

The Modern Approaches for Representation of Semantic Information in AI Systems and Large Language Models

Igor Sinitsyn^{1,†}, Anatoliy Doroshenko^{1,2†}, Dmitry Zhora^{1,*†}, and Andrii Cherniaiev^{1,†}

¹ Institute of Software Systems, National Academy of Sciences of Ukraine, Glushkov Avenue 40, Build. 5, Kyiv, 03187, Ukraine

² National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Peremohy Ave. 37, Kyiv, 03056, Ukraine

Abstract

In recent years, we have witnessed the rapid development of software solutions built upon large language models (LLMs). It appears, the probabilistic algorithms can solve reasoning tasks and can even attempt passing the Turing test. The public perception of this phenomenon has led to widespread mystifications and overly high expectations regarding AI technologies. Meanwhile, the inherent capacity of such models to represent and reason over structured ontological and semantic information remains an area of active research. This article proposes an integrated view of AI instruments, many of which are publicly available as open-source projects. It begins with a description of hypervector representations for natural-language documents, analyzes the capabilities of hardware-based AI accelerators, demonstrates how to encode the collection of semantic triples, explores modern embedding models for multimedia information, provides a technical overview of vector databases and approximate nearest-neighbour search algorithms, examines the internal structure of large language models, explains the workflow of retrieval-augmented generation, and concludes with a concise overview of cloud-based services in the AI domain. Such broad coverage of topics is essential because these technologies are closely interconnected.

Keywords 1

natural language processing, machine learning, multi-layer perceptron, hyperdimensional computing, unbinding operator, ontology modeling, semantic triples, token embedding, document embedding, vector database, semantic search, large language model, retrieval-augmented generation, cloud computing

1. Introduction

The modern machine learning algorithms mostly fall into two categories: neural networks [1] and decision tree-based ensemble methods [4]. Ensemble methods are newer, train faster, and often generalize better on unseen data. Nevertheless, neural networks and especially multi-layer perceptron have become an integral part of current solution for large language models. This may be attributed to the fact that neural networks were studied for a longer period and had wider adoption. Also, it's worth noting that neural networks typically have a smaller memory footprint.

The key concept that bridges business domain knowledge and neural network is the notion of embedding. It is the representation of business entity or relationship as a real-valued vector of high dimensionality (or hypervector). In particular, high dimensionality allows to solve complex non-linear problems. It also enables the encoding of a fairly large amount of unstructured data.

2nd Workshop "Intelligent information technologies" UkrProg-IIT'2026 co-located with 16th International Scientific and Practical Programming Conference UkrPROG'2026, May 5-6, 2026, Kyiv, Ukraine

* Corresponding author.

† These authors contributed equally.

✉ ips2014@ukr.net (I. Sinitsyn); doroshenkoanatoliy2@gmail.com (A. Doroshenko); dmitry.zhora@gmx.com (D. Zhora); andrew.cherniaev@gmail.com (A. Cherniaiev)

ORCID 0000-0002-4120-0784 (I. Sinitsyn); 0000-0002-8435-1451 (A. Doroshenko); 0009-0006-6073-7751 (D. Zhora); 0009-0006-2731-2591 (A. Cherniaiev)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

The semantic information within large language model can be stored primarily in two locations: token embeddings and synaptic weights of neural network. Each approach has distinct advantages and limitations. In particular, the embeddings offer greater interpretability, while neural networks represent more traditional hetero-associative memory mechanism.

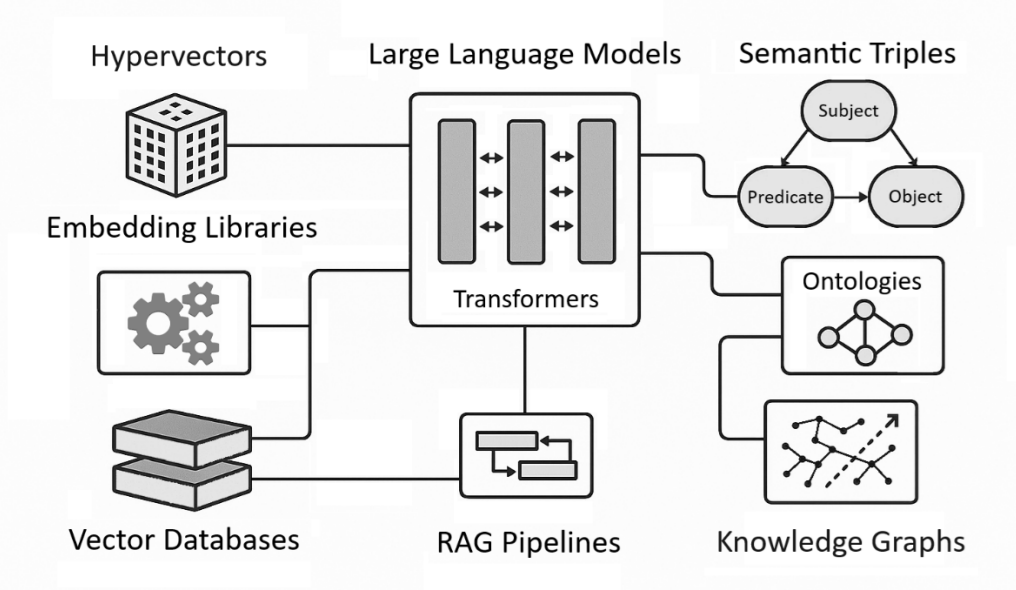


Figure 1: Main components for representation of semantic information.

As traditional LLMs require billions of documents for training, the semantic and ontology-based models benefit from documents with corresponding semantic markup. The popular formats that can represent semantic information include [5]:

- RDF, Resource Description Framework (subject–predicate–object triples);
- OWL, Web Ontology Language (built on RDF, but adds class hierarchies);
- SMW, Semantic Media Wiki (semantic annotations in wiki pages, uses RDF/OWL).

2. Hypervector data representation

The hyperdimensional computing is an emerging trend in connectionist networks that brings together various technologies with one common feature: high-dimensional representation of real-world entities [7]. This approach is inspired by studies of brain and biological systems, it is noise-tolerant, can be easily parallelized and it is efficient on low-powered devices.

The dimensionality of hypervectors ranges from hundreds to millions, and these vectors can be based on binary, bipolar or real-valued components. In recent years, the real-valued representation is more common because of computational power provided by GPU and NPU hardware. The key idea is that random hypervectors are nearly orthogonal, which allows for efficient encoding and comparison of information. Here are the main algebraic operations applied to hypervectors, along with descriptions of their purposes:

- **Bundling** or superposition (the addition followed by normalization, majority voting for binary case), Combines multiple hypervectors into one that represents a group;
- **Binding** (Hadamard product, that is element-wise multiplication or binary XOR), Provides associations or relationships between encoded entities;
- **Permutation** (typically vector rotation), Encodes sequence information like order of words in the sentence.

Another important metric is cosine similarity between vectors that is calculated as the dot product divided by product of two vector norms. It's worth noting, that for normalized vectors the cosine

similarity coincides with the dot product. Similarly, the Hamming distance is used for binary case, and this metric is not normalized.

$$\cos(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| \cdot |\mathbf{b}|} = \frac{\sum_i (a_i \cdot b_i)}{\sqrt{\sum_i a_i^2} \cdot \sqrt{\sum_i b_i^2}} \quad (1)$$

As with ordinary real numbers, the bundling and binding operations on vectors are associative, commutative, and binding distributes over superposition (or bundling) proportionally. The space of dimensionality d can contain about 2^d nearly orthogonal vectors. Each such vector can be degraded up to 30% to remain the most similar to original one than any other vector [8].

Here's an example of how to encode the phrase "The cat sat on the mat." using the hypervector representation. First, it is needed to generate random vectors representing the sentence parts and meaningful words. The most typical approach is to use normal distribution with zero mean and unit vector norm for this purpose.

- **SUBJECT = encode(subject)**
- **VERB = encode(verb)**
- **PREP = encode(preposition)**
- **OBJECT = encode(object)**

- **CAT = encode("cat")**
- **SAT = encode("sat")**
- **ON = encode("on")**
- **MAT = encode("mat")**

The role of the word in the sentence can be defined using the binding operation.

- **CAT $\otimes$ SUBJECT**
- **SAT $\otimes$ VERB**
- **ON $\otimes$ PREP**
- **MAT $\otimes$ OBJECT**

Finally, the sentence can be encoded as a superposition of obtained vectors.

$$\mathbf{SENTENCE} = (\mathbf{CAT} \otimes \mathbf{SUBJECT}) + (\mathbf{SAT} \otimes \mathbf{VERB}) + (\mathbf{ON} \otimes \mathbf{PREP}) + (\mathbf{MAT} \otimes \mathbf{OBJECT})$$

The main application areas for hypervector embeddings include classification tasks, associative search algorithms, the representation of symbolic and numerical information for usage in machine learning algorithms, such as large language models.

