extracted from all documents, so it should merge data where appropriate rather than silo by document (though we always retain links back to documents).

- **Graph Enrichment:** In some cases, after initial loading, we might enrich the graph with additional relationships not directly extracted. For example, we could add an ontology-driven relationship: if DocumentA is a newer version of DocumentB (maybe identified by naming convention), we create a *NEXT_VERSION* link. Or a domain expert might add missing links manually if the automated extraction missed them. This step ensures the graph is **complete and correct** in domain terms.

- **Storing Unstructured Context:** Note that our KG can also store some unstructured information as properties. For instance, we might store a definition text as a property on a node (e.g. Parameter X node has a property "definition_text" with the sentence defining it). The Neo4j example we looked at showed storing text embeddings on nodes [21], which is another possibility (more on embeddings in the next section). The point is the graph can hold both the structured links *and* relevant text snippets or metadata, making it a one-stop knowledge base [22] [23].

Graph Query Capabilities: Once built, the knowledge graph will allow us to answer certain queries directly via graph traversal. For example, "What are the sub-sections under 'Electrical Specs' in DDR5 spec?" or "List all parameters defined in JEDEC Standard XYZ" can be answered by Cypher or GraphQL queries on the KG. This is complementary to the vector search. In fact, one of the strengths of having a KG is the ability to handle **multi-hop or aggregation questions** by reasoning over relationships rather than relying on pure text search [24] [25]. We'll discuss in the query workflow how we might combine graph and vector search, but it's worth noting here: the KG makes our system far more powerful for complex questions that involve linking information across the corpus.

Pitfalls & Mitigation in KG Building: A potential failure point is if the extracted data doesn't fit the ontology well. This could manifest as a lot of "miscellaneous" nodes or relationships that don't map to any defined type (i.e. the ontology was incomplete). To mitigate, we must iterate on ontology design – likely after ingesting a sample of documents, refine the ontology to cover those cases. Another issue is **data quality**: the extraction might have incorrect links (false relationships) leading to a noisy or even misleading KG. This is dangerous for query answers. Our approach should include a **validation step** – e.g. have the SMEs spot-check the graph for obvious errors. We could even run some sanity queries (like ensure no parameters with obviously wrong values, or check that every Section has a parent Document, etc.). Finally, performance could be a concern if the graph grows huge, but given our scope (a few hundred large docs at most), a single graph database instance can likely handle it. Neo4j, for example, can comfortably manage millions of nodes/edges on decent hardware; our scale might be in the low millions of nodes (if each 600-page doc yields tens of thousands of triples, for say 10–20 such docs, it's manageable).

3. Embeddings and Vector Database for Semantic Search

Why a Vector Index: While the knowledge graph provides structured query capabilities, a lot of user questions will be answered by retrieving relevant passages from the documents – especially when a question is nuanced or requires verbatim information (like "According to the DDR4 spec, what is the maximum voltage?"). For this we incorporate a **vector database** to enable semantic search on document text. By embedding chunks of text into high-dimensional vectors (using an ML model), we can find text pieces semantically similar to a user's query, even if they don't share keywords. This addresses the scenario where an engineer asks a question in natural language that might not exactly match the wording in the documents. The vector DB will help pull the most relevant pieces of text to feed into the LLM, grounding its answer [26].

Embedding Generation: As part of the document processing (after chunking, possibly in parallel with KG extraction), we generate an **embedding vector for each text chunk**. This involves using an embedding model (for example, a SentenceTransformer or a domain-specific transformer) to produce, say, a 384-dimensional or 768-dimensional numeric vector for the chunk [27]. We should choose an embedding model that suits our domain: if we have access to a pretrained model on technical text, that's ideal; otherwise, something like *sentence-transformers/all-MiniLM-L6* or similar can work for technical Q&A. Since we cannot use external API calls (data is confidential), we'll likely run this embedding model on-prem (which is feasible with libraries like HuggingFace Transformers). The resulting vector represents the chunk's meaning in vector space.

Vector Database Choice: The vectors need to be stored in a specialized index for similarity search. There are a few paths here:

- A dedicated **Vector Database** like Milvus or Weaviate (both open-source and support on-prem installation). These are built to handle large volumes of vectors with efficient similarity search (usually using HNSW or IVF indices under the hood). For moderate scales (tens of thousands to millions of vectors), they offer fast cosine similarity or inner product queries. They also often support metadata filtering (e.g. "only search among chunks from Document X").

- Using an extension in a traditional DB: e.g. PostgreSQL with the pgVector extension can store and query vectors. This could be a compromise if the team prefers sticking to a relational system. However, performance might not match a system purpose-built for vectors, especially as data grows or if we need approximate nearest neighbor search for speed.

