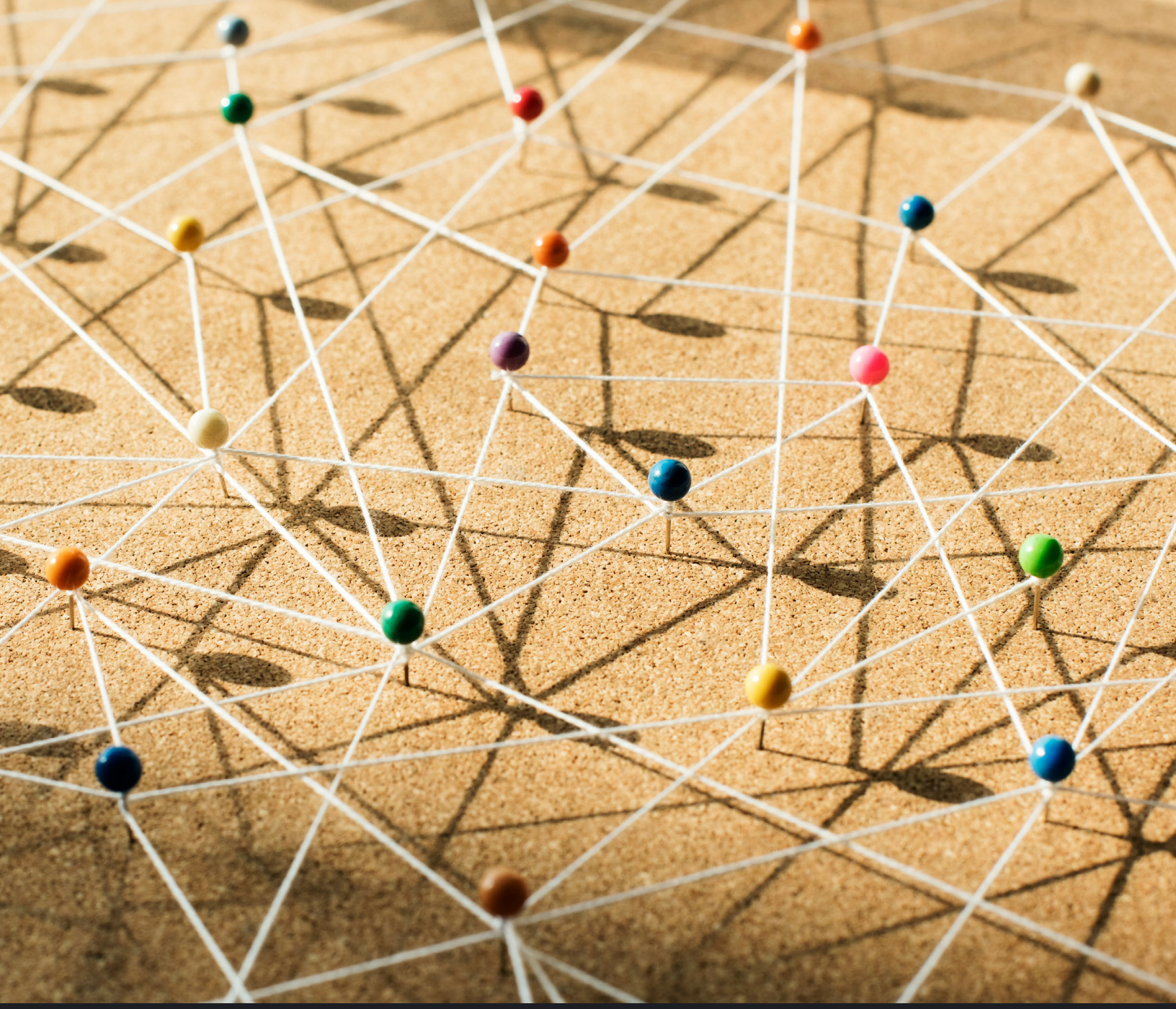


Context engineering with hybrid search for agentic AI

elastic search labs

Table of contents

 |

INTRODUCTION

Our brand new agentic AI world

The pace at which AI capabilities are evolving has developers of all kinds feeling both giddy and astonished. We first saw large language models (LLMs) and vector search launch us into the semantic revolution, where we were no longer hunting and pecking with keywords to find things. Then, LLMs showed us new ways of interacting with our data, using chat interfaces to transform natural language requests into responses that distill vast knowledge bases into easily consumable summaries. We now (already!) have the beginnings of automated LLM-driven logic in the form of "agentic AI" workflows that can semantically understand an incoming request, reason about the steps to take, and then choose from the tools available to iteratively execute actions to achieve those goals.

The promise of agentic AI is forcing us to evolve from primarily using **prompt engineering** to shape our generative AI interactions, to focusing on how we can help agentic tools obtain the most relevant and efficient additional information the LLM needs to consider when generating its responses — **context engineering** is the next frontier. Hybrid search is by far the most powerful and flexible means to surface relevant context, and the Elasticsearch Platform opens up a whole new way to leverage data in service to context engineering. We're going to discuss how LLMs have changed the world of information retrieval from two angles, as well as how they can work together for greater results. There's quite a lot of ground to cover ....

PART I

LLMs and search

Let’s start from the angle of how LLMs have changed the ways we access and retrieve information.

Our lexical legacy

We've all been living in the somewhat limited lexical search world (pretty well, as best we can) for a long time. Search is the first tool we reach for when researching or beginning a new project, and until recently, it was up to us to word our queries in a way that a lexical search engine understands. Lexical search relies on matching some form of query terms to keywords found in a document corpus — regardless of whether the content is unstructured or structured. For a lexical search to return a document as a hit, it has to have matched on that keyword (or had a controlled vocabulary like a synonym list or dictionary make the conceptual connection for us).

```
JSON
POST my-index/_search {
  "size": 10,
  "query": {
    "multi_match": {
      "query": "machine learning applications",
      "fields": ["title", "content"]
    }
  }
}
```

An example lexical multi-match query

At least search engines have the ability to return hits with a relevance score. Search engines provide a wealth of query syntax options to target indexed data effectively and built-in relevance algorithms that score the results relative to the intent of the user's query syntax. Search engines reap the benefits of decades of advances in relevance ranking algorithms, and that makes them an efficient data retrieval platform that can deliver results scored and sorted by their relevance to the query. Databases and other systems that use SQL as their main method for retrieving data are at a disadvantage here: There is no concept of relevance in a database query; the best they can do is sort results alphabetically or numerically. The good news is you'll get all the hits (recall) with those keywords, but they're not necessarily in a helpful order relative to *why* you were asking for them (precision). That's an important point, as we'll see shortly.

Enter the (semantic) dragon

The potential for vector representations of information as an alternative to keyword search has been researched for quite a long time. Vectors hold a lot of promise because they get us out of the keyword-only mode of matching content. Because they're numeric representations of terms and weights, vectors allow concepts to be mathematically close based on a language model's understanding of how terms relate to each other in the training domain. The long delay for general-purpose vector search was due to models being mostly limited to specific domains — they simply weren't large enough to sufficiently understand the many different concepts a term might represent within different contexts.