Not all mathematical operations used in hyperdimensional computing are employed in large language models. Typically, bundling and dot product operations are sufficient. There are LLMs that use sentence-level representations of input text. However, as of today the most well-known products like ChatGPT rely on sequences of token-level representations.

3. Unbinding operation on hypervector

The hypervector data representation allows to perform logical operations on encoded data. In particular, the unbinding operation allows the retrieval of sentence components. For example, to determine "who sat on the mat" the unbinding operator can be applied to sentence representation with the subject as an argument.

$$\mathbf{WHO} = \mathbf{SENTENCE} \oslash \mathbf{SUBJECT} = \mathbf{SENTENCE} \otimes (\mathbf{SUBJECT})^{-1}$$

If the hypervector representing **SUBJECT** consists of elements s_i , then hypervector representing $(\mathbf{SUBJECT})^{-1}$ consists of elements $1/s_i$. In practice, some small number is added to denominator to avoid the division by zero. So, if the binding operation is defined as element-wise multiplication,

then unbinding operation is essentially element-wise division. Now, the expression for **SENTENCE** obtained previously can be substituted into the expression above.

$$\begin{aligned}
\text{WHO} &= \text{SENTENCE} \oslash \text{SUBJECT} = \text{SENTENCE} \otimes (\text{SUBJECT})^{-1} = \\
&= ((\text{CAT} \otimes \text{SUBJECT}) + (\text{SAT} \otimes \text{VERB}) + (\text{ON} \otimes \text{PREP}) + (\text{MAT} \otimes \text{OBJECT})) \otimes (\text{SUBJECT})^{-1} = \\
&= (\text{CAT} \otimes \text{SUBJECT} \otimes (\text{SUBJECT})^{-1}) + \\
&((\text{SAT} \otimes \text{VERB}) + (\text{ON} \otimes \text{PREP}) + (\text{MAT} \otimes \text{OBJECT})) \otimes (\text{SUBJECT})^{-1} = \text{CAT} + \text{NOISE} \approx \text{CAT}
\end{aligned}$$

As for expression $((\text{SAT} \otimes \text{VERB}) + (\text{ON} \otimes \text{PREP}) + (\text{MAT} \otimes \text{OBJECT})) \otimes (\text{SUBJECT})^{-1}$, all vectors that constitute it are independently generated and orthogonal to the vector **CAT**. So, this term can be interpreted as noise and entity **CAT** should prevail during decoding or clean-up operation.

This can be demonstrated by the following dot product calculations, which take into account the usage of normalized vectors. Also, when the result of unbinding operation is compared with a different subject, the vector correlation is not present, and the dot product is close to zero.

$$\begin{aligned}
&(\text{SENTENCE} \oslash \text{SUBJECT}) \cdot \text{CAT} = \\
&= (\text{CAT} + ((\text{SAT} \otimes \text{VERB}) + (\text{ON} \otimes \text{PREP}) + (\text{MAT} \otimes \text{OBJECT})) \otimes (\text{SUBJECT})^{-1}) \cdot \text{CAT} \approx \\
&\approx (\text{CAT} \cdot \text{CAT}) + 0 = 1 \\
&(\text{SENTENCE} \oslash \text{SUBJECT}) \cdot \text{DOG} = \\
&= (\text{CAT} + ((\text{SAT} \otimes \text{VERB}) + (\text{ON} \otimes \text{PREP}) + (\text{MAT} \otimes \text{OBJECT})) \otimes (\text{SUBJECT})^{-1}) \cdot \text{DOG} \approx \\
&\approx 0 + 0 = 0
\end{aligned}$$

As an example, the hypervector with a dimensionality of 1024 can store tens of sentences with 99% recall accuracy. The storage capacity depends greatly on sentence content, word and phrase correlations. At the same time, it increases linearly with the vector dimensionality.

4. Alternative vector architectures

The element-wise multiplication scheme described in previous two sections is often referenced as MAP-C [9]. At the same time, it is not the only widely used vector symbolic architecture (VSA). Table 1 below represents the alternative schemes that operate over diverse element spaces, feature distinct initial distribution of elements, and employ different similarity metrics.

Table 1

Element type, distribution and similarity characteristics for vector symbolic architectures.

VSA Code	VSA Name	Elements	Distribution	Similarity
MAP-C	Multiply-Add-Permute on Continuous Elements	Real	$U(-1,1)$	Cosine
MAP-B	Multiply-Add-Permute on Bipolar Elements	Bipolar	$B(0.5) \cdot 2 - 1$	Cosine
MAP-I	Multiply-Add-Permute on Integer Elements	Integer	$B(0.5) \cdot 2 - 1$	Cosine
BSC	Binary Spatter Code	Binary	$B(0.5)$	Hamming
BSDC-CDT	Binary Sparse Distributed Coding with Context Dependent Thinning	Binary	$B(p \ll 1)$	Overlap
BSDC-S	Binary Sparse Distributed Coding with Shifting and Optional Thinning	Binary	$B(p \ll 1)$	Overlap
BSDC-SEG	Binary Sparse Distributed Coding with Segment Shifting	Binary	$B(p \ll 1)$	Overlap
HRR	Holographic Reduced Representation	Real	$N(0, 1/D)$	Cosine
FHRR	Fourier Holographic Reduced Representation	Complex	$e^{iU(-\pi, \pi]}$	Angle
VTB	Vector Derived Transformation Binding	Real	$N(0, 1/D)$	Cosine
MBAT	Matrix Binding of Additive Terms	Real	$N(0, 1/D)$	Cosine

Here $U(a, b)$ denotes a uniform distribution over the interval $[a, b]$, $N(\mu, \delta^2)$ is a normal distribution with mean μ and standard deviation δ , $B(p)$ is a Bernoulli distribution with probability p , and D is the hypervector length. In the FHRR scheme, the possible set of elements belongs to the unit circle in the complex plane, and the angle is uniformly distributed.

Each vector symbolic architecture has its own characteristics. The computational complexity of the bundling operation is $O(D)$ across all architectures. In contrast, the complexity of the binding operation ranges from $O(D)$ to $O(D^2)$. The commutativity and associativity properties of bundling, binding and unbinding operations are not always guaranteed.

The bundling capacity and the ability to store bound key-value pairs are shown in Table 2. It is worth noting that binary schemes use only one bit per element, while popular FHRR architecture employs two real numbers per element, which may require between 8 and 128 bits.

Table 2

Bundling and binding pair capacities for $D = 1024$ and 99% retrieval accuracy.

VSA Code	Bundling	Binding	Unbinding	Binding Complexity	Bundling Capacity	Pair Capacity
MAP-C	Addition with Normalization	Multiplication	Division	$O(D)$	~23	~24
MAP-B	Addition with Threshold	Multiplication	Multiplication	$O(D)$	~18	~19
MAP-I	Addition	Multiplication	Multiplication	$O(D)$	~32	~31
BSC	Addition with Threshold	XOR	XOR	$O(D)$	~20	~20
BSDC-CDT	Disjunction with Opt. Thinning	XOR	XOR	$O(D)$	~47	~28
BSDC-S	Disjunction with Opt. Thinning	Shifting	Shifting	$O(D)$	~47	~28
BSDC-SEG	Disjunction with Opt. Thinning	Segment Shifting	Segment Shifting	$O(D)$	~47	~36
HRR	Addition with Normalization	Circular Convolution	Circular Correlation	$O(D^2)$	~30	~30
FHRR	Addition of Angles	Fourier Multiplication	Fourier Division	$O(D \log(D))$	~45	~45
VTB	Addition with Normalization	VTB	Transpose VTB	$O(D^2)$	~30	~29
MBAT	Addition with Normalization	Matrix Multiplication	Inverse Matrix Multiplication	$O(D^2)$	~30	~27

The vector symbolic architectures can represent complex data structures such as sets, stacks, histograms, sequences, dictionaries, trees, graphs, finite-state automata, context-free grammars and even JSON documents [10]. Also, it was demonstrated that these computational schemes have the potential to implement features of a Turing-complete system.

5. Semantic triple data encoding

There are several complementary approaches for representation of knowledge as an ontology, and it is worth noting that all of them combine three entities with distinct roles.

- subject-predicate-object, RDF triples or semantic graphs
- head-relation-tail, concept-level knowledge representation
- entity-attribute-value, heterogeneous data modeling

Despite their simple form, semantic triples can represent highly complex data structures. One hypothesis suggests that arbitrary natural language text can be represented as a collection of nested RDF triples. Consider the following sentence: “The cat sat on the mat because it is tired”. Here is how this knowledge can be represented using RDF triples.

((cat, sat, mat), because, (cat, is, tired))

Thus, the ability to encode ontology triples into hypervectors should enable hyperdimensional computing for knowledge graphs and databases.