- **Graph DB as Vector Store:** Interestingly, some graph databases (like Neo4j 5.5+) have vector indexing capabilities [21] [28]. In fact, we could consider storing the embedding as a property on each *Chunk* node in the graph and use Neo4j's index to perform similarity search directly in the graph. Neo4j's latest versions support an algorithm (HNSW) for similarity on vector properties [29]. This approach means we might not need a separate vector DB at all – the graph DB would double as the vector search engine for text chunks. The Neo4j Developer Blog even demonstrates storing embeddings on nodes and querying them similarly to a vector DB [21]. The trade-off is that we'd be pushing the graph DB to do double-duty; we'd need to ensure it's configured for this and test performance. If the volume is manageable, it could simplify the architecture (one less database). On the other hand, a specialized vector DB might offer more flexibility (e.g. vector-specific optimization, easy scaling, and perhaps easier integration with future models).

Given that *vector search is a core part of RAG*, I lean towards using a well-supported vector search solution. **Milvus** is an attractive choice: it's open-source, designed for high-dimensional data, and can be deployed on-prem. **Weaviate** is another option that even allows hybrid searches (combining vector + keyword). Since our user count and data size aren't massive, even a single-node Milvus or a simpler FAISS index could suffice. But a proper DB gives us ability to update (e.g. if we add documents, we can upsert vectors) and query easily via API.

Storing and Linking Vectors: Each vector entry should be associated with an identifier that links back to the source chunk (and hence to the document and/or KG node). A typical setup is to use a unique key like *documentID_chunkID* as the vector's key, and store metadata like the document name or section. This way, when we retrieve similar vectors for a query, we not only get the chunk text but know which document (and section) it came from. In fact, this linking is how the system can present an answer and cite the source. We will maintain consistency between the vector DB and the knowledge graph: e.g. every chunk node in the KG could have an ID that matches the vector's ID, enabling cross-reference. If we use Neo4j's integrated vector index, this is handled inherently (because the node's properties are used for search).

Index Building: Building the vector index is straightforward: after chunking and embedding generation, bulk insert the vectors into the vector DB (or add them to graph index). This is mostly a one-time (or infrequent) operation since documents don't arrive too often. We should plan for re-indexing if the embedding model is updated or if chunks change. Also, if an ontology change leads to re-chunking or something, we'd regenerate those embeddings. These operations can be done offline (not affecting the running chatbot) and then hot-swapped or updated.

Potential Issues in Vector Search: One known issue with pure semantic search is that it can retrieve something that is semantically related but contextually irrelevant to the specific question – this is sometimes called *vector noise* or *embedding drift*. For example, a query about "power consumption in DDR4" might accidentally retrieve a chunk about "power management in a different context" because of some similarity in phrasing. This irrelevant chunk, if passed to the LLM, could confuse it or **poison the context** [30]. To mitigate this, we can do a couple of things: (a) retrieve not just by vector similarity but also apply some filters (e.g. same document if the question is within a doc context, or ensure certain keywords overlap), essentially a **hybrid search** combining semantic and lexical filtering [31]. (b) retrieve multiple candidates and have the LLM or a re-ranker model pick the truly relevant ones. Another approach (c) is use the knowledge graph to validate or filter vector hits – for instance, if the query mentions a specific standard, we can constrain the vector search to chunks from that standard document only (or at least rank them higher). This kind of integration is possible because our KG knows which chunks belong to which document and what that document is about.

We will also consider **explainability**: With vector search alone, it's not obvious why a chunk was retrieved (beyond "the model thought it was similar"). By tying chunks to the KG, we can sometimes explain results like "this section was pulled because it's about the concept you asked (which the KG links to your query concept)". This isn't directly shown to end users typically, but helps in debugging and refining the system. It's noted that one disadvantage of vector-only approaches is lack of explainability for odd results [20] – our use of a KG alongside can alleviate that.

Data Storage Decision – Vector DB vs NoSQL: Some colleagues might question why not use an existing relational DB for embeddings. The reason is largely performance and convenience. A vector is a 300+ dimensional numerical array; doing nearest-neighbor search on those using SQL alone is not efficient

unless we add specialized extensions. NoSQL in this context (a vector DB or a graph DB) is **designed for this kind of semi-structured data and query**. Since we are already embracing a graph DB (Neo4j) for the KG, adding one more specialized store is not unreasonable. However, to minimize the tech stack, one could attempt to consolidate: for example, ArangoDB can store documents and do vectors and graphs in one system, or Neo4j alone (with new features) could do both graph and vector. These are architectural trade-offs – using separate components (each best at what they do) vs. using a multi-purpose system for simplicity. My recommendation is to use a **vector database** because it's likely to give the best retrieval performance and we can always justify it by the importance of fast semantic search for user experience. We will, however, keep the integration tight: the vector DB doesn't stand alone; it's essentially an index to our content that works in concert with the knowledge graph.