It wasn't until LLMs came along a few years ago with their ability to train on much larger amounts of data (using transformers and attention) that vector search became practical. The size and depth of LLMs finally allowed vectors to store enough nuance for them to actually capture semantic meaning. That sudden increase in the depth of understanding allowed LLMs to serve a large number of natural language processing (NLP) functions that were previously locked, perhaps the most impactful being the ability to infer the most likely next term in a sequence given the context of what's in the sequence so far. Inference is the process that gives generative AI its near-human ability to produce text. AI-generated text is informed by the LLM's understanding of how terms are related within its training data. The LLM uses the phrasing of the request to disambiguate between different contexts the terms might appear in.

As magical as generative AI is, there *are* limitations to LLMs that cause errors in quality and accuracy, commonly called hallucinations. Hallucinations occur when the LLM doesn't have access to the information (or isn't guided to the correct context) to base its answer in truth so, being helpful, it will instead generate a confident and plausible-sounding response that's made up. Part of the cause is that while LLMs learn the usage of language within large domains of diverse information, they have to stop training at a certain point in time. This means the model can only know what was accurate up to the time it stopped training. In addition, the model usually doesn't know about privately held data (data not available on the public internet), and that's especially significant when that data contains specific terms and nomenclature.

Vector databases

LLMs vectorize content to its model space using a technique called text embedding, which refers to embedding or mapping the content's semantic meaning within the model's worldview based on the training it received. There are a few steps involved to prepare and process content for embedding, including chunking and tokenization (and subword tokenization). The result is typically a set of dense vectors representing the model's understanding of that chunk of content's meaning within its vector space. Chunking is an inexact process that's meant to fit content into the limitations of a model's processing constraints for generating embeddings, while also attempting to group related text into a chunk using semantic constructs like sentence and paragraph indicators.

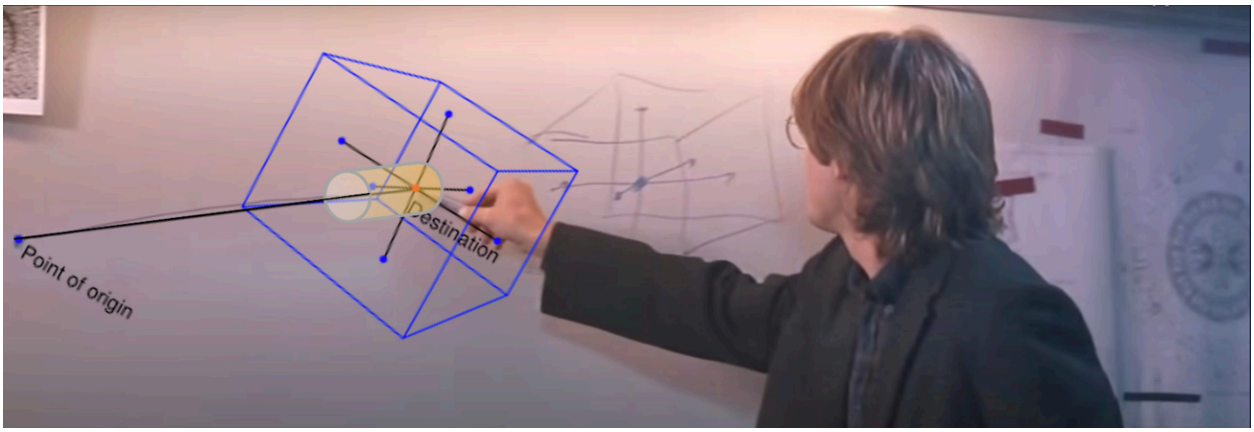
The need for chunking can create a bit of semantic lossiness in an embedded document because individual chunks aren't fully associated with other chunks from the same document. The inherent opaqueness of neural networks can make this lossiness worse. An LLM is truly a "black box" where the connections between terms and concepts made during training are non-deterministic and not interpretable by humans. This leads to issues with explainability, repeatability, unconscious bias, and potentially a loss of trust and accuracy. Still, the ability to semantically connect ideas and not be tied to specific keywords when querying is extremely powerful:

```
JSON
POST my-index/_search {
  "size": 10,
  "query": {
    "semantic": {
      "query": "machine learning applications",
      "field": "semantic-content-field"
    }
  }
}
```

An example semantic query

There's one more issue to consider for vector databases: They're not search engines, they're databases! When a vector similarity search is performed, the query terms are encoded to find a set of (embedding) coordinates within the model's vector space. Those coordinates are then used as the bullseye to find the documents that are the "nearest neighbors" to the bullseye — meaning a document's rank (or placement in the results) is determined by the calculated similarity distance of that document's coordinates from the query's coordinates. In which direction should ranking take precedence? Which of the possible contexts is closest to the user's intent?

It's like a scene from the movie Stargate, where we have the six coordinate points that intersect to tell us the destination (the bullseye), but we can't get there without knowing the "7th symbol" — the coordinates of the starting point representing the user's subjective intention. So rather than the relative ranking of vectors being based on an ever-expanding and undifferentiated sphere of similarity, by considering the subjective intent of the query through expressive syntax and relevance scoring, we can instead get something resembling a *cylinder* of graduated subjective relevance.



Source: Amazon MGM Studios

The inference capabilities of an LLM might help identify the most probable context *it has* for the query, but the problem is that *without help*, the incoming query's coordinates can *only* be determined by how the model was originally trained.

In some ways, you could say vector similarity goes to the opposite extreme from a strictly keyword match. Its strength lies in its ability to overcome the issues of term mismatch, but almost to a fault: LLMs tend toward unifying related concepts rather than differentiating between them. Vector similarity improves our ability to match content semantically, but it doesn't guarantee precision. It can overlook exact keywords and specific details that aren't disambiguated enough by the model. Vector similarity search is powerful in itself, but we need ways to correlate the results we retrieve from a vector database with results from other retrieval methods.

Reranking techniques

Now is a good time to mention a general technique called reranking, which re-scores or normalizes result sets to a unified rank order. The need for reranking could be due to results from multiple sources or retrieval methods having different ranking/scoring mechanisms (or none at all, SQL!), or reranking could be used to align the results from non-semantic sources to the user's query semantically. Reranking is a second-stage operation, meaning a set of results that have been collected by some *initial retrieval* method (i.e., SQL, lexical search, vector search) are then reordered with a different scoring method.

There are several approaches available, including Learning-to-Rank (LTR) and reciprocal rank fusion (RRF). LTR is useful for capturing search result features (e.g., likes, ratings, clicks) and using those to score and boost or bias results. RRF is perfect for merging results returned from different query modalities (e.g., lexical and vector database searches) together into a single result list. Elastic also provides the flexibility to adjust scores using linear reranking methods.

One of the most effective reranking techniques, however, is semantic reranking. It uses an LLM's semantic understanding to analyze the vector embeddings of both the query and results together and then apply relevance scoring/rescoring to determine the final order. Semantic reranking requires a connection to a reranking model, of course, and you can now use reranker models (from Jina AI) natively in Elasticsearch on managed GPUs with Elastic Inference service (EIS), enabling fast, reliable search and agentic workflows. In addition, Elasticsearch provides an Inference API that lets you create `rerank` endpoints that use built-in models (Elastic Rerank), imported third-party models, or externally-hosted services like Cohere or Google Vertex AI. You can then perform reranking through the retriever query abstraction syntax:


```
JSON
POST my-index/_search {
  “size”: 10,
  “retriever”: {
    “text_similarity_reranker”: {
      “retriever”: {
        “rrf”: {
          “retrievers”: [
            {
              “standard”: {
                “query”: {
                  “multi_match”: {
                    “query”: “machine learning applications”,
                    “fields”: [“title”, “content”]
                  }
                }
              }
            },
            {
              “knn”: {
                “field”: “semantic-content-field”,
                “k”: 10,
                “num_candidates”: 100,
                “query_vector_builder”: {
                  “text_embedding”: {
                    “model_id”: “my-text-embedding-model”,
                    “model_text”: “machine learning applications”
                  }
                }
              }
            }
          ],
          “rank_window_size”: 50,
          “rank_constant”: 20
        }
      }
    },
    “field”: “content”,
    “inference_id”: “my-reranker”,
    “inference_text”: “machine learning applications”,
    “rank_window_size”: 20
  }
}
```

An example multistage retriever reranking operation

 |

Sounds great, right? We can perform reranking on results from disparate sources and get close to semantic understanding of all types of content. Semantic reranking can be expensive both computationally as well as in processing time required. Because of that, semantic reranking can only feasibly be done on a limited number of results, which means that *how* those initial results get retrieved is important.