Taking into account the analogy with sentence example considered earlier, the representation of a single triple can be calculated using the role hypervectors.

$$\mathbf{TRIPLE} = (\mathbf{CAT} \otimes \mathbf{SUBJECT}) + (\mathbf{SAT} \otimes \mathbf{PREDICATE}) + (\mathbf{MAT} \otimes \mathbf{OBJECT})$$

As discussed previously, all six vectors in the formula above are randomly generated, fixed, and quasi-orthogonal. Given the hypervector representing a triple, the components can be decoded by unbinding the corresponding roles, the example below assumes MAP-C scheme.

$$\begin{aligned} \mathbf{WHAT} &= \mathbf{TRIPLE} \oslash \mathbf{OBJECT} = \mathbf{TRIPLE} \otimes (\mathbf{OBJECT})^{-1} = \\ &= ((\mathbf{CAT} \otimes \mathbf{SUBJECT}) + (\mathbf{SAT} \otimes \mathbf{PREDICATE}) + (\mathbf{MAT} \otimes \mathbf{OBJECT})) \otimes (\mathbf{OBJECT})^{-1} = \\ &= ((\mathbf{CAT} \otimes \mathbf{SUBJECT}) + (\mathbf{SAT} \otimes \mathbf{PREDICATE})) \otimes (\mathbf{OBJECT})^{-1} + \\ &+ (\mathbf{MAT} \otimes \mathbf{OBJECT} \otimes (\mathbf{OBJECT})^{-1}) = \mathbf{NOISE} + \mathbf{MAT} \approx \mathbf{MAT} \end{aligned}$$

The standard approach for combining a group of hypervectors is to use bundling. However, in this case, bundling has already been used to build the triple. Therefore, to enable decomposition of the collection of triples, the permutation operation has to be applied. Here, $\mathbf{P}^i$ denotes the circular rotation of the hypervector by i positions.

$$\mathbf{KNOWLEDGE_BASE} = \mathbf{TRIPLE}_1 + \mathbf{P}^1(\mathbf{TRIPLE}_2) + \dots + \mathbf{P}^{N-1}(\mathbf{TRIPLE}_N)$$

The alternative approach is to use a permutation operation when building the triple.

$$\mathbf{TRIPLE} = \mathbf{CAT} \otimes \mathbf{P}^1(\mathbf{SAT}) \otimes \mathbf{P}^2(\mathbf{MAT})$$

$$\mathbf{WHAT} = \mathbf{P}^{-2}(\mathbf{TRIPLE}) = \mathbf{P}^{-2}(\mathbf{CAT}) \otimes \mathbf{P}^{-1}(\mathbf{SAT}) \otimes \mathbf{MAT} = \mathbf{NOISE} \otimes \mathbf{MAT}$$

In this case, the collection can be represented as follows:

$$\mathbf{KNOWLEDGE_BASE} = \mathbf{TRIPLE}_1 + \mathbf{TRIPLE}_2 + \dots + \mathbf{TRIPLE}_N$$

The capacity characteristics of both encoding schemes have yet to be studied. The first approach is more suitable for large collections but requires an iterative loop for search. The second scheme is faster but it is more vulnerable to collisions during decoding.

In general, there are more advanced embedding schemes for ontologies that take into account context and semantic correlations between entities. One of the common ideas is to use a two-layer perceptron trained on ontology data and take the hidden layer representations as embeddings for the original entities. Records with discrete data are converted into perceptron inputs and outputs using one-hot encoding. Here are some popular methods that are actively researched today [15]: Word2Vec, RDF2Vec, OWL2Vec, OPA2Vec, TransE, TransC, ComplEx, HoLE, KG-BERT, Geometric Modeling, and Graph Neural Networks.

6. Capabilities of modern hardware

Table 3 illustrates the diversity of numeric representations supported by modern hardware AI accelerators, ranging from low-precision integer formats (int4, int8) to high-precision floating-point formats (fp64). This heterogeneity is critical for hypervector computations, where operations often require a careful balance between precision and throughput. For instance, int8 and Google-introduced bf16 formats dominate in inference workloads due to their efficiency. The fp16 and fp32

formats remain essential for training and symbolic reasoning tasks, while fp64 is better suited for modeling tasks that demand numerical stability.

The modern trend is that newer hardware generations increasingly support smaller numerical representations to improve arithmetic throughput and reduce memory requirements by lowering the precision where full accuracy is not necessary. The formats such as fp16, bf16, int8, and even lower-bit integers are now first-class citizens in modern CPUs, GPUs, and specialized accelerators. This shift enables faster training and inference for machine learning workloads while reducing power consumption. As a result, systems can process larger models and datasets within the same hardware and energy constraints.

Table 3

Hypervector element types implemented in workstation-grade AI accelerators.

Processing Unit	int4	int8	int16	int32	int64	fp4	fp8	bf16	fp16	tf32	fp32	fp64
NVidia Jetson AGX Orin	+	+	+	+	+	-	+	+	+	+	+	+
Core Ultra 9 285K iGPU	-	+	+	+	+	-	-	-	+	-	+	+
Core Ultra 9 285K NPU	+	+	+	+	-	-	-	+	+	-	+	-
AMD Ryzen 8700G iGPU	-	+	+	+	+	-	-	-	+	-	+	+
AMD Ryzen 8700G NPU	+	+	+	-	-	-	-	+	+	-	+	-
Threadripper Pro 7995WX	-	+	+	+	+	-	-	-	+	-	+	+
Apple M4 iGPU	-	+	+/-	+	-/+	-	-	+/-	+	-	+	-/+
Apple M4 NPU	-	+	-/+	-/+	-	-	-	-	+	-	-/+	-
Apple M5 iGPU	-	+	+/-	+	-/+	-	-	+/-	+	-	+	-/+
Apple M5 NPU	-	+	-/+	-/+	-	-	-	-	+	-	-/+	-
Snapdragon X Elite iGPU	+	+	+	+	+	-	+	+	+	-	+	-
Snapdragon X Elite NPU	+	+	+	-	-	-	+	+	+	-	+	-
Intel Arc A770	+	+	+	+	+	-	-	+	+	-	+	-/+
Intel Gaudi 3 AI Accelerator	-	+	-	-	-	-	+	+	+	+	+	-
AMD Radeon RX 7900 XTX	-	+	+	+	+	-	-	+	+	-	+	+
AMD Radeon Pro W7900	-	+	+	+	+	-	+	+	+	-	+	+
AMD Radeon RX 9070 XT	+	+	+	+	+	-	+	+	+	-	+	-/+
AMD Radeon AI Pro R9700	+	+	+	+	+	-	+	+	+	-	+	-/+
AMD Instinct MI300X	+	+	-	+	+	-	+	+	+	+	+	+
AMD Instinct MI300A	+	+	-	+	+	-	+	+	+	+	+	+
NVidia RTX 3090	-/+	+	+	+	+	-	-	+	+	+	+	+
NVidia RTX 4090	+	+	+	+	+	-	+	+	+	+	+	+
NVidia RTX 5090	+	+	+	+	+	+	+	+	+	+	+	+
NVidia RTX 6000 Ada	+	+	+	+	-/+	-	+	+	+	+	+	+
NVidia RTX Pro 6000	+	+	+	+	-/+	-	+	+	+	+	+	+
NVidia H100 GPU	+	+	+/-	+	+	-	+	+	+	+	+	+
NVidia H200 GPU	+	+	+/-	+	+	-	+	+	+	+	+	+
NVidia B100 GPU	+	+	+/-	+	+	+	+	+	+	+	+	+
NVidia B200 GPU	+	+	+/-	+	+	+	+	+	+	+	+	+
AWS Trainium2	-	+	-	-	-	-	+	+	+	+	+	-
AWS Trainium3	+	+	+	-	-	+/-	+	+	+	+	+	-
Google TPU v6	-	+	-	+/-	-	-	-	+	+	-	+	-
Google TPU v7	-	+	-	+/-	-	-	+	+	+	-	+	-
Microsoft Maia 100	+	+	-	+/-	-	-	-	+	-	-	+	-
Microsoft Maia 200	-	+	-	+/-	-	+	+	+	+	-	+	-

As discussed, modern AI accelerators can efficiently perform hypervector operations such as binding, bundling, and permutation [11]. Similarly, they can handle distributed representations for

vast ontologies. The growing hardware capabilities like increased parallelism, memory capacity, and bandwidth have numerous implications:

- symbolic computations are no longer restricted to be purely discrete,
- hypervector reasoning can scale horizontally within a single node,
- larger memory footprints enable richer symbolic spaces,
- in-memory, whole-ontology training becomes practical,
- embeddings can now better approximate symbolic semantics,
- massively parallel execution enables fine-grained symbolic search,
- ontology embeddings become living artifacts that track knowledge drift.