4. RAG Agent Chatbot: Query Processing Workflow

This is the heart of the user-facing side. Let's walk through what happens when an engineer poses a question to our system, and highlight the architecture's support for that:

1. **User Query Input:** A user asks a question via the chatbot UI (for example: *"What does the JEDEC DDR4 spec say about refresh timing intervals?"*). The query is received by our **RAG Agent**, which is an application (possibly built with a framework like LangChain, or custom logic) that orchestrates the retrieval and generation steps.

2. **Query Analysis (Optional):** We may do some quick analysis on the query to decide how to retrieve information. For instance, we might detect if specific entities are mentioned (like "DDR4" which is a standard, or "refresh timing" which might correspond to a known parameter). If we detect known entity names, we could directly query the knowledge graph for those. In our example, "DDR4" would map to a *MemoryStandard* node for DDR4 in the KG, and "refresh timing interval" might map to a *Parameter* or concept if present. This kind of entity linking can help focus the search.

3. **Information Retrieval:** This is the retrieval phase of RAG. We leverage both the **Vector DB** and the **Knowledge Graph** (hybrid approach) for maximum effectiveness:

4. **Vector Retrieval:** We take the user's query (or a refined version of it) and compute its embedding using the same model as the chunks. Then we query the vector database for the *k* most similar chunks of text [32] [29]. These chunks are likely to contain content related to the question. For example, the vector search might return a chunk from the DDR4 spec text that describes the refresh timing, and perhaps another chunk from a DDR3 spec if similar wording exists (even if the query didn't explicitly say DDR3).
5. **Knowledge Graph Retrieval:** In parallel, or as a second step, we query the graph database. If our query was recognized to involve certain entities (like *DDR4* and *refresh timing*), we can run a Cypher query to pull structured info: e.g. find the node for "refresh timing interval" in the context of DDR4 and get its relationships or values. Even if we didn't do entity detection upfront, we could use the results from vector search to identify entities (e.g. if a retrieved chunk is from Section "Refresh" of DDR4, we now know to look at that part of the KG).

We might use a **prompt-to-query** approach: i.e. have the system or even the LLM translate the natural language question into a graph query (Cypher/SPARQL) [33]. For instance, an automated step could generate a Cypher query like `MATCH (p:Parameter {name:"Refresh Interval"})-[:DEFINED_IN]-`

`>(sec:Section)-[:PART_OF]->(doc:StandardDocument {name:"DDR4"}) RETURN p.value` if it infers such structure. This is an advanced capability, but it can yield very precise answers (like numeric values or relationships) directly from the KG [34] [35]. LangChain and other frameworks have utilities for LLMs to execute such graph queries if needed.

Additionally, the KG can be used to **filter or validate** vector results. For example, if the question specifically asks about DDR4, we can filter out any chunk that isn't from the DDR4 document. The KG knows which document each chunk belongs to (via relationships like *CHUNK_OF* Document) [36]. This ensures our context stays on target (no mixing DDR3 info if not relevant) and addresses the "context poisoning" issue [37] [38]. In practice, a *hybrid retrieval* might mean: first use vectors to get candidate passages, then use the KG to re-rank or filter those passages based on entity matching, as suggested in literature [39] [40]. This can significantly improve relevance and give us more confidence in the supporting data.

1. **Collating Context:** Now we have a collection of support data – likely a handful of text chunks and possibly some structured facts from the KG. We then construct a **prompt** for the LLM that includes the user's question and this retrieved context. For example, we might feed the LLM something like: "*User question: What does DDR4 spec say about refresh timing intervals?*" followed by "*Relevant information:*" and then bullet points or paragraphs from the retrieved chunks (and if applicable, a line like "The DDR4 standard defines *tREFI* (Refresh Interval) as 7.8 µs【source】" extracted from the KG). The exact formatting is a matter of prompt design, but the idea is to **augment the LLM's input with the relevant data** so it doesn't have to rely on its own memory (which may be outdated or hallucinated) [1].

2. **LLM Answer Generation:** With the query and context, the **LLM (Large Language Model)** generates an answer. This could be an open-source model deployed on-prem (such as LLaMA-2, GPT-J, etc., possibly fine-tuned on Q&A style), since we cannot use an external API. We need a model with sufficient capacity to handle the technical content and the prompt length (which could be a few thousand tokens of context for a detailed question). A model with at least 8k token context window is preferable so it can take in long spec excerpts. The model will be instructed (via system prompt) to use the provided context and not make up facts outside it. This mitigates hallucination: since we "ground" the model with actual snippets from the documents, it builds the answer from those [26]. In essence, RAG turns the LLM into a sort of fluent composer of the information we supply, rather than a knowledge source itself.