The context retrieval method matters

Subjective intent is an important factor in determining the accuracy of a result and scoring its relevance. Without the ability to consider the user's intent for performing the query (as expressed through flexible syntax, or by second-stage reranking), we can only select from the existing contexts already encoded within the model space. The way we typically address this lack of context is through techniques like retrieval augmented generation (RAG). The way RAG works is it effectively shifts the query's coordinates by including additional related terms returned from a pre-query for contextually relevant data. That makes the engine providing that additional context and *its* initial method for performing retrieval all the more important to the accuracy of the context!

Let's review the different context retrieval methods and how they can help or hurt a RAG operation:

- **Hybrid search retrieval without a search engine still lacks subjective relevance.** If the platform providing RAG is primarily SQL-based underneath (which includes most "data lake" platforms), it's lacking relevance scoring at the initial retrieval stage. Many data lake platforms provide their own version of hybrid retrieval (not search), usually combining reranking techniques like semantic reranking and RRF on their SQL-based retrieval and vector database results. A simple sort is obviously insufficient for subjective ranking, but even when used as the basis for a second-stage semantic reranking operation, SQL as the first-stage retrieval becomes a problem when the semantic reranking is performed only on the "top k" hits. Without some way to score results at retrieval, what guarantee do we have that the *best* results are actually in the top results?

- **Vector similarity alone isn't good enough for RAG.** It's really due to a compounding set of issues: It's the lossiness of embedding, along with naive chunking methods, how similarity is calculated, and the crucial missing component of subjective intent. One of the main goals of RAG is to ground generative AI interactions in objective truth, both to prevent hallucinations as well as to inform the LLM about the private information it had no knowledge of during training. We can use the additional context provided through RAG to constrain and direct LLMs to consider the connections and

details we know are most important to answering the question at hand. To do that, we need to use *both* semantic and lexical approaches.

- **File-based grep/regex RAG.** There are some quarters of the agentic AI universe pointing to the use of vastly enlarged context windows that access local files via grep and regex for RAG rather than external retrieval platforms. The idea is that with a much larger context window available, LLMs will be able to make conceptual connections within their own thinking space rather than relying on chunked bits and pieces and multiple retrieval methods to collect relevant information. While it's true in theory that having an entire document delivers a fuller picture than document segments, this can only work in small data domains (or, for example, when supplying files for vibecoding), and even then the initial retrieval method is a scan of all documents with a keyword-only match.

Search is more than retrieval

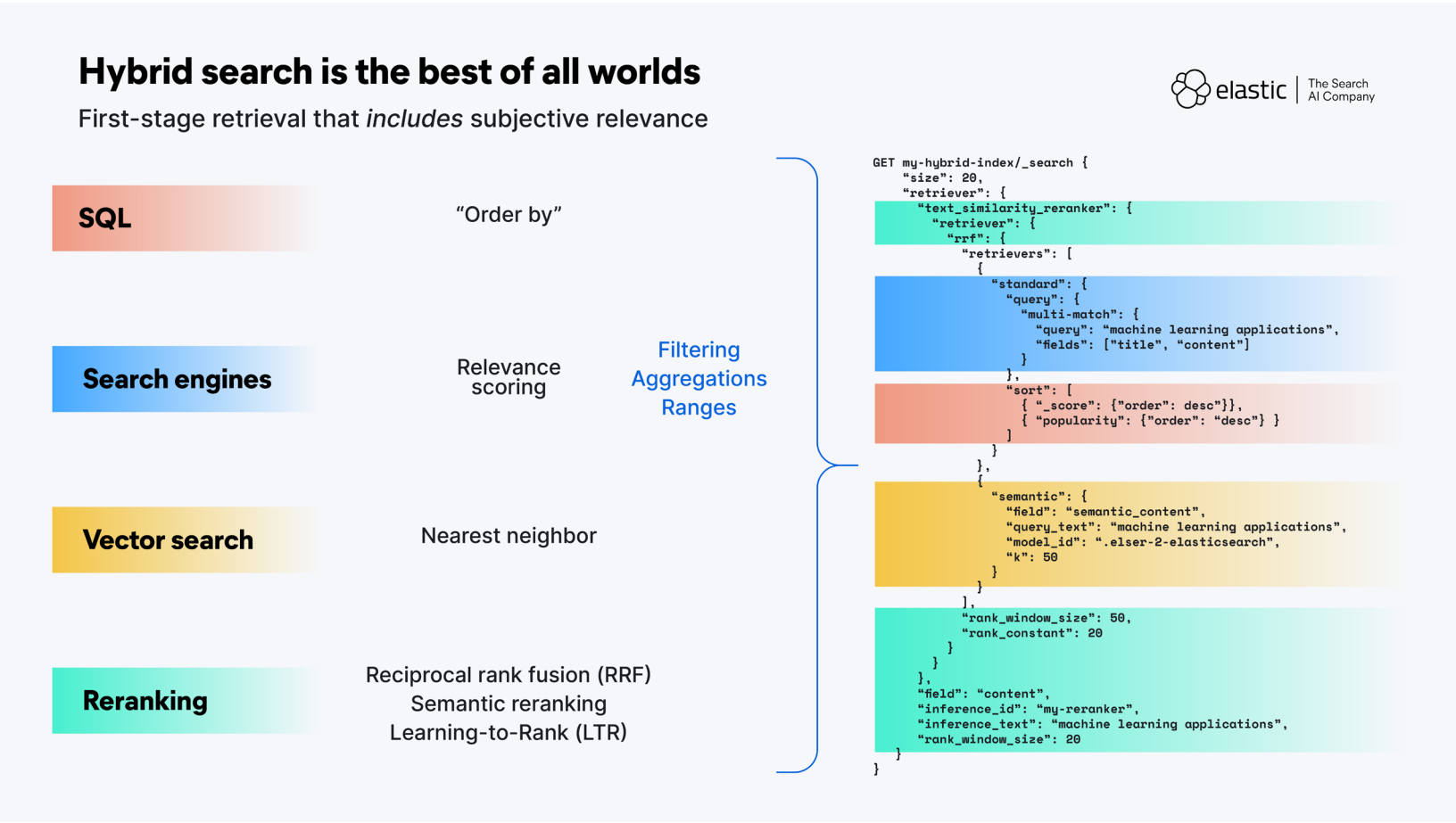
Search engines are purpose-built for making queries as fast and flexible as possible. Internally, they utilize specialized data structures for storing and retrieving different kinds of data in ways that cater to those data types. Elasticsearch provides optimized storing and querying of essentially all types of data, including unstructured/full-text lexical search (match, phrase, proximity, multi-match), fast keyword (exact match) matching and filtering, numeric ranges, dates, and IP addresses. It is very flexible in how it stores document structures (e.g., nested or flattened docs). Elasticsearch is also a native vector database that can store and query both sparse and dense vector types, and we continue to explore innovative ways (for example, Better Binary Quantization (BBQ) and DiskBBQ) to maintain search fidelity while improving the speed, scalability, and costs associated with vectorized content.