In summary, symbolic systems can now prioritize expressivity and correctness over aggressive minimization strategies that distort semantics. Ontologies are no longer interpreted as background schemas, they become active control structures for inference and retrieval. Symbolic computations increasingly run on the same hardware stack as embeddings, so model embeddings stop being lossy approximations and instead become operational semantics.

The large language models increasingly rely on retrieval-augmented generation pipelines, in which symbolic knowledge graphs and dense embeddings coexist. Hypervector representations offer a bridge between these paradigms: ontology triples can be embedded into hypervectors and stored in vector databases, enabling semantic queries that align with LLM token embeddings. In other words, current hardware trends favor hybrid symbolic–numeric pipelines.

7. Production-grade vector databases

A vector database is the solution to store the hypervector representations of documents, such as text, code, sensor data, image, audio, and video. Rather than storing the raw documents themselves, it functions as high-performance index designed primarily for approximate similarity search tasks. The vector database can be conceptually represented as a table with fields described below [16].

Table 4
Record structure for the document in a vector database.

Field	Type	Description
Id	GUID, integer or string	Unique document key or identifier
Vector	Hypervector	Document embedding that preserves semantic similarity
Payload	JSON document (typically)	Arbitrary metadata with a reference to original document

The large documents are often split into chunks. In this case, the payload field should contain the corresponding chunk number. In addition to the document reference (usually URI or file path) and chunk number (if applicable) the payload may contain the original text chunk, document id for external database lookups, timestamps, keywords, categories, and tags. It is worth noting that the hypervector representation of a document is lossy and it cannot be used to precisely reconstruct the original document content.

Table 5
Modern landscape of vector search solutions and integrations.

Category	Examples
Managed and hosted databases	Pinecone, Redis (Cloud), Milvus (Cloud)
Open-source databases	Milvus, Qdrant, Weaviate, Vald, OpenSearch
Relational databases with vector search	Postgres (pgvector), SQL Server
Approximate nearest neighbor libraries	FAISS, HNSWlib, Annoy, Sptag
Emerging solutions with analytics	LanceDB, DeepLake

Vector databases address different usage patterns and can be classified into several categories according to their hosting and integration capabilities, as summarized in Table 5. Cloud-hosted and managed databases offer reduced maintenance effort and greater flexibility, particularly when databases need to be quickly provisioned or decommissioned. Open-source databases are well suited for self-hosting scenarios and for research purposes. Vector search functionality can be an essential addition when organization already relies on an established relational database solution. Many hypervector processing algorithms are implemented as standalone libraries, which can be integrated directly into the application layer or used as part of a custom database solution that provides its own hosting. Finally, a recent emerging trend in this area is the concept of a data lake for hypervectors, offering analytics and version control capabilities.

Table 6

Hypervector databases with integration and scalability characteristics.

Vector Database	Strengths and Features	GPU Support	License Type
Elasticsearch	Mature search engine; hybrid text+vector search; distributed; strong ecosystem	Limited	SSPL 1.0 and Elastic 2.0
OpenSearch	Apache-licensed fork of Elasticsearch; hybrid search; AWS integrations	Limited	Apache 2.0
Redis	In-memory hybrid vector search; semantic caching; vector sets; high-performance real-time use cases	No	BSD/MIT, Proprietary
Postgres (pgvector)	Relational database integration; simple setup; suitable for small-medium datasets	No	~BSD/MIT
SQL Server	Enterprise features; integrated search extension; security	No	Proprietary
Milvus	Multi-tenant scalable distributed vector database; GPU acceleration; hybrid search; columnar storage	Yes	Apache 2.0
Qdrant	Rust-based; efficient ANN search; rich filtering; quantization	No	Apache 2.0
Pinecone	Fully managed; low latency; auto-scaling; sparse+dense search	No	Proprietary
Weaviate	GraphQL API driven; built-in vectorizers; hybrid search; modular system; backups	Optional	BSD 3-Clause
Vespa	High-performance hybrid search; real-time indexing	Optional	Apache 2.0
Vald	Cloud-native scalable ANN search; CNCF ecosystem	Optional	Apache 2.0
Marqo	Multimodal vector and text search; user-friendly API	No	Apache 2.0
LanceDB	Columnar analytics; persistent storage; pipeline integration	No	Apache 2.0
DeepLake	Vector and tensor storage; multimodal datalake; analytics; data version control, branches, commits and rollbacks	Optional	Apache 2.0
FAISS	High-performance ANN library; CPU/GPU optimized; very fast	Yes	MIT License
HNSWlib	Lightweight ANN search library; strong quality/latency; widely embedded in databases	No	Apache 2.0
Chroma	Developer-friendly; LLM features; vector+full-text+regex	No	Apache 2.0
Annoy	Lightweight ANN library; simple file-based search; low memory requirements	No	Apache 2.0
Sptag	Microsoft open ANN search; supports large datasets	No	MIT

There is an important concept in the field of vector databases known as native hyperscaling. This concept primarily applies to managed cloud services and large-scale distributed systems. A database is considered to provide native hyperscaling when the following conditions are met:

- designed for horizontal scaling from day one,
- implements automatic sharding and replication,

- enables elastic scale-out and scale-in without application changes,
- handles billions of vectors across many nodes,
- requires no manual partitioning or do-it-yourself orchestration.

The following solutions can be considered to provide native hyperscaling: Pinecone, Weaviate, Milvus, Vespa, Vald, and Marqo. These systems are capable of handling billions of hypervectors. Elasticsearch, OpenSearch, and Redis offer lower capacity, typically measured in tens of millions, even though they support native sharding. Relational databases, such as Postgres and SQL Server, generally handle only millions of vectors, as they were not designed for distributed storage.

A vector database primarily involves two core operations: data encoding and semantic search. During the encoding phase, an embedding model converts raw data into dense numerical vectors. By mapping similar inputs close to each other in vector space, these models enable many powerful capabilities such as search, clustering, recommendations, and retrieval-augmented generation [17].

There are several approaches for implementation of search operations in a vector database. The most widely used method is Approximate Nearest Neighbor (ANN) search. The approximate search is important because it enables significantly faster performance compared to the classic k-Nearest Neighbors (KNN) algorithm. Table 7 presents a comparison of key features of these algorithms.

Table 7

Comparison of exact and approximate hypervector search features.

Algorithm Feature	KNN, Exact Search	ANN, Approximate Search
Accuracy	100%, finds the absolute closest vector	High, finds close enough matches
Speed	Quite slow, scales poorly with data	Lightning fast, scales to billions
Implementation	Compares query vector to every point	Uses smart indexing to skip data

Here's what a typical vector database usage workflow looks like:

- find the approximate nearest neighbour using the search query,
- locate the corresponding vector in database using the returned Id,
- retrieve the payload and extract the external document Id,
- fetch the full document blob from external document database.

It is quite common for ANN search to return multiple approximate neighbors, each with its own similarity or distance score. When several suitable results are available, the application can choose among them using the scores provided. This approach helps avoid strictly deterministic responses and makes hypervector-based reasoning behave more similarly to natural intelligence.

Vector databases rely on specialized ANN indexing methods to enable efficient similarity search at scale. Instead of performing exhaustive search, these algorithms organize hypervectors in a way that significantly reduces computational complexity. The most widely adopted approaches include the following presented in Table 8. The notion of high recall denotes the proportion of true nearest neighbors recovered by an approximate search algorithm [18].

Table 8

Effective indexing methods for approximate nearest neighbor search.

Method	Full Name	Indexing Method Description
HNSW	Hierarchical Navigable Small World	A multi-layer graph structure enabling efficient long-range navigation in upper layers and fine-grained search in lower layers. HNSW provides strong recall-latency performance and is considered the leading practical approach available today.
IVF	Inverted File Index	A clustering-based approach that partitions the hyperspace into Voronoi cells. At query time, only the nearest clusters are searched,

		significantly reducing the set of candidate vectors. IVF is commonly combined with quantization to optimize memory usage.
PQ	Product Quantization	A compression technique that splits vectors into smaller subspaces and quantizes each subspace separately. This approach allows fast distance calculations while keeping memory usage low, though it introduces a moderate amount of quantization error.
ScaNN	Scalable Nearest Neighbors	A hybrid method combining tree-based space partitioning, anisotropic vector quantization, and optimized distance computation. It is designed for high recall with low latency, particularly for large-scale embeddings used in retrieval and recommendation systems.

The popular VectorDBBench benchmark allows the estimation of vector database performance for datasets containing one million records at a monthly cost of \$1'000. According to the results presented in Figure 2, the fastest solutions are ZillizCloud (Milvus), OpenSearch and ElasticCloud.