3. **Including Sources:** It's highly desirable in this use-case to provide sources or references in the answer (engineers will trust an answer more if they can see it came from the official spec). Because our pipeline keeps track of source for each chunk (and each KG fact is linked to a document section), we can have the LLM or the system include citations. For instance, after the LLM drafts an answer, we might post-process to append reference tags pointing to the document and section (much like the citation format we see in our research notes). Alternatively, we can format the input context such that the LLM learns to quote or cite the source (some fine-tuning might help with this behavior). Regardless, the architecture needs to ensure traceability: each piece of info has a pointer to where it came from in the docs or KG. This not only builds user trust but also helps debugging (if a wrong answer was given, we can check which source might have misled the model).

4. **Answer Delivery:** The chatbot front-end presents the LLM's answer, along with any source links. If the user is not satisfied or has a follow-up, that becomes a new query, possibly with conversation

context. We might include previous Q&A in the prompt for context if doing a conversational agent, but since the domain is technical, users often ask one-off factual questions. Still, handling follow-ups like "What about DDR3?" after asking about DDR4 is valuable – we could carry over context or detect the mention of DDR3 and fetch relevant info anew.

Throughout this process, **robustness** is key: if any step fails (say the vector DB is temporarily down or returns nothing, or the KG has no info on a term), the system should handle it gracefully (maybe fall back to keyword search as a last resort, or apologize that not enough info was found). For example, if vector search comes up empty, our code might then try a full-text search on a smaller index of documents to avoid total failure.

Agent Orchestration: We can implement the above logic with a framework (LangChain has chains for combined vector + graph queries) or custom code. The "agent" could even be partially an LLM decision: for instance, one might allow the LLM to call tools like a VectorSearch tool or a GraphQuery tool (some implementations give the LLM a set of tools, and it decides to first call a search, then read results, etc.). Given this is our first architecture discussion, we can keep it simpler – a deterministic pipeline that first does retrieval then calls LLM is easier to reason about and test. As we mature, we could experiment with more dynamic agents.

Performance Considerations: For ~200 users with up to, say, 20 queries each per day (so ~4k queries/day), performance should be manageable. The vector search (HNSW) is very fast (milliseconds for thousands of vectors). The graph query is also quick for targeted lookups (especially if using indexes on entity names). The slowest part is the LLM generation. If we host a large model (say 13B or 30B parameters) on a GPU, responses might take a few seconds each. We should ensure we have sufficient compute (maybe a couple of GPU servers if expecting concurrent queries). Another tactic is to use a smaller model or quantized model to speed up, but we must balance answer quality – technical accuracy is more important than a 2-second faster reply, in my opinion. Caching can also help: if certain queries or documents are frequently asked, we might cache those answers or at least the retrieved context to skip redundant steps.

5. Data Storage and Technology Choices

Now, let's explicitly outline the databases and storage solutions in this architecture, addressing the point about SQL vs NoSQL and where each is appropriate:

- **Raw Document Store (Staging)**: *Type:* Could be a NoSQL document DB or just files on disk. *Purpose:* hold the extracted JSON graph from each document before it's integrated into the KG. **Why NoSQL?** Because this data is essentially unstructured or semi-structured – each document's JSON may have a different shape, and we may iteratively refine it. A document DB allows storing each document's data as a single JSON object with nested structure, which maps well to how the data is naturally represented [11]. We gain flexibility to query within the JSON (e.g. find all docs where a certain term was extracted) without having to define a rigid schema upfront. Also, since this is mostly a transient/working storage (not user-facing), we prioritize write flexibility over complex query needs. A relational DB here would force us into EAV (entity-attribute-value) tables or lots of NULL columns for differing fields, which is not ideal. That said, if the team is more comfortable with SQL and our JSON extraction is fairly structured, Postgres could be used with JSONB columns – it offers some of the same flexibility, just not as seamlessly. The volume is small enough that either would work, but the **schema-on-read approach of NoSQL** aligns with our uncertain, evolving extraction structure [41].

- **Knowledge Graph Store**: *Type:* Graph Database (Neo4j recommended). *Purpose:* Store the curated knowledge graph (entities and relationships), supporting graph queries. **Why Graph DB (NoSQL)?** The graph's whole point is relationships; forcing this into a relational model would complicate queries greatly. Graph DBs let us naturally store nodes with properties and edges, and query patterns of connections efficiently [13]. We will likely get pushback about adding a new DB type, but we can point to industry practice: knowledge graphs in enterprises are almost always managed by graph databases for agility and performance. Additionally, Neo4j (for example) provides ACID guarantees, so we don't sacrifice consistency – it's transactional and reliable, just not relational tables. If absolutely needed, one could implement a graph in an SQL DB using join tables, but for multi-hop traversals, performance would degrade and queries become unreadable. We can cite that **graph databases are specifically optimized for connected data and can handle complex queries across relationships that would be very expensive in SQL** [19]. Also, our team's goal is to leverage the graph for things like multi-hop reasoning and explainability, which is exactly what a graph DB excels at [24].