The Elasticsearch Platform also provides built-in data resilience and high availability, and it includes data lifecycle management capabilities, such as searchable snapshots that allow you to keep infrequently accessed or long-term retention data on cost-effective object storage while remaining fully searchable.

Hybrid search is the best of all worlds

Hybrid search (not just hybrid retrieval!) combines the strengths of traditional lexical search with the semantic understanding of LLMs and vector similarity search. This synergy allows for targeting highly relevant results at the retrieval stage through any of

the flexible query syntax options a search engine provides: intent-driven syntax options and relevance scoring, multimodal data retrieval, filtering, aggregations, and biasing. With search syntax like Elasticsearch Query Language (ES|QL) and multistage retrievers, we can flexibly combine traditional search with semantic search, filters, and multiple reranking techniques all in one request.



One of the biggest benefits of hybrid search is that your queries can use specialized syntax for multiple different data types simultaneously. Those different query syntaxes can be used not just for finding results but also as filters or aggregations on the results. For example, one of the most common query types that is frequently combined with other syntax is geospatial analysis. You can do things like query for results that have geo coordinates within a specified distance of a point or request aggregations on your results by region or to track and alert on movements into/out of a zone. With hybrid search, you have the flexibility to combine syntaxes to target results in the most accurate way to retrieve the content closest to your context.

PART II

LLMs and chat

With that (fairly extensive) background on the ways LLMs have changed the underlying processes of information retrieval, let's see how they've also changed the way we query for data.

A new way of interacting with data

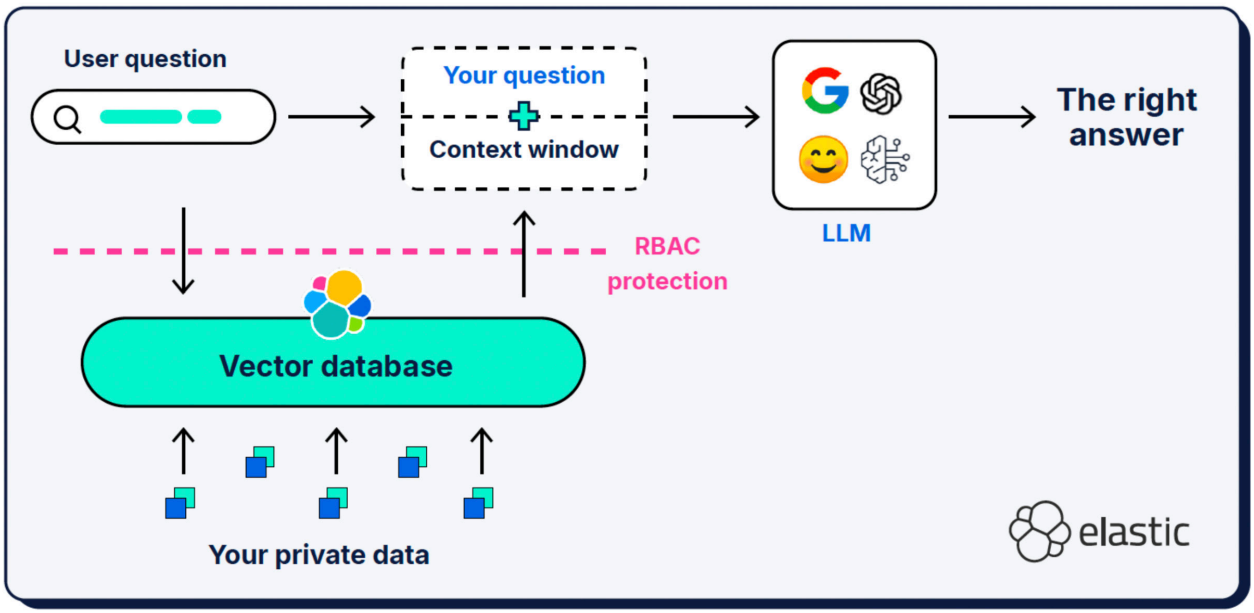
Generative (GenAI) and agentic AI do things differently than traditional search. Whereas the way we used to begin researching information was a search ("let me Google that ..."), the initiating action for both GenAI and agents is usually through natural language entered into a chat interface. The chat interface is a discussion with an LLM that uses its semantic understanding to turn our question into a distilled answer, a summarized response seemingly coming from an oracle that has a broad knowledge of all kinds of information. What really sells it is the LLM's ability to generate coherent, thoughtful sentences that string together the bits of knowledge it surfaces — even when it's inaccurate or totally hallucinated, there's a truthiness to it.

That old search bar we've been so used to interacting with can be thought of as the RAG engine we used when *we ourselves* were the reasoning agent. Now, even internet search engines are turning our well-worn "hunt and peck" lexical search experience into AI-driven overviews that answer the query with a summary of the results, helping users avoid the need to click through and evaluate individual results themselves.

Generative AI and RAG

Generative AI tries to use its semantic understanding of the world to parse the subjective intention stated through a chat request and then uses its inference abilities to create an expert answer on the fly. There are several parts to a generative AI interaction: It starts with the user's input or query. Previous conversations in the chat session can be used as additional context, and the instructional prompt tells the LLM how to reason and what procedures to follow in constructing the response. Prompts have evolved from simple "explain this to me like I am a five-year-old" type of guidance to complete breakdowns for how to process requests. These breakdowns often include distinct sections describing details of the AI's persona or role, pre-generation reasoning and internal thought process, objective criteria, constraints, output format, and audience, as well as examples to help demonstrate the expected results.

In addition to the user's query and the system prompt, RAG provides additional contextual information in what's called a "context window." RAG has been a critical addition to the architecture; it's what we use to inform the LLM about the missing pieces in its semantic understanding of the world.



Context windows can be kind of persnickety in terms of what, where, and how much you give them. Which context gets selected is very important, of course, but the signal-to-noise ratio of the provided context also matters, as well as the length of the window.

Too little information

Providing too little information in a query, prompt, or context window can lead to hallucinations because the LLM can't accurately determine the correct semantic context to generate a response from. There are also issues with the vector similarity of document chunk sizes — a short, simple question may not semantically align with the rich, detailed documents found in our vectorized knowledge bases. Query expansion techniques such as Hypothetical Document Embeddings (HyDE) have been developed that use LLMs to generate a hypothetical answer that is richer and more expressive than the short query. The danger here, of course, is that the hypothetical document is itself a hallucination that takes the LLM even farther astray from the correct context.

Too much information

Just like it does to us humans, too much information in a context window can overwhelm and confuse an LLM about what the important parts are supposed to be. Context overflow (or context rot) affects the quality and performance of generative AI operations; it greatly impacts the LLM's "attention budget" (its working memory) and dilutes relevance across many competing tokens. The concept of "context rot" also includes the observation that LLMs tend to have a positional bias — they prefer the content at the beginning or end of a context window over content in the middle section.