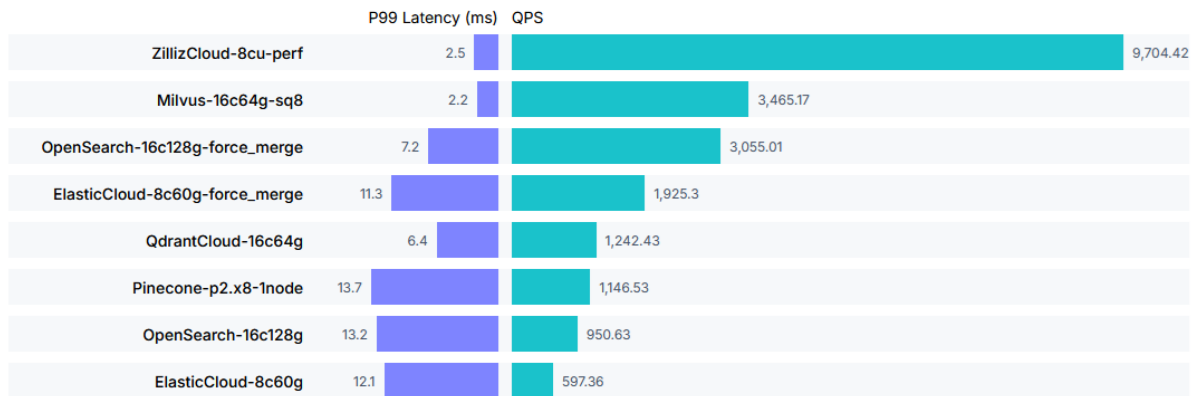


Figure 2: Vector search latency and the number of queries per second.

8. Text and multimedia data encoding

It is a non-trivial task to convert document text into a hypervector in a way that ensures similar texts receive similar vector representations. At the same time, numerous libraries already exist that successfully perform this conversion [19]. Some of them are open-source and can be used locally, while commercial solutions like OpenAI or Cohere expose their embedding models via REST API. Although modern embedding algorithms are generally language-agnostic, most available libraries are implemented for the Python ecosystem.

The history of natural language processing began in the twentieth century with algorithms such as Bag-of-Words (BoW) and N-grams. A key limitation of these early approaches is that they do not capture semantic relationships, they ignore broader linguistic context, and they cannot handle polysemy – words that have multiple meanings. Word2Vec, introduced later, provides noticeable semantic improvements. It is a shallow neural embedding model that learns static word vectors, uses hidden layer representation of linear perceptron and has two main variants: CBOW (Continuous Bag-of-Words), which predicts a target word from its surrounding context, and Skip-gram, which predicts context words from a target word.

Table 9

The popular embedding models that preserve semantic similarity.

Algorithm Name	Full Name	Supported Data	Strengths and Features	License Type
----------------	-----------	----------------	------------------------	--------------

BERT	Bidirectional Encoder Representations from Transformers	text	Context-aware embeddings using bidirectional attention; strong performance on NLP tasks	Apache 2.0
SBERT	Sentence-BERT	text	Optimized for sentence and paragraph embeddings; efficient semantic similarity	Apache 2.0
RoBERTa	Robustly Optimized BERT Approach	text	Improved BERT training strategy; stronger representations with larger datasets; greater efficiency	MIT License
E5 Models	Embedding Models from Bidirectional Encoder Representations	text	Designed for retrieval tasks; instruction-style inputs improve search and ranking; strong open-source alternative to OpenAI	Apache 2.0
Instructor	Instructor Embeddings	text	Instruction-guided embeddings; adapts representations for tasks	Apache 2.0
Cohere	Cohere Embedding Models	text	High-quality commercial embeddings; strong multilingual and retrieval performance	Proprietary
GTE	General Text Embeddings	text	Lightweight and efficient; good balance of speed and accuracy	Apache 2.0
CLIP	Contrastive Language-Image Pretraining	text+ image+ [video]	Maps text and images into the same vector space; enables cross-modal search; video extension is under development	MIT License
OpenAI	Open Artificial Intelligence (company name)	text+ limited	High-quality text embedding models trained on diverse corpora; multilingual support	Proprietary
wav2vec	wav2vec 2.0	multimodal audio	Self-supervised speech embeddings; strong performance in ASR and speech tasks	MIT License
CodeBERT	Code BERT	code+ text	Joint embeddings for programming and natural languages	MIT License
SciBERT	Scientific BERT	scientific text	Pretrained on scientific papers; strong for academic domain	Apache 2.0
BioBERT	Biomedical BERT	biomedical text	Domain-specific embeddings for biomedical and clinical text	Apache 2.0
DINO	Self-Distillation with No Labels	image	Self-supervised visual embeddings; strong for clustering and transfer learning	Apache 2.0
ViT	Vision Transformer	image	Transformer-based image embeddings; scalable and effective for vision tasks	Apache 2.0

The modern algorithms that provide embeddings for textual and multimedia information are listed in Table 9. In addition to plain text, these models support other document types such as code, images, and audio. Also, it is worth noting the concept of multimodal embeddings, which combine information from multiple data types (modalities) into a single shared embedding space [20].

The majority of popular embedding algorithms, beginning with BERT, rely on the Transformer architecture [21]. This approach offers faster training compared with recurrent or convolutional neural networks typically used for sequence processing. A transformer block consists of two main sublayers: multi-head self-attention and position-wise feed-forward neural network – typically a multi-layer perceptron with a single hidden layer and GELU activation function.

In summary, transformers are currently the most effective and widely adopted architecture for capturing semantic information in text. For example, they can efficiently distinguish between the concepts of a ‘river bank’ and an ‘investment bank’, they produce highly correlated embeddings for tokens like ‘crocodile’ and ‘alligator’.

Several embedding models for video are also under active development, though the claims in this area are still largely forward-looking. Examples include CLIP-based video models (X-CLIP, VideoCLIP, InternVideo) and transformer-based architectures such as TimeSformer, Video Swin Transformer, SlowFast, and VideoMAE, which remain experimental and research-focused.

9. Characteristics of popular LLMs

While the development of natural language processing (NLP) algorithms began in the 1950s, a revolutionary breakthrough occurred only in recent years with the introduction of large language model architectures. Table 10 below presents the current landscape of commercial and open-source solutions in this domain and summarizes the essential quantitative characteristics for each model. It is worth noting that Copilot is not listed here, as it is using GPT-4 and GPT-5 architectures.

Table 10

The comparison of internal characteristics of major large language models.

Language Model	Release Year	Number of Parameters	Embedding Size	Transformer Layers	Context Window	Attention Heads	Training Tokens
GPT-1	2018	117M	768	12	512	12	~5B
GPT-2	2019	1.5B	1024	48	1024	25	~40B
GPT-3 6.7B	2020	6.7B	4096	32	2048	32	~300B
GPT-3 13B	2020	13B	5120	40	2048	40	~300B
GPT-3 175B	2020	175B	12288	96	2048	96	~300B
GPT-4	2023	500B+	12288+	>100	~128K	n/a	10T+
GPT-5	2025	2.0T+	16384+	n/a	400K	n/a	~20T
GPT-5.1	2025	2.2T+	16384	n/a	~128K	n/a	~20T
GPT-5.2	2025	3.5T+	24576+	n/a	~128K	n/a	~20T
Claude 3.7	2025	175B+	~12288	24	~200K	32	n/a
Claude 4.0	2025	400B+	16384+	n/a	200K+	n/a	n/a
Gemini 2.5	2025	100-500B	12288	n/a	1-2M	n/a	n/a
Gemini 3.0	2025	1T+	12288+	n/a	~2M	n/a	n/a
LLaMa 3.3	2024	~70B	8192	~80	128K	64	~15T
LLaMa 4 Scout	2025	~109B	5120+	48	~10M	~128	~30T
Maverick	2025	~400B	12288	120	~1M	64-128	~30T
Mistral 7B	2024	7.3B	4096	32+	~32K	32	2T+
Mistral Large	2024	675B	12288	88	~128K	96	5T+
Qwen 3.5	2025	397B	2048	n/a	~256K	32	~18T
DeepSeek V3	2024	~671B	7168	~61	64-128K	128	14.8T
DeepSeek 3.2	2026	~685B	7168	~61	~128K	~128	~15T
GLM-4.7	2026	355B	n/a	~92	~200K	64-96	~12T
GLM-5	2026	744B	n/a	~78	~200K	96-128	28.5T
Grok-3	2025	2.7T	n/a	n/a	128K+	n/a	n/a
Grok-4	2026	1.7T	n/a	n/a	~256K	n/a	n/a
Kimi K2.5	2026	1.04T	7168	n/a	~256K	n/a	~15T

Many characteristics, especially for newer models, are provided only approximately because companies often choose not to disclose exact specifications in order to preserve their competitive advantage. Some characteristics are not published at all and are difficult to estimate.

The number of parameters represents the total count of weights, biases, and other coefficients used in the model. Typically, the more parameters a model has, the more powerful it is. However, this increased capability comes with higher compute and memory requirements.

The embedding size is the dimensionality of a vector representing a token. It defines the width of the data bus when information is passed from one transformer layer to the next. Expanding the embedding dimension improves the representational capacity at the expense of larger model and higher compute requirements per layer.