- **Vector Database (Embedding Index)**: *Type:* Specialized Vector DB (Milvus/Weaviate) or integrated in graph DB. *Purpose:* Enable fast similarity search on text embeddings to retrieve relevant document chunks for questions. **Why a Vector DB?** Traditional databases are not designed for nearest-neighbor search in high-dim spaces. A vector DB provides approximate search algorithms that give results in milliseconds even with many vectors. We anticipate maybe tens of thousands of chunks – which might be okay to brute-force in memory, but if this grows or if we want sub-second response, an indexed approach is best. This is a relatively new technology, but it has matured quickly, and open-source options exist that we can host internally. If colleagues are skeptical, we can mention that even PostgreSQL now has a plugin for vectors – indicating that this is a recognized need in modern data systems. The complexity of adding one more component is offset by the benefit: **accurate semantic search** that a keyword search cannot achieve alone. Plus, as discussed, the vector search combined with the KG gives us a powerful retrieval mechanism that improves answer quality and reduces hallucination [42] [26].

- If opposition is strong, an alternative is using Neo4j's vector indexing (so we only have Neo4j as the data platform). This might be an attractive middle ground: Neo4j would store the graph and chunk nodes with embeddings, and we'd use its procedure to do similarity search. This eliminates a separate DB for vectors at the cost of possibly tying ourselves to Neo4j's ecosystem more. It's worth evaluating a proof-of-concept to see if the performance is sufficient. The Neo4j blog example indicated it works similarly to a standalone vector DB for moderate data volumes [21] [28].

- **Relational Database?** Interestingly, so far we haven't mandated any pure relational DB. It might be unusual in an enterprise setting not to use one at all. We could use a relational DB for *metadata* if needed – for instance, storing user info, access control lists, or even a simple index of documents (doc ID, title, etc.). However, these tasks can be handled by the graph or doc store too. If there's an existing SQL infrastructure that the team wants to use, we could integrate by, say, storing the documents and their basic info in SQL and then linking to the graph. But from an architecture purity perspective, it's not necessary. We're essentially using the graph DB as our source of truth for knowledge and the vector DB as an index. This is a **polyglot persistence** approach: using different stores for different needs (graph for relationships, vector for similarity, document for raw) [39] [43]. This is often the most optimized approach, but it does require maintaining integrations between them (ensuring if something is updated, we update all relevant stores). Given our update frequency

is low (documents updated infrequently), maintaining consistency across the stores is feasible with proper pipeline scripts.

NoSQL Justification Summary: We should prepare to justify NoSQL choices: - *Document store (NoSQL)* – because we have unstructured data extraction that doesn't fit neatly in tables (flexible schema needed) [11]. - *Graph DB (NoSQL)* – because our core data model is a graph of relationships which is clumsy to implement in tables; graph DB provides both flexibility and performance for this use-case [13]. - *Vector DB (typically NoSQL or key-value under the hood)* – because advanced similarity search is needed, which is outside the capability of traditional SQL indexes. Essentially, **the nature of the problem (unstructured text and knowledge relationships) calls for non-relational approaches**. Trying to force everything into a relational schema might technically work for storage, but would make the application layer far more complex and brittle, and likely underperform in retrieval. We can cite that many companies building QA bots or knowledge systems use exactly this combo (graph + vector DB) because each component is specialized [34] [35].

That said, we will also highlight that these NoSQL components can be run on-prem and securely, and we can manage them similarly to any database (backup, ACID (in Neo4j's case), etc.). There's nothing inherently less safe about NoSQL – the hesitation usually comes from unfamiliarity or perceived lack of consistency. We can reassure that the graph DB will handle transactions consistently (Neo4j is ACID for example), and eventual consistency (like in some NoSQL) is not a big factor here since we're not doing high-frequency updates.

6. Potential Breakpoints, Pain Points, and Mitigations

It's important to anticipate where this architecture might fail or encounter difficulties, so we can discuss solutions proactively. Here are key pain points and how to handle them:

- **Document Parsing Failures:** If some documents fail to parse (due to format quirks or new document types), the pipeline could break. *Mitigation:* Use robust parsing libraries (e.g. PyMuPDF for PDFs [44], Apache POI or similar for Office files) and have a fallback for unsupported content. Log and continue processing other documents rather than crashing. If certain diagrams are important but not machine-readable, consider asking SMEs to manually annotate them or provide separate data (could be added to KG later).