Distracting or conflicting information

The larger a context window gets, the more chance there is that it might include superfluous or conflicting information that can serve to distract the LLM from selecting and processing the correct context. In some ways, it becomes a problem of garbage in/garbage out. Just dumping a set of document results into a context window gives the LLM a lot of information to chew on (potentially too much), but depending on how the context was selected there is a greater possibility for conflicting or irrelevant information seeping in.

Agentic AI

Agentic AI is a very exciting new usage of LLM chat interfaces that expands on generative AI's (can we call it "legacy" already?) ability to synthesize responses based on its own knowledge and contextual information you provide.

As generative AI became more mature, we realized there was a certain level of tasking and automation we could have LLMs perform, initially relegated to tedious low-risk activities that can easily be checked/validated by a human. Over a short period of time, that initial scope grew: An LLM chat window can now be the spark that sends an AI agent off to autonomously plan, execute, and iteratively evaluate and adapt its plan to achieve its specified goal. Agents have access to their LLMs' own reasoning, chat history, and thinking memory (such as it is), and they also have specific tools they can utilize to achieve that goal. We're also now seeing architectures that allow a top-level agent to function as the orchestrator of multiple sub-agents, each with their own logic chains, instruction sets, context, and tools.

Agents are the entry point to a mostly automated workflow. They're self-directed in that they are able to chat with a user and then use "logic" to determine what tools they have available to help answer the user's question. Tools are usually considered passive compared to agents and are built to do one type of task. The *types* of tasks a tool could perform are kind of limitless (which is really exciting!), but a primary task tools perform is to gather contextual information for an agent to consider in executing its workflow.

As a technology, agentic AI is still in its infancy and prone to the LLM equivalent of attention deficit disorder — it easily forgets what it was asked to do and often runs off to do other things that weren't part of the brief at all. Underneath the apparent magic, the "reasoning" abilities of LLMs are still based on predicting the next most likely token in a sequence.

For reasoning (or someday, artificial general intelligence (AGI)) to become reliable and trustworthy, we need to be able to verify that when given the correct, most up-to-date information, they will reason through the way we expect them to (and perhaps give us that little extra bit more that we might not have thought of ourselves). For that to happen, agentic architectures will need the ability to communicate clearly (protocols), adhere to the workflows and constraints we give them (guardrails), remember where they are in a task (state), manage their available memory space, and validate their responses are accurate and meet the task criteria.

Talk to me in a language I can understand

As is common in new areas of development (especially so in the world of LLMs), there were initially quite a few approaches for agent-to-tool communications, but they quickly converged on the Model Context Protocol (MCP) as the de facto standard. The definition of Model Context Protocol is truly in the name: It's the **protocol** a **model** uses to request and receive **contextual** information. MCP acts as a universal adapter for LLM agents to connect to external tools and data sources; it simplifies and standardizes the APIs so that different LLM frameworks and tools can easily interoperate. That makes MCP a kind of pivot point between the orchestration logic and system prompts given to an agent to perform autonomously in the service of its goals, and the operations sent to tools to perform in a more isolated fashion (isolated at least with regards to the initiating agent).

This ecosystem is all so new that every direction of expansion feels like a new frontier. We have similar protocols for agent-to-agent interactions as well as other projects for improving agent reasoning memory, for selecting the best MCP server for the job at hand, and using semantic analysis, such as zero-shot classification and pattern detection on input and output as guardrails to control what an agent is allowed to operate on.

The underlying intent of each of these projects is to improve the quality and control of the information returned to an agent or GenAI context window. While the agentic AI ecosystem continues to develop the ability to handle that contextual information better (to control, manage, and operate on it), there will always be the need to retrieve the most relevant contextual information as the grist for the agent to mill on.

Welcome to context engineering!

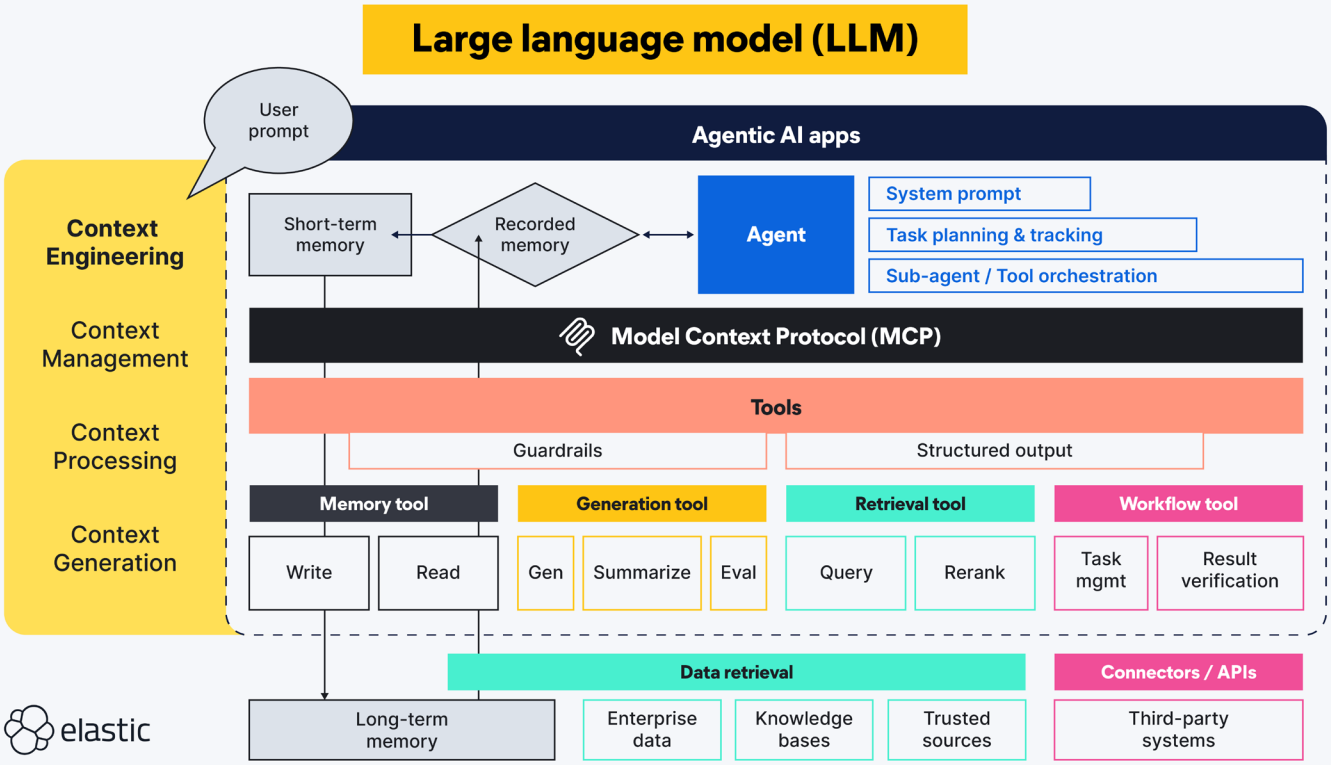
If you're familiar with generative AI terms, you've probably heard of "prompt engineering." At this point, it's almost a pseudo-science of its own. Prompt engineering is used to find the best and most efficient ways of proactively describing the behaviors you want the LLM to use in generating its response. Context engineering extends prompt engineering techniques beyond the agent side to also cover available context sources and systems on the tools side of the MCP protocol, and it includes the broad topics of context management, processing, and generation:

- **Context management:** Context management is related to maintaining state and context efficiency across long-running and/or more complex agentic workflows. It involves iterative planning, tracking, and orchestration of tasks and tool calling to accomplish the agent's goals. Due to the limited "attention budget" agents have to work within, context management is largely concerned with techniques that help refine the context window to capture both the fullest scope and the most important bits of context (its precision versus recall). Techniques include compression, summarization, and persisting context from previous steps or tool calls to make room in working memory for additional context in subsequent steps.