The context window represents the maximum number of tokens that can precede the current one. In other words, it defines the span of tokens the model can process at once. In contrast to the BERT family of embedding models, LLMs cannot attend to future tokens during generation.

The self-attention mechanism within a transformer layer is designed to capture dependencies between tokens such as subject-verb, verb-object, or adjective relationships, pronoun coreference, etc. Multiple self-attention blocks allow the model to learn more complex logical, semantic, and syntactic patterns. Increasing the number of attention heads generally improves model capacity, though this comes at the cost of higher compute and memory requirements. Each self-attention head is initialized randomly and acquires its functional role during training.

Generally, millions of books and billions of documents and web pages are used to train modern large language models. The total number of training tokens reflects the amount of text the model has been exposed to during training. Typically, increasing the volume and diversity of the training corpus improves a model's generalization performance, as it allows the system to capture a wider range of linguistic structures and real-world knowledge [22].

10. Structure of large language model

It is well known that a large language model generates output tokens one by one when a text query is provided as input. This approach is often referred to as GPT, which stands for Generative Pre-trained Transformer. The GPT algorithm (generative autoregressive model) can be considered the opposite of BERT. While BERT converts tokens into hypervector representations, GPT converts the hypervector representation of an input query into output tokens [22].

The information processing pipeline of a large language model is shown below in Figure 3. For simplicity, consider the input phrase “The cat sat on the” and output token ‘mat’. The first phase is tokenization, which converts input text into a sequence of tokens. Here, simple words like “cat” are represented by single token. At the same time, more complex words may be represented by several tokens to reflect the morphological structure of natural language. As an example, the word “cats” would be transformed into two tokens: ‘cat’ and ‘s’. Many language models also include special tokens to represent the start and end of the sequence, chat-format markers, padding tokens, and other structural elements.

The size of natural language vocabulary V is typically around 50'000. Mathematically, a token can be converted into a digital representation using one-hot encoding, which produces the vector of dimensionality V containing all zeros except for a single component set to 1. The concept of an embedding matrix E is used to obtain a token embedding. Basically, this matrix has dimensions $V \times d$ and stores embeddings in rows for the whole vocabulary. The token embedding vector can be calculated as $e = h^T E$, where h is one-hot vector. In real-world implementations, the algorithm typically performs a lookup of the embedding vector by token id [25].

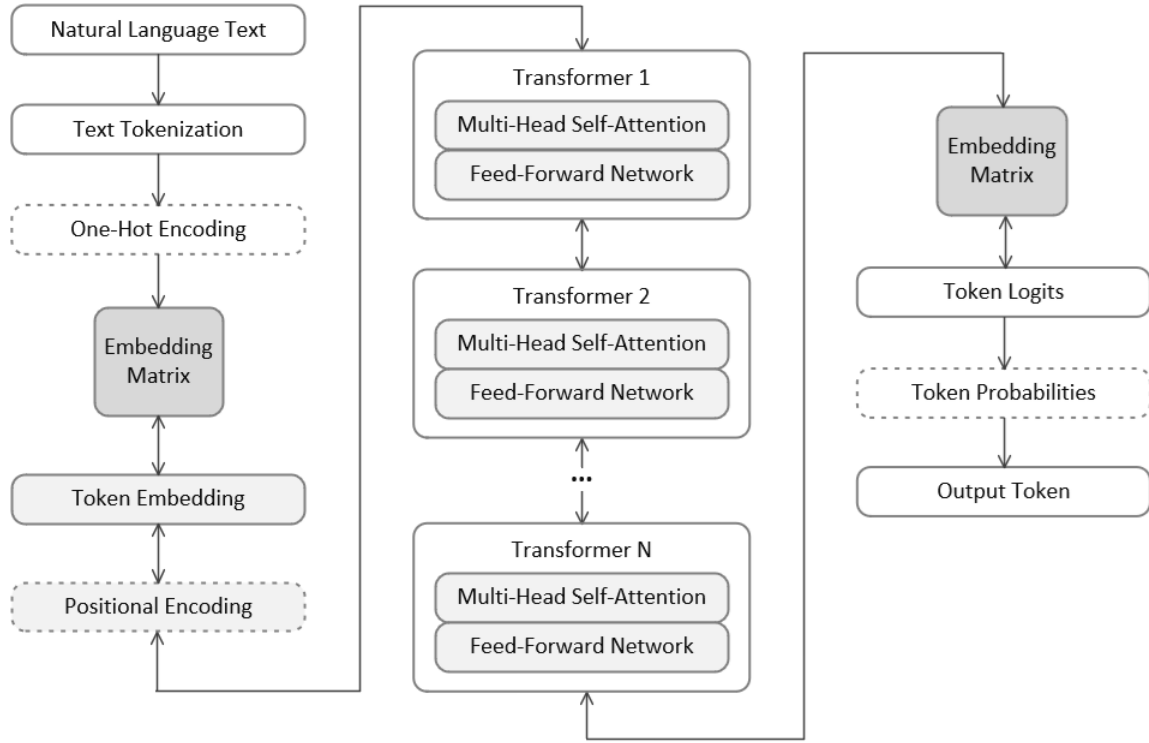


Figure 3: The conceptual scheme of modern large language model.

Once token embedding is available, the positional encoding is mixed in to specify the token’s location within the input phrase. The classic approach uses sine and cosine functions to encode such information. Also, modern LLMs often delegate positional encoding to the transformer. It is important to highlight at this stage that the sequence of tokens from the input phrase is converted into a sequence of vectors. The dependencies between tokens are analyzed later by transformers.

The core component of large language model architecture is the sequence of transformer layers. There is an analogy here with deep learning approach in neural networks: the more hidden layers a network has, the more complex functional dependencies it can learn. In the same way, the more transformer layers a language model has, the more complex semantic relationships it can capture. This concept happened to be crucial for transition from GPT-3 model to GPT-4. Eventually, this led to significant investments in datacenters and model training infrastructure that often outpaced the business requirements.

The internal structure of a transformer is quite complex, and this article does not aim to analyze it in detail. Essentially, it consists of two main components: a multi-head self-attention mechanism and a feed-forward network. Both components are followed by residual addition operations and by layer normalization steps. The self-attention mechanism is a neural network that uses four types of weight matrices for query, key, value, and output projection. The feed-forward network is typically a multi-layer perceptron with one hidden layer employing GELU activation function and a linear output layer [26].

The data processing pipeline, which starts with token embedding, includes positional encoding and all transformer layers, maintains uniform dimensionality d that matches the embedding size. This uniform structure provides greater flexibility in design and allows for scaling of transformer layers. Also, this dimensionality matches the embedding matrix used for token decoding.

Consider the example input phrase “The cat sat on the”. The corresponding token embeddings are generated and then processed sequentially by transformer layers. The embeddings produced before entering the transformers are largely independent of each other. However, after passing through (any) transformer layer, the embedding of the token becomes contextualized. For example, the contextualized embedding of last token ‘the’ now incorporates information about the preceding tokens and, to some extent, about the tokens that are likely to follow.

As shown above, the LLM architecture has a single shared embedding matrix that is used in two places: for token embedding and for token decoding. If the vector e_N represents the output of last transformer then the real-valued vector of dimensionality V representing the token logits can be computed as $z = Ee_N$, where E is the embedding matrix. After that, the logits are transformed into token probabilities using the softmax function.

$$p_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \quad (2)$$

Eventually, the output token is selected randomly according to the computed probabilities. This explains why LLMs can produce different outputs for the same input query. Once the output token is determined, it is appended to the input query for the next iteration of feed-forward LLM run. As a note, only the last token's hidden state is used to predict the next token. In other words, in order to predict the token 'mat' the LLM must process the entire preceding phrase: "The cat sat on the".

So far, we have been discussing the inference workflow of a large language model. The training process begins with assignment of small random values for all matrices involved. The entire LLM architecture is then trained using back-propagation algorithm, which adjusts all matrices at once. In addition, the embedding matrix is updated twice – at decoding phase and at token embedding phase – this concept is commonly referred to as weight tying. The scope of error propagation is illustrated in the diagram using the reverse arrows.

The prediction error is evaluated first at token logits layer after a forward pass. The calculated probabilities are compared with the one-hot vector representing the next token from the training text. This error is then backpropagated through the embedding matrix and all transformer layers. As a result, the embedding matrix acquires semantically meaningful representations of tokens.

11.Retrieval-augmented generation

As discussed, training a large language model is an extremely expensive process that only major enterprises can afford. Typically, it requires a datacenter with thousands of high-end GPUs. For example, the GPT-4 model, reportedly with about one trillion parameters, was trained on a cluster of 25'000 NVIDIA A100 GPUs over the course of 100 days.

On the other hand, many companies maintain large databases of confidential documents. Such documents cannot be shared with external LLM vendors, and renting an entire data center is not financially feasible for a mid-sized enterprise. One of the options is to apply additional training to an existing model, also known as fine-tuning. However, this still requires several high-end GPUs and deep understanding of the LLM training process.