- **Incomplete or Incorrect Extraction:** The NLP extraction might miss entities or label relationships wrongly, leading to **garbage-in KG**. *Mitigation:* incorporate a **human-in-loop validation** for critical data. For example, after auto-extraction, present the SMEs with an interface or reports to verify key entities (like ensure all major sections were captured, all important parameters identified). We can prioritize review of things that will be heavily used. Over time, as the system learns (or if we fine-tune an extraction model on corrections), the quality will improve. Additionally, maintain the link from extracted facts to source text – this way, if an answer seems off, we can trace it back and correct the source extraction rather than blindly trusting the KG.

- **Ontology Gaps:** The initial ontology might not cover a concept that appears in documents, so extracted data doesn't map well (e.g. a new type of spec item appears). *Mitigation:* Design the pipeline to flag "unknown" types of data. For instance, if the extractor finds a relationship it can't classify into the ontology, it could tag it or store it in a generic "OtherRelation" bucket. In the

architecture meeting, be prepared to suggest an **ontology update process**: when SMEs notice such gaps, they will extend the ontology (add a new node/edge type) and we reprocess or adjust the KG to incorporate that. Ontology development is iterative – emphasize that we expect to refine it as we ingest more documents, which is normal.

- **Data Integration and Consistency:** When merging data into the KG, mistakes could cause duplicates (e.g. two nodes for the same concept with slightly different names, like "DDR5" vs "DDR5 Standard"). *Mitigation:* Implement **identity resolution** rules. Perhaps normalize names (uppercase, remove spaces) or assign IDs (if JEDEC has standard codes, use them). The KG loading script can check if a node with the same key exists and reuse it instead of creating a new one. We might also periodically run a script to find and merge duplicates if any slip in.

- **Performance Bottlenecks:** Potential slow points are the LLM and maybe the graph queries if not indexed. *Mitigation:* For LLM, ensure we have adequate hardware (GPUs) and possibly use a model that's a good trade-off of speed vs quality (we might start with a 7B or 13B model and see if fine-tuned it can perform well; if not, scale up to a larger model). We can also limit context size by retrieving only the top few chunks (maybe 3–5) to feed the LLM, to keep inference faster. For the graph DB, make sure to index frequently searched fields (like entity name) so lookups are fast. The vector DB should be fine if using approximate search, but we should monitor memory usage of the index and configure appropriately.

- **LLM Hallucination or Errors:** Despite providing context, the LLM might still occasionally produce incorrect or irrelevant answers (especially if the context had some noise or if the question goes beyond provided info). *Mitigation:* Use a strict prompting technique: instruct the LLM to answer only from given data and say "I don't know" if not in data. We could even have the LLM output which sources it used for each sentence (some approaches do that). Another safeguard is to have a lightweight verification layer – e.g. after answer is generated, check if the key facts in the answer appear in the retrieved context (this could catch if the LLM introduced something not supported by sources). If a hallucination is detected, we could refuse the answer or try a different strategy (like query more data).

- **System Failover and Fault Tolerance:** If one component is down (say the graph DB is offline), the system might still answer some questions with just vector search, but quality drops; or vice versa. *Mitigation:* While we expect to run all components reliably on-prem, we should handle exceptions in code. If a graph query fails, the agent could log it and proceed with vector results only (and maybe indicate limited info). We should also plan for backups of each DB (dump the Neo4j DB, snapshot the Milvus index, etc.) for disaster recovery.

- **Security Concerns:** Since data is confidential, all data stores must be access-controlled. *Mitigation:* Deploy them in a secure network zone, use authentication for the DBs, and ensure the chatbot application has proper user auth (especially if not all 300 users should see all data – though likely they all have clearance). Also, no data should leave the environment: ensure the LLM is self-contained (no hidden calls to external APIs). We might consider logging queries for auditing (to make sure sensitive info isn't being misused or to detect if someone tries to ask something they shouldn't). We also need to ensure the answers do not leak data beyond the user's permissions if such granularity is needed (in many internal cases, all engineers have access to all docs, but if not, incorporate a filter by user role when retrieving context).

- **Resistance to NoSQL/Graph:** A non-technical but real "pain point" could be pushback on using new types of databases. *Mitigation:* Come armed with success stories or references: for example, Neo4j is used in knowledge graphs at NASA, healthcare, etc., because relational DBs struggled with highly connected data. Also point out that our usage of NoSQL is targeted – we're not replacing a transactional system, we're adding capabilities that relational systems lack out-of-the-box (flexible JSON storage, graph traversals, vector similarity). Emphasize that we will still maintain rigor (backups, ACID where needed, etc.), and perhaps offer training or documentation to the team on these new techs. The learning curve is worth the benefits in this project.