- **Context processing:** Context processing follows logical and mostly programmatic steps to integrate, normalize, or refine context acquired from disparate sources, so that the agent can reason across all the context in a somewhat uniform manner. The underlying work is to make context from all sources (prompts, RAG, memory, etc.) consumable by the agent as efficiently as possible.

- **Context generation:** If context processing is about making retrieved context usable to the agent, then context generation gives the agent the outreach to request and receive that additional contextual information at will, but also with constraints.

 |



The various ephemera of LLM chat applications map directly (and sometimes in overlapping ways) to those high-level functions of context engineering:

- **Instructions/system prompt:** Prompts are the scaffolding for how the generative (or agentic) AI activity will direct its thinking toward accomplishing the user's goal. Prompts are context in their own right; they aren't just tonal instructions. They also frequently include task execution logic and rules for things like "thinking step by step" or "take a deep breath" before responding to validate the answer fully addresses the user's request. Recent testing has shown markup languages are very effective at framing the different parts of a prompt, but care should also be taken to calibrate the instructions to a sweet spot between too vague and too specific; we want to give enough instruction for the LLM to find the right context, but not be so prescriptive that it misses unexpected insights.

- **Short-term memory (state/history):** Short-term memory is essentially the chat session interactions between the user and the LLM. These are useful in refining context in live sessions and can be saved for future retrieval and continuation.

- **Long-term memory:** Long-term memory should consist of information that is useful across multiple sessions. And it's not just domain-specific knowledge bases accessed through RAG; recent research uses the outcomes from previous agentic/generative AI requests to learn and reference within current agentic interactions. Some of the most interesting innovations in the long-term memory space are related to adjusting how state is stored and linked to so that agents can pick up where they left off.

- **Structured output:** Cognition requires effort, so it's probably no surprise that even with reasoning capabilities, LLMs (just like humans) want to expend less effort when thinking, and in the absence of a defined API or protocol, having a map (a schema) for how to read data returned from a tool call is extremely helpful. The inclusion of structured outputs as part of the agentic framework helps to make these machine-to-machine interactions faster and more reliable, with less thinking-driven parsing needed.

- **Available tools:** Tools can do all sorts of things, from gathering additional information (e.g., issuing RAG queries to enterprise data repositories, or through online APIs) to performing automated actions on behalf of the agent (like booking a hotel room based on the criteria of the request from the agent). Tools could also be sub-agents with their own agentic processing chains.

- **Retrieval augmented generation (RAG):** RAG is the technique for providing the additional information the LLM didn't have access to when it was trained, or it's a reiteration of the ideas we think are most important to get the right answer — the one that's most relevant to our subjective query.

 |

Phenomenal cosmic power, itty bitty living space!

Agentic AI has so many fascinating and exciting new realms to explore! There are still lots of the old traditional data retrieval and processing problems to solve, but also brand new classes of challenges that are only now being exposed to the light of day in the new age of LLMs. Many of the immediate issues we're grappling with today are related to context engineering, about getting LLMs the additional contextual information they need without overwhelming their limited working memory space.

The flexibility of semi-autonomous agents that have access to an array of tools (and other agents) gives rise to so many new ideas for implementing AI, it's hard to fathom the different ways we might put the pieces together. Most of the current research falls into the field of context engineering and is focused on building memory management structures that can handle and track larger amounts of context — that's because the deep-thinking problems we really want LLMs to solve present increased complexity and longer-running, multiphased thinking steps where remembering is extremely important.

A lot of the ongoing experimentation in the field is trying to find the optimal task management and tool configurations to feed the agentic maw. Each tool call in an agent's reasoning chain incurs cumulative cost, both in terms of compute to perform that tool's function as well as the impact to the limited context window. Some of the latest techniques to manage context for LLM agents have caused unintended chain effects, like "context collapse," where compressing/summarizing accumulated context for long-running tasks gets *too* lossy. The desired outcome is tools that return succinct and accurate context, without extraneous information bleeding into the precious context window memory space.

So many possibilities

We want separation of duties with flexibility to reuse tools and components, so it makes complete sense to create dedicated agentic tools for connecting to specific data sources. Each tool can specialize in querying one type of repository, one type of data stream, or even one use case. But beware: In the drive to save time, save money, and prove something is possible, there's going to be a strong temptation to use LLMs as a federation tool … but try not to — we've been down that road before! Federated query acts like a "universal translator" that converts an incoming query into the syntax that the remote repository understands and then has to somehow rationalize the results from multiple sources into a coherent response. Federation as a technique works okay at small scales, but at large scales and especially when data is multimodal, federation tries to bridge gaps that are just too wide.

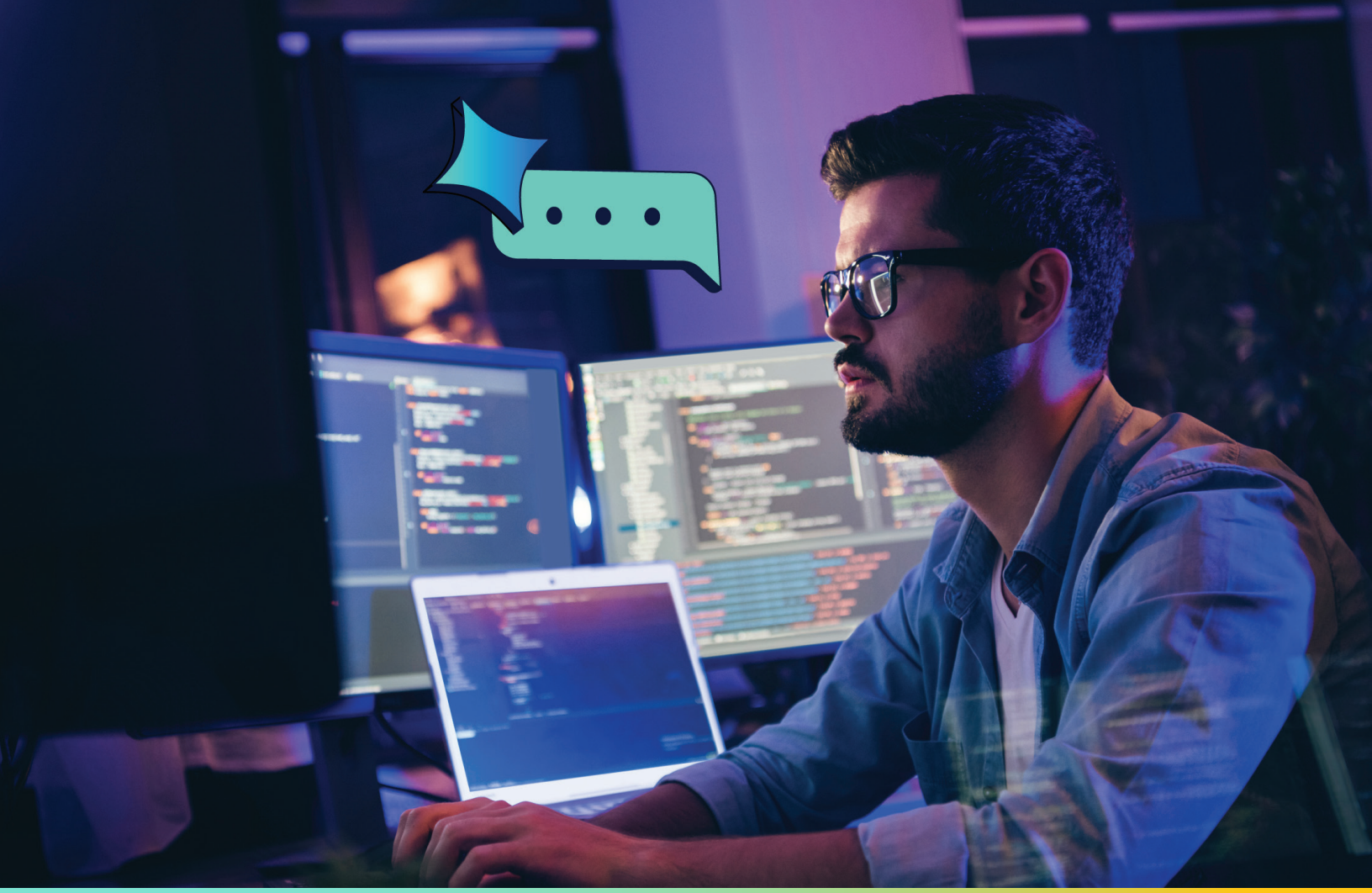
In the agentic world, the agent would be the federator and the tools (through MCP) would be the manually defined connections to disparate resources. Using dedicated tools to reach out across unconnected data sources might seem like a powerful new way to dynamically unite different data streams on a per query basis, but using tools to ask the same question to multiple sources will likely end up causing more issues than it solves. Each of those data sources is likely a different type of repository underneath, each with its own capabilities for retrieving, ranking, and securing the data within it. Those variances or "impedance mis-matches" between repositories add to the processing load, of course. They also potentially introduce conflicting information or signals, where something as seemingly innocuous as a scoring misalignment could wildly throw off the importance given to a bit of returned context and affect the relevance of the generated response in the end.