While modifying a standard large language model is difficult, many of them provide the ability to supply additional context along with the user's original query. This enables a third approach for leveraging the power of language model with custom confidential documents: retrieval-augmented generation (RAG). It requires even fewer computing resources, as the basic setup typically involves one machine for the vector database and one GPU-powered machine for the language model [30].

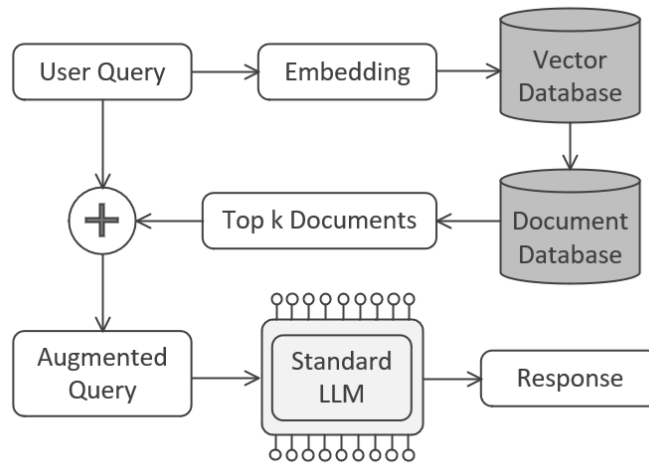


Figure 4: The architecture of retrieval-augmented generation pipeline.

The information processing workflow for this approach is shown in Figure 4. It assumes that embeddings for all corporate documents or chunks have already been generated and stored in the vector database. Once the user provides a query text, the first step is to generate the corresponding query embedding. The embedding model must match the one used to populate the vector database. Ideally, it should also match the model of a standard LLM used later in the pipeline.

The resulting embedding is used to perform an ANN search against the vector database and retrieve the top k most relevant documents. The number of documents to retrieve typically ranges between 5 and 10. Higher numbers can be justified by large embedding size and large LLM context. The textual representations of these k documents are then appended to the original user query in the context section. The augmented query is provided to a standard LLM, enabling it to generate a response that incorporates custom document content without requiring retraining.

The overall finding is that conversational agents (CAs), such as RAG-based systems, typically outperform traditional document search and wiki systems, as users locate the required information more quickly. The LLM responses include references to source documents, a feature commonly associated with explainable AI (XAI). Even better results can be achieved with retrieval-augmented fine-tuning (RAFT) systems, which combine both approaches: additional training of the LLM and the storage of corporate documents in a vector database.

While RAG systems may produce less refined output than traditional LLMs (especially those that use long context), they outperform them in terms of cost and efficiency. Hybrid systems are also possible, allowing for a trade-off between answer quality and the associated cost [31].

12.AI-related cloud services

Since cloud computing has become a modern trend for many enterprise companies, it is natural to integrate with cloud-based services that offer vector storage, LLM hosting, and RAG workflows. The most popular solutions are listed in Table 11 below. In addition to the main building blocks, cloud services often provide orchestration frameworks with tracing, debugging, evaluation, and monitoring capabilities, as observability is an important business consideration. It is helpful when the service has a standardized interface and is compatible with Model Context Protocol (MCP).

Table 11

Cloud services providing solutions for artificial intelligence tasks.

Service Name	Functionality	Service Description
AWS OpenSearch	Vector Database	Scalable search and vector database that supports efficient vector-driven and hybrid search on billions of embeddings

AWS Aurora and RDS pgvector	Vector Database	Managed Postgres database with native vector type support; extensive scalability and replication options for Aurora
AWS Bedrock Knowledge Bases	Vector Database	RAG pipeline service with automated ingestion and embedded vector storage engine; integrates with Aurora and Neptune
AWS Bedrock	LLMs	Fully managed AWS service that provides enterprise-grade access to leading foundation LLM models through a unified API for building and scaling generative AI applications
AWS SageMaker and JumpStart	Machine Learning LLMs	Cloud-native service that allows users to build, train, host and deploy machine-learning models at scale; integrated tools; JumpStart is built-in hub that provides foundation LLM models
AWS Neptune Analytics	Vector Database	Graph and vector analytics service optimized for fast similarity search, large-scale graph queries, and real-time analytics
AWS DynamoDB Vector Search	Vector Database	Integrated similarity search capability of DynamoDB, enables hybrid semantic–lexical retrieval and RAG workloads
Azure Cosmos DB Vector	Vector Database	Primarily NoSQL database with integrated vector database capabilities, supporting HNSW, IVF, DiskANN algorithms
Azure AI Search	Vector Database	Enterprise similarity search and retrieval service; unifies text, vector and multimodal search to agents and applications
Azure OpenAI	LLMs	Offers top-level LLM models, embeddings, and multimodal models; integrates tightly with vector search and RAG
Azure ML	Machine Learning	Enterprise-grade, scalable cloud service that accelerates and manages full machine learning lifecycle and operations
Azure Postgres with pgvector	Vector Database	Postgres database offering with pgvector extension for scalable vector similarity search and hybrid RAG workloads
GCP AlloyDB with pgvector	Vector Database	Postgres-compatible managed database that integrates Google- optimized pgvector and high-performance ScaNN algorithm
GCP Vertex AI LLMs and Vector Search	LLMs Machine Learning Vector Database	Unified platform supporting embedding models, content generative models, LLM fine-tuning; foundational regression and classification ML algorithms integrated with vector search; contains native vector database built on ScaNN, the latencies below 10 milliseconds; integrates with Vertex AI embeddings
GCP BigQuery Vector Search	Vector Database	Vector and similarity search capability of BigQuery; enables low-latency hybrid search and scalable RAG workloads
Milvus on Google Cloud	Vector Database	Milvus deployment available through Zilliz Cloud on GCP; high-performance vector storage and similarity search
MongoDB Atlas Vector Search	Vector Database	Document database with native vector search capabilities, available across cloud providers like AWS, Azure, and GCP
Fast.io Cloud	Vector Database LLMs	Alternative approach to vector databases, automatic document indexing for RAG without requiring manual embedding or vector database management, works with any LLM
Weaviate Cloud	Vector Database	Hybrid AI-native vector database, suitable for multimodal search and modular AI pipelines; integrated vectorizers

Qdrant Cloud	Vector Database	High-performance open-source vector database with managed cloud offering; strong filtering; latencies below 5 milliseconds
Zilliz Cloud	Vector Database	Fully managed, scalable, cost-effective, and high-performance vector database service built on open-source Milvus
Chroma Cloud	Vector Database	Serverless vector database service for storing embeddings and performing fast semantic and full-text search in AI applications
Hugging Face	LLMs	Popular platform offering managed LLM inference, typically used when companies need open-source LLMs in production
LangSmith Cloud	RAG Workflow	Cloud-based service that hosts all LangChain infrastructure with observability, evaluation, and deployment tooling

The runtime cost of required cloud services can be reduced by using a semantic cache [33]. It is a storage layer that retains query-result pairs based on their semantic meaning, not the exact text. It enables an application to avoid recomputing vector searches and LLM-generated answers when a new query is semantically similar to a previous one. A vector database is often used for semantic cache implementation, and it typically stores the following information: the user’s original query, the query embedding, the retrieved context vectors, and the final LLM answer. When a new query arrives, the system converts it to an embedding and compares it to cached query embeddings using cosine similarity. If the similarity exceeds a set threshold, the algorithm immediately returns the cached result. Otherwise, it performs a fresh vector search and uses LLM to produce new output.

13.Conclusion

The modern landscape of intelligent data processing is rapidly evolving, driven by advances in vector-based representations and the increasing availability of high-performance hardware and cloud-native services [34]. Hypervector architectures demonstrate how high-dimensional algebra can encode complex symbolic structures and how conceptually rich representations of real-world knowledge can be obtained from encoded collections of semantic triples [35]. Distributed vector databases and scalable multimodal embedding pipelines have made these vector representations operationally effective at industrial scale. In addition, the capabilities of modern large language models continue to expand, enabling deeper reasoning, multimodal understanding, and targeted retrieval-augmented generation workflows.

This work illustrates how new technologies enable organizations to integrate their proprietary datasets with the generalization strengths of frontier language models while maintaining privacy and cost efficiency. It offers a comprehensive and up-to-date selection of AI hardware accelerators, embedding libraries, vector databases, LLM services, and other components used in RAG pipelines. All computational infrastructure can also be deployed in a cloud environment, which now provides integrated AI services that simplify model orchestration, enhance security, and enable streamlined integration with enterprise architecture. Taken together, these components establish a data-centric, user-oriented approach in which private knowledge becomes a first-class citizen.

Declaration on Generative AI

During the preparation of this work, the authors used LLM-based AI assistants for grammar and spelling checks. After using this service, the authors reviewed and edited the content as needed and take full responsibility for the publication’s final content.