- **What *Will Not Work*:** It may be useful to explicitly mention a few approaches that were considered and rejected:

- *Just use keyword search without a KG:* This would fail to handle complex queries that require linking concepts or multi-hop reasoning. A pure text search might find individual mentions but cannot synthesize an answer that pulls together information from different parts of documents or different documents. Also it lacks understanding of context, leading to either too many or too few results.
- *Just use the LLM on the raw documents (no retrieval):* This is not feasible because the LLM would either not have the documents in its training data (especially internal ones) or would exceed context limits to input them. It would also not be explainable or reliable, likely to hallucinate. The whole point of RAG is to ground the LLM – without retrieval, the LLM would be making things up or guessing [2].
- *Storing everything in a relational database:* As argued, forcing documents into tables, or trying to model the graph in tables, would lead to a fragile and slow system. For instance, imagine a SQL query to find a path between two entities through various relations – it would require multiple self-joins on a edges table, which gets very messy and non-performant compared to a graph query. The development speed would suffer greatly as well; we'd end up effectively writing our own graph processing atop SQL, which is reinventing the wheel poorly.
- *Using only the vector database (no KG):* This could answer many questions (the baseline RAG approach), but would struggle on questions that need aggregation or understanding of multiple pieces. For example, "compare feature X across DDR4 and DDR5" requires knowing to retrieve info on X from two places and comparing – a vector search might find the passages, but the LLM would have to notice the connection. A KG could explicitly store differences or at least help gather the two pieces systematically. Moreover, relying solely on vectors means less control – we might get some unrelated content sneaking in and have no way to enforce constraints like "only show me results about DDR4" except by keyword filtering. GraphRAG research has shown that integrating a KG yields more accurate and **comprehensive answers for complex queries** [45] [46]. The KG acts as a smart filter and context provider, reducing noise.
- *Not having a raw data store:* If we directly attempted to parse into the graph, any errors or ontology mismatches would be harder to fix. The raw JSON store gives us a checkpoint – we can adjust mappings or ontology and re-run the transformation without re-parsing the docs from scratch. Skipping this might not break the system, but it would make maintenance harder.

Each of these "what not to do" points reinforces why the chosen architecture is balanced and robust.

7. Security and Deployment Considerations

Given the confidential nature of the data, our entire stack will live **on-premise** (or in a secure private cloud not accessible to public). Here are some specifics:

- **On-Prem LLM:** We will deploy an appropriate LLM model on local servers. If we have GPU servers, we can run a model like LLaMA 2 13B or 30B, possibly fine-tuned on domain Q&A. This ensures no query or document text is sent to external services. We must also monitor the model's outputs to ensure it doesn't inadvertently log or output something it shouldn't (some LLM hosting solutions have logging disabled to maintain privacy).
- **Access Control:** The system is for internal engineers; we might integrate it with our existing authentication (e.g. the user logs in via SSO to the chatbot). The knowledge base itself might have some segmentation (if certain documents are only for certain teams). If needed, the graph can include access tags, or we maintain a simple ACL mapping of users to document IDs. Then at query time, we filter vector and graph results to only those the user is allowed to see [38]. This is a detail to design if required by policy.
- **Network and Data Security:** Each database should ideally run in our secure network, with encryption at rest enabled if available (Neo4j Enterprise supports TDE, many vector DBs support encryption, or we rely on disk encryption). Communication between components (the chatbot backend to DBs) should be over secure channels (within a data center this might be less of an issue, but still). We will likely dockerize these services for ease of deployment, but ensure secrets (like DB passwords) are managed properly (in vaults or configs not exposed).
- **Audit and Monitoring:** We should log queries and system actions. This can help detect any unusual activity (like a user trying to extract large chunks of data via the bot, which might indicate misuse) and can also help improve the system (logs of failed queries or where the bot said "I don't know" could highlight content gaps to address). These logs must themselves be protected since they could contain sensitive queries or excerpts.

- **Scalability:** While current load is low, if this becomes popular and more teams onboard, we should ensure the architecture can scale. Neo4j can scale vertically or through clustering (though clustering is more for HA than for sharding, as it's not trivially sharded). Vector DB like Milvus can cluster or we can upgrade hardware. The stateless parts (the API logic, etc.) can be containerized and scaled horizontally behind a load balancer. Essentially, our design can handle an increase in data or users by allocating more resources or distributed instances, as long as we plan for how to keep data consistent (for instance, if vector DB is clustered, it handles distribution internally; if we needed a distributed graph, something like NebulaGraph could be a choice, but that's probably overkill here).

- **Maintenance:** We will maintain the KG as living documentation. New document versions will be ingested similarly. We should consider how to **version the knowledge** – e.g. if a JEDEC spec is updated, do we replace the old info or keep both? A good approach is to keep old versions in the KG but mark them as superseded, so the chatbot could answer "DDR4 (Rev 2023) had X = 1.2V, updated to 1.1V in Rev 2024" if asked. The ontology can include a *versionOf* relationship to link document versions. This way, our KG becomes a time-aware repository of knowledge evolution, which can be very useful for engineering queries about changes over time.