Context switching is hard for computers, too

When you send an agent out on a mission, often its first task is to find all relevant data it has access to. Just as it is with humans, if each data source the agent connects to replies with dissimilar and disaggregated responses, there will be cognitive load (though not exactly the same kind) associated with extracting the salient contextual bits from the retrieved content. That takes time and compute, and each little bit adds up in the agentic logic chain. This leads to the conclusion that, just like what's being discussed for MCP, most agentic tools should instead behave more like APIs — isolated functions with known inputs and outputs, tuned to support the needs of different kinds of agents. Heck, we're even realizing that LLMs need context for context. They do much better at connecting the semantic dots, especially when it's a task like translating natural language into structured syntax, when they have a schema to refer to.

7th inning stretch!

Now we've covered the impact LLMs have had on retrieving and querying for data, as well as how the chat window is maturing into the agentic AI experience. Let's put the two topics together and see how we can use our newfangled search and retrieval capabilities to improve our results in context engineering.



PART III

The power of hybrid search in context engineering

We've discussed both hybrid search and context engineering. Now, let's dive into how they work together for the greatest effect in supplying targeted context to RAG and agentic AI operations.

Search isn't dead, it just moved

So we've had this shift from primarily searching for context through a text box and using the information (the context) returned ourselves to construct the answers, to now using natural language to tell an agent what we want and letting it automatically research and compile the answer for us. Many in the tech world are pointing to this shift and proclaiming "search is dead," but search is still absolutely critical to agentic operations — it's just largely performed out of sight via tools now.

Previously, humans were the main arbiters of subjective relevance: Each user has their own reasons for running the search, and their personal experience colors the relative accuracy of the results. If we are to trust that agents can come to the same conclusion (or better) than we would have, we need to ensure the contextual information they have access to is as close to our subjective intent as possible. We have to engineer the context we provide LLMs to point them toward that goal.

Generating context with hybrid search retrieval

Elastic's hybrid search combines the strengths of traditional keyword-based search (syntax flexibility, keyword precision, and relevance scoring) with the semantic understanding of vector similarity search, and it offers multiple reranking techniques. This synergy (a truer usage of that word has never been found!) allows for highly relevant results, with queries that can be much more nuanced in how they target content. It's not just that you can apply subjective relevance as one of your retrieval stages; it's really that the first-stage retrieval can include relevance scoring along with all of those other modes at once.

Superior accuracy and efficiency

Using a data platform that can provide distributed search, retrieval, and reranking as your primary context retrieval engine makes a lot of sense. You're able to use advanced query syntax to add the missing component of subjective intent and filter out content that might distract from or muddy the value of the contextual information returned. You can select from any of the individual syntax options available or combine modalities into a single search that targets each type of data in the manner it understands best, and then combine or re-order them with reranking. You can have the response filtered to only include the fields and values you want, keeping extraneous data at bay. In service to agents, that targeting flexibility lets you build tools that are extremely accurate in how they retrieve context.

Context refinement (aggregations and non-content signals)

Aggregations can be especially useful in shaping the content a tool delivers to the context window. Aggregations naturally provide numerical-based facts about the shape of the contextual data returned, which makes it easier and more accurate for LLMs to reason over. Because aggregations can be hierarchically nested, it's an easy way to add multilevel detail for the LLM to generate a more nuanced understanding. Aggregations can also help with managing the context window size — you can easily reduce a query result of 100,000 documents to a few hundred tokens of aggregated insights.

Non-content signals are the inherent indicators in your data that tell you the bigger picture of what you're looking at; they're the additional characteristics of the results, things like popularity, freshness, geo-location, categories, host diversity, or price bands. These bits of information can be useful for informing the agent in how it weighs the importance of the context it has received. Some simple examples might help illustrate this best:

- **Boosting recently published and popular content:** Imagine you have a knowledge base of articles. You want to find articles relevant to a user's query, but you also want to boost articles that are both recent and have been found helpful by other users (e.g., have a high "likes" count). In this scenario, we can use a hybrid search to find relevant articles and then rerank them based on a combination of their publication date and popularity.

- **Ecommerce search with sales and stock adjustment:** In an ecommerce setting, you want to show customers products that match their search term, but you also want to promote products that are selling well and are in stock. You might also want to down-rank products with low stock to avoid customer frustration.

- **Prioritizing high-severity issues in a bug tracker:** For a software development team, when searching for issues, it's crucial to surface high-severity, high-priority, and recently updated issues first. You can use non-signals like "criticality" and "most-discussed" to weigh different factors independently, ensuring that the most critical and actively discussed issues rise to the top.

These example queries and more can be found in the accompanying Elasticsearch Labs content page.

Security enforcement

A critical advantage of using a search-powered speed layer like Elastic for context engineering is its built-in security framework. Elastic's platform ensures that context delivered to agentic and generative AI operations respects and protects sensitive privately held information through granular role-based access control (RBAC) and attribute-based access control (ABAC). This means that not only are queries handled with efficiency, but also that the results are filtered according to the specific permissions of the agent or the user initiating the request.