References

- [1] S. Haykin, Neural networks: a comprehensive foundation, Prentice Hall, 1998. doi: 10.5555/541500.
- [2] C. M. Bishop, Pattern recognition and machine learning (Information Science and Statistics), Springer, 2006. URL: <https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf>
- [3] R. O. Duda, P. E. Hart, and D. G. Stork, Pattern Classification, 2nd ed. New York: Wiley-Interscience, 2001.
- [4] L. Breiman. (2001). Random Forests. Machine Learning, 45(1), 5–32. doi: 10.1023/A:1010933404324
- [5] C. Shah, A Hands-On Introduction to Machine Learning, 1st ed., Cambridge University Press, Cambridge, 2023.
- [6] Y. Koren, Working with MediaWiki. CreateSpace Independent Publishing Platform, 2014. doi: 10.5555/3153523.
- [7] HD Computing, High-Performance Computing Solutions, 2025. URL: <https://www.hd-computing.com/>.
- [8] P. Kanerva, Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. Cognitive Computation, 2009, 1(2), 139–159. doi: 10.1007/s12559-009-9009-8.
- [9] K. Schlegel, P. Neubert, P. Protzel. A Comparison of Vector Symbolic Architectures. arXiv preprint arXiv:2001.11797v3, 2020. URL: <https://arxiv.org/abs/2001.11797v3>.
- [10] D. Kleyko, M. Davies, E. P. Frady, P. Kanerva, S. J. Kent, B. A. Olshausen, E. Osipov, J. M. Rabaey, D. A. Rachkovskij, A. Rahimi, F. T. Sommer. Vector Symbolic Architectures as a Computing Framework for Emerging Hardware. arXiv preprint arXiv:2106.05268, 2021. URL: <https://arxiv.org/pdf/2106.05268>.
- [11] D. Kleyko, D. A. Rachkovskij, E. Osipov, A. Rahimi (2023). A Survey on Hyperdimensional Computing aka Vector Symbolic Architectures, Part I: Models and Data Transformations. ACM Computing Surveys, 55(9), Article 184. doi: 10.1145/3538531.
- [12] D. Kleyko, D. A. Rachkovskij, E. Osipov, A. Rahimi (2023). A Survey on Hyperdimensional Computing aka Vector Symbolic Architectures, Part II: Applications, Cognitive Models, and Challenges. ACM Computing Surveys, 55(9), Article 175. doi: 10.1145/3558000
- [13] M. Heddes, I. Nunes, T. Givargis, A. Nicolau, A. Veidenbaum. Hyperdimensional computing: a framework for stochastic computation and symbolic AI, Journal of Big Data 11, 2024, 145. doi: 10.1186/s40537-024-01010-8.
- [14] Y. Sun, R. V. Hu, Y. Zhang, J. Tang, and W. Wang. JOIE: Jointly Learning Ontological and Entity Embeddings for Knowledge Graph Alignment. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD 2019), pages 1996–2006. ACM, 2019. URL: https://web.cs.ucla.edu/~yzsun/papers/2019_KDD_JOIE.pdf.
- [15] J. Chen, O. Mashkova, F. Zhapa-Camacho, R. Hoehndorf, Y. He, I. Horrocks. Ontology Embedding: A Survey of Methods, Applications and Resources. arXiv preprint 2024, URL: <https://arxiv.org/html/2406.10964v1>.
- [16] R. Schwaber-Cohen. What is a Vector Database & How Does it Work? Use Cases + Examples. Pinecone, May 3, 2023. URL: <https://www.pinecone.io/learn/vector-database>.
- [17] V. Slocum. Vector Search Explained. Weaviate Blog, November 21, 2024. URL: <https://weaviate.io/blog/vector-search-explained>.
- [18] Surya. How Vector Databases Search: A Practical Guide to IVF, HNSW, PQ & ScaNN. In: Hoomanely Tech Blog, 2025. URL: <https://tech.hoomanely.com/how-vector-databases-search-a-practical-guide-to-ivf-hnsw-pq-scann>.
- [19] H. Cao, Recent advances in universal text embeddings: A Comprehensive Review of Top-Performing Methods on the MTEB Benchmark, 2024, arXiv:2406.01607v2. URL: <https://arxiv.org/abs/2406.01607v2>.

- [20] C. Li, Z. Gan, Z. Yang, J. Yang, L. Li, J. Wang, J. Gao (2023). Multimodal Foundation Models: From Specialists to General-Purpose Assistants. arXiv:2309.10020 [cs.CV]. URL: <https://arxiv.org/abs/2309.10020>.
- [21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention Is All You Need, in: Proceedings of the 31st Conference on Neural Information Processing Systems (NeurIPS 2017), 2017, pp. 6000–6010. URL: <https://papers.nips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>
- [22] S. Minaee, T. Mikolov, N. Nikzad, M. Chenaghlu, R. Socher, X. Amatriain, J. Gao. Large Language Models: A Survey. arXiv preprint arXiv:2402.06196 (2025). URL: <https://arxiv.org/abs/2402.06196>.
- [23] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever. Improving Language Understanding by Generative Pre-Training. OpenAI Technical Report (2018). URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.
- [24] I.D. Mienye, N. Jere, G. Obaido, O.O. Ogunraku, E. Esenogho, C. Modisane. Large language models: an overview of foundational architectures, recent trends, and a new taxonomy. Discover Applied Sciences 7, 1027 (2025). doi: 10.1007/s42452-025-07668-w.
- [25] S. Wolfram, What is ChatGPT doing and why does it work, 2023. URL: <https://writings.stephenwolfram.com/2023/02/what-is-chatgpt-doing-and-why-does-it-work/>.
- [26] M. Shao, A. Basit, R. Karri, and M. Shafique (2024). Survey of Different Large Language Model Architectures: Trends, Benchmarks, and Challenges. arXiv preprint arXiv:2412.03220. URL: <https://arxiv.org/abs/2412.03220>.
- [27] S. Yin, C. Fu, S. Zhao, K. Li, X. Sun, T. Xu, E. Chen, A Survey on Multimodal Large Language Models. 2024, arXiv preprint arXiv:2306.13549. URL: <https://arxiv.org/abs/2306.13549>.
- [28] S. Raschka. The Big LLM Architecture Comparison: From DeepSeek-V3 to Kimi K2 – A Look At Modern LLM Architecture Design. Ahead of AI, Substack, July 19, 2025. URL: <https://magazine.sebastianraschka.com/p/the-big-llm-architecture-comparison>.
- [29] S. Raschka. Understanding Multimodal LLMs: An Introduction to the Main Techniques and Latest Models. Ahead of AI, Substack, November 3, 2024. URL: <https://magazine.sebastianraschka.com/p/understanding-multimodal-llms>.
- [30] M. Klesel, H. F. Wittmann. Retrieval-Augmented Generation (RAG). Business & Information Systems Engineering, vol. 67, pp. 551–561, 2025. URL: <https://link.springer.com/article/10.1007/s12599-025-00945-3>.
- [31] S. Gupta, R. Ranjan, S. N. Singh. A Comprehensive Survey of Retrieval-Augmented Generation (RAG): Evolution, Current Landscape and Future Directions. arXiv:2410.12837, 2024. URL: <https://arxiv.org/abs/2410.12837>.
- [32] S. Ganatra, P. Bhattacharyya. Retrieval Augmented Generation: A Comprehensive Survey on Bridging Language Models and External Knowledge. IIT Bombay, 2025. URL: https://www.cfilt.iitb.ac.in/resources/surveys/2025/Shrey_Survey%20%281%29.pdf.
- [33] Z. Jing, Y. Su, Y. Han, B. Yuan, H. Xu, C. Liu, K. Chen, and M. Zhang. When Large Language Models Meet Vector Databases: A Survey. arXiv:2402.01763 [cs.DB], 2024. URL: <https://arxiv.org/abs/2402.01763>.
- [34] A. Yu. Doroshenko, Formalni ta intelektualni metody avtomatyzatsii proiektuvannia prohramnykh system (stenohrama dopovidi na zasidanni Prezidii NAN Ukrainy 19 liutoho 2025 r.) [Formal and intellectual methods of software systems design automation (transcript of the report at the meeting of the Presidium of the NAS of Ukraine, February 19, 2025)]. Visnyk NAN Ukrainy, (4), 2025, 62–68. doi: 10.15407/visn2025.04.062 (in Ukrainian).
- [35] I. Sinityn, A. Doroshenko, J. Rogushyna, K. Yurchenko, and O. Yatsenko. Ecosystem Modelling for Studying and Implementing Experience: Integration of Ontological Analysis, LLMs and Chatbots. In Proceedings of the International Conference on Modern Ontology-Driven Systems (MODS 2025), Chernihiv, Ukraine, 2025. CEUR Workshop Proceedings, to appear.