Conclusion and Recommendations

In summary, the proposed architecture uses a combination of specialized components to tackle the complexity of a **graph-based RAG system** for semiconductor documents. We ingest and process documents through an NLP pipeline to build a rich knowledge graph (with the guidance of a curated ontology), and we index the document text in a vector database for semantic search. A chatbot agent then uses both the knowledge graph and the vector search to retrieve relevant information and feed it to an LLM, which generates answers for the user. This architecture addresses the needs for accuracy, reasoning over technical content, and security, by **grounding the AI in verified company knowledge** and keeping all data on-prem.

We have chosen tools suited to each task: a graph database for relationships, a vector engine for embeddings, and JSON stores for flexibility, acknowledging that a one-size-fits-all database would struggle to meet all requirements efficiently. While this polyglot approach introduces multiple technologies (which might be novel to some team members), each is justified by a specific benefit (e.g. flexible schema, explainable multi-hop queries, or semantic similarity search) that directly contributes to the end goal of an intelligent, reliable engineering chatbot. In architecture discussions, I will emphasize how these components work together and highlight how we mitigate potential issues (from data errors to system failures), referencing both best practices and the latest developments (such as GraphRAG techniques that combine vectors and KGs for better QA performance [45] [47]).

By being aware of edge cases (like "context poisoning" from vector results [30] or the need for query-time access control [38]) and having strategies to handle them, we show a thorough understanding of the system's robustness. Moreover, we have a plan for ongoing improvement: using user feedback and SME input to refine the knowledge graph and ontology over time, ensuring the system stays accurate and up-to-date.

I'm confident that with this architecture, we can deliver a powerful internal tool: engineers will be able to ask complex questions and get trustworthy answers with supporting evidence, saving countless hours digging through PDFs. And as the corpus grows, the knowledge graph will only become more useful, revealing connections across documents. This design not only solves the immediate problem but lays a foundation for advanced capabilities (like automated reasoning or recommendations in the future).

Going into the meeting, I'll be prepared to discuss why each piece is necessary and how we'll implement it, and I'll back it up with the insights from similar efforts (industry examples and research) that validate our approach. This way, I will come across as someone who has considered all angles – from technical details and edge cases to the high-level vision – and can guide the team in building a successful Knowledge-augmented RAG system.

Figure 2: Conceptual architecture of the RAG chatbot using a knowledge graph. The user's query is processed by a "smart retrieval" layer that uses both vector similarity search and the knowledge graph to find relevant information. The LLM then generates an answer grounded in these results, improving accuracy and trust in the response [32] [34].

---

1 21 22 23 32 Using a Knowledge Graph to implement a RAG application
https://neo4j.com/blog/developer/knowledge-graph-rag-application/

2 18 19 26 42 Enhance Your RAG Applications with Knowledge Graph RAG | Build Intelligent Apps With SingleStore
https://www.singlestore.com/blog/enhance-your-rag-applications-with-knowledge-graph-rag/

3 4 6 7 8 9 27 28 29 31 36 44 Knowledge Graph Extraction and Challenges - Graph Database & Analytics
https://neo4j.com/blog/developer/knowledge-graph-extraction-challenges/

5 10 From Text to a Knowledge Graph: The Information Extraction Pipeline
https://neo4j.com/blog/genai/text-to-knowledge-graph-information-extraction-pipeline/

11 12 13 Relational vs Nonrelational Databases - Difference Between Types of Databases - AWS
https://aws.amazon.com/compare/the-difference-between-relational-and-non-relational-databases/

14 15 16 17 Domain Ontologies: Indispensable for Knowledge Graph Construction – Aneesh Sathe
https://aneeshsathe.com/2025/01/15/domain-ontologies-indispensable-for-knowledge-graph-construction/

20 30 33 34 35 37 38 39 40 43 How to Implement Graph RAG Using Knowledge Graphs and Vector Databases | by Steve Hedden | TDS Archive | Medium
https://medium.com/data-science/how-to-implement-graph-rag-using-knowledge-graphs-and-vector-databases-60bb69a22759

24 25 45 46 47 GraphRAG Explained: Enhancing RAG with Knowledge Graphs | by Zilliz | Medium
https://medium.com/@zilliz_learn/graphrag-explained-enhancing-rag-with-knowledge-graphs-3312065f99e1

41 SQL vs NoSQL: What's the Right Choice for Your Data in 2025?
https://weld.app/blog/sql-or-nosql-databases-which-one-is-best-for-storing-data-in-your-organisation