Agents run as the authenticated user, so security is implicitly applied through the security features built into the platform:

- **Fine-grained permissions:** Define access at the document, field, or even term level, ensuring that AI agents only receive data they are authorized to see.
- **Role-based access control (RBAC):** Assign roles to agents or users, granting access to specific datasets or functionalities based on their defined responsibilities.
- **Attribute-based access control (ABAC):** Implement dynamic access policies based on attributes of the data, the user, or the environment, allowing for highly adaptable and context-aware security.
- **Document-level security (DLS) and field-level security (FLS):** These capabilities ensure that even within a retrieved document, only authorized portions are visible, preventing sensitive information from being exposed.
- **Integration with enterprise security:** Seamlessly integrate with existing identity management systems (like LDAP, SAML, OIDC) to enforce consistent security policies across the entire organization.

By integrating these security measures directly into the context retrieval mechanism, Elastic acts as a secure gatekeeper, ensuring that AI agents operate within defined data boundaries, preventing unauthorized data exposure, and maintaining compliance with data privacy regulations. This is paramount for building trust in agentic AI systems that handle confidential or proprietary information.

As an added bonus, by using a unified data speed layer over your enterprise data sources, you alleviate the unexpected ad hoc query loads on those repositories that agentic tools would create. You get a single location to search everything in near real time and one place to apply security and governance controls.

Hybrid search-based tools

There are some core features (with more coming all the time) of the Elasticsearch Platform that turbo boost the pursuit of context engineering. The platform offers a multitude of ways to achieve things, with the flexibility to adapt, change, and expand methods as the AI ecosystem advances.

Introducing Elastic Agent Builder

Elastic Agent Builder is our first foray into the realm of agentic AI tools built to chat with the data you're already storing in Elastic. Agent Builder offers a chat interface that enables users to create and manage their own agents and tools within Kibana. It comes with built-in MCP and A2A servers, programmatic APIs, and a set of prebuilt system tools for querying and exploring Elasticsearch indices and for generating ES|QL queries from natural language. Agent Builder allows you to create custom tools that target and sculpt the contextual data returned to the agent through expressive ES|QL query syntax.

How does ES|QL perform hybrid search, you ask? The core capability is accomplished through the combination of the semantic_text field type and the FORK/FUSE commands (FUSE uses RRF by default to merge results from each fork). Here's a simple example for a fictitious product search:

```
SQL
FROM products
| FORK
  (MATCH description "high performance gaming laptop" |
EVAL search_type = "bm25"),
  (MATCH description_semantic "high performance gaming
laptop" | EVAL search_type = "semantic")
| FUSE
| LIMIT 20
| KEEP product_name, description, _score, search_type
```


The EVAL clause included with each of the FORK branches in the example above isn't strictly necessary; it's only included to demonstrate how you could track which search modality a given result was returned from.

Search templates

Let's say you want to point your own external agentic tools to your Elastic deployment. And instead of ES|QL, you want to use multistage retrievers or reuse existing DSL syntax you've developed, and you also want to be able to control the inputs the query accepts, the syntax used to execute the search, and the fields returned in the output.

Search templates allow users to define predefined structures for common search patterns, improving efficiency and consistency in retrieving data. This is particularly beneficial for agentic tools interacting with search APIs, as they help standardize boilerplate code and enable faster iteration on search logic. And if you ever need to adjust any of those factors, you just update the search template and voilà, the changes are implemented. If you're looking for an example of search templates in action with agentic tools, take a look at the Elasticsearch Labs blog MCP for intelligent search, which utilizes a search template behind a tool call from an external MCP server.

Integrated workflows

One of the most difficult things to navigate in our new agentic AI world is the non-deterministic nature of semi-autonomous, self-directed "reasoning" agents. Context engineering is a critical discipline to agentic AI: It's what determines the techniques that help narrow the possible conclusions our agent can generate down to what we know of ground truth. Even with a highly accurate and relevant context window (when we get outside the realm of numerical facts), we're still missing that bit of reassurance that the agent's response is fully repeatable and dependable.

When you run the same request to an agent multiple times, the answers might be *essentially* the same with *just that little bit* of difference in the response. That's usually fine for simple queries, maybe barely noticeable, and we can try to shape the output with context engineering techniques. But as the tasks we ask of our agents become more complex, there's more of a chance that one or more of the sub-tasks could introduce a variance that slightly changes the end result. It'll likely get worse as we begin to rely more on agent-to-agent communications, and those variances will become cumulative. This points again to the idea that the tools our agents interact with need to be very flexible and tuneable to precisely target contextual data, and that they should respond in an expected output format. It also indicates that for many use cases we have a need to direct agent and tool interactions — this is where workflows enter into the picture!

Elastic will soon have completely customizable workflows built into the core of the platform. These workflows will be able to operate with agents and tools in a bi-directional manner, so workflows will be able to call agents and tools, and agents and tools will be able to call workflows. Having these capabilities fully integrated into the same platform where all of your data lives will be transformational.

Elastic as the unified memory bank

By virtue of being a distributed data platform that's made for near real-time search, Elastic naturally performs the long-term memory functions for agentic AI systems. With the built-in Agent Builder chat experience, we also have tracking and management of the short-term memory and chat history. And because the entire platform is API-first, it's extremely easy to utilize Elastic as the platform to persist a tool's contextual output (and to be able to refer to it later) that might overwhelm the agent's context window; this technique is sometimes called "note-taking" in context engineering circles.

Having short-term and long-term memory both on the same search platform leads to a lot of intrinsic benefits: Imagine being able to use chat histories and persisted contextual responses as part of the semantic influencers to future chat interactions, or to perform threat analytics, or to create persisted data products that are automatically generated from frequently repeated tool calls … the possibilities are endless!

Conclusion

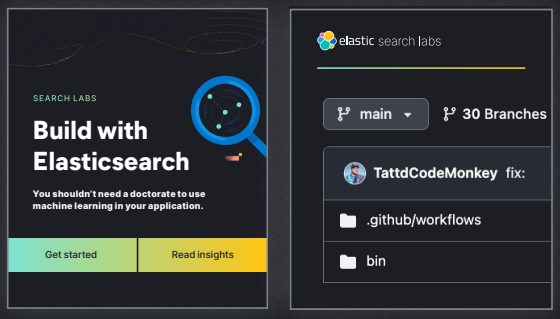
The emergence of large language models has changed the way we're able to match content and the methods we use to interrogate our data. We're rapidly shifting away from our current world, where humans perform the research, contextual consideration, and logical reasoning to answer their own questions, to one where those steps are largely automated through agentic AI. In order for us to trust the generated answers we receive, we need assurance that the agent has considered *all* of the *most relevant* information (including the factor of subjective relevance) in generating its response. Our primary method for making agentic AI trustworthy is by grounding the tools that retrieve additional context through RAG and context engineering techniques, but how those tools perform the *initial retrieval* can be critical to the accuracy of the response.

The Elasticsearch Platform provides the flexibility and advantage of hybrid search, along with several built-in features that help agentic AI in terms of accuracy, performance, and scalability; in other words, Elastic makes a fantastic platform for several aspects of context engineering. In standardizing context retrieval via a search platform, we simplify agentic tool operations on several fronts — and similar to the oxymoron "slow down to go faster," simplicity at the context generation layer means faster and more trustworthy agentic AI.

elastic search labs

Looking for more content like this, plus guides, integrations, notebooks, and example apps?

Elasticsearch Labs is your one-stop destination for learning how to build advanced search experiences with generative AI, embedding models, reranking capabilities, and more. Get guides, integrations, notebooks, and example apps — everything you need to conduct research, experiment, and start building.



Visit Elasticsearch Labs

Want to hear from the experts first?
Watch our webinar on advanced semantic search concepts with AI.

Watch now

 |