ChatGPT

Weaviate Handbook

Executive Summary

Weaviate is an **open-source vector database** that stores both objects (JSON documents) and their vector embeddings [1]. It enables advanced **semantic search** (search by meaning) in large datasets, as well as **hybrid search** (combining vector similarity with keyword filters) [2]. Weaviate also supports **Retrieval-Augmented Generation (RAG)**: it can retrieve relevant data and feed it into large language models to answer questions or generate content [3]. Additionally, Weaviate features **agentic AI workflows** (Weaviate Agents) to simplify complex queries and data transformations.

As of early 2026, the latest stable release of Weaviate is **version 1.35.x** (released December 2025) [4]. Key capabilities include built-in support for various embedding models (openAI, Cohere, Hugging Face, etc.), multi-modal data (text, images, audio), automatic indexing with HNSW, and modules for tasks like question-answering, summarization, and more. Weaviate can run on-premise or in the cloud (WCS) and offers high availability via sharding and replication. Its combination of vector search, rich query language, and modular architecture makes it ideal for modern AI applications like semantic Q&A, product search, chatbots, and knowledge bases.

Core Concepts and Terminology

- **Objects and Classes (Collections):** Weaviate stores data objects (records) as JSON documents. Related objects are grouped into **classes** (also called collections) that share a common schema [5]. For example, a `Product` class might have properties like `name`, `description`, and `price`. Each object has one or more **properties** (fields) and typically includes a **vector**

embedding (a numeric array) that encodes its semantic content [5]. In short, a class = schema definition (like a table schema), and an object = one row with values and a vector.

- **Vector Embeddings:** A vector is a list of numbers (e.g. length 1536) representing the meaning of text, image, or other data. Weaviate supports *named vectors* (multiple embedding per object) and vector fields. The vector is used for similarity search. By default, Weaviate uses **cosine distance** to compare vectors [6], but it also supports dot-product or Euclidean metrics if configured.

- **Schema:** Defines classes, their properties, and modules used. Weaviate's schema is explicit (JSON or via client APIs), though it can auto-generate on ingestion (not recommended for production) [7]. A schema class lists each property name, data type (string, number, etc.), and which module/vectorizer to use if any.

- **Vectorizer Modules:** Modules that convert raw data into vectors. Common vectorizers include `text2vec-openai`, `text2vec-transformers`, `text2vec-cohere` for text, `img2vec-neural` or `multi2vec-clip` for images, and `multi2vec-bind` for any-media embeddings [8]. Without a vectorizer, you can also supply your own embeddings manually.

- **Modules:** Weaviate's functionality is extended by *modules*. These include vectorizer modules (described above), as well as **reader/generator modules** (e.g. `qna-transformers`, `generative-openai`) that process retrieved data with LLMs, and **rerank modules** that refine search results [9]. Modules must be enabled explicitly in the configuration. When no modules are attached, Weaviate becomes a "pure" vector store that still requires you to provide vectors for each object [10].

- **Distance Metrics:** The *distance metric* defines how similarity between vectors is measured. Weaviate's default is cosine distance, which works well for most embedding models [6]. You can change it per class (e.g. to dot-product or Euclidean). Note that changing the metric requires re-indexing all data.

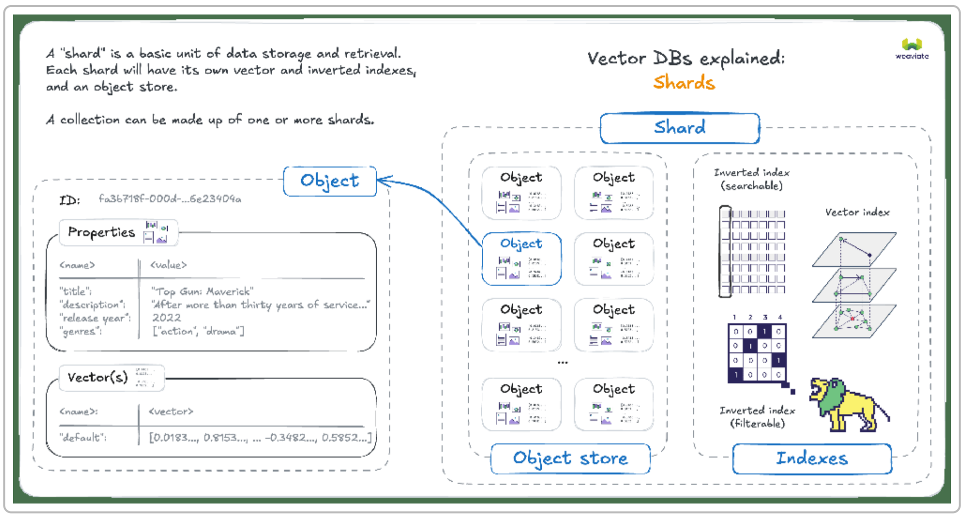
- **Hybrid Search:** Weaviate supports **hybrid search**, which combines vector search with keyword (BM25) search [11]. In a hybrid query, you can specify both a vector (or text for embedding) and exact search terms, balancing semantic matching with lexical matching. This allows for precise filtering (e.g. category tags) alongside "fuzzy" semantic relevance [11].

- **Retrieval-Augmented Generation (RAG):** RAG is a method where a query is first answered by retrieving relevant data, and then a generative model (LLM) produces the final answer. In Weaviate, a RAG query involves a search query and a generative prompt. Weaviate retrieves relevant objects via vector search, then passes their content (and the prompt) to a generative module, returning the model's output [12]. This enables intelligent Q&A, summarization, and other tasks that require both knowledge retrieval and language generation.

Architecture and Components

Weaviate is built as a distributed vector database with a modular architecture. Each **Weaviate node** runs the core engine plus any enabled modules (vectorizers, read/generate, etc.). Data is distributed across nodes using **shards** and **replicas** for scalability and reliability [13] [14]:

- **Shards:** A collection's data is split into shards, which are the basic unit of storage and retrieval [15]. Each shard contains its own object store, **vector index** (HNSW by default), and an inverted index for filters. The diagram below illustrates a shard's contents:



A "shard" is a basic unit of data storage and retrieval.
Each shard will have its own vector and inverted indexes,
and an object store.
A collection can be made up of one or more shards.
Vector DBs explained:
Shards
weaviate
Shard
Object
ID: fa3b718f-000d-...6e23404a
Properties
<name>
<value>
"title": "Top Gun: Maverick"
"description": "After more than thirty years of service..."
"release year": 2022
"genres": ["action", "drama"]
Vector(s)
<name>:
<vector>
"default": [0.0183..., 0.8153..., ... -0.3482..., 0.5852...]
Object
Object store
Inverted index (searchable)
Vector index
Inverted index (filterable)
Indexes

Figure: Illustration of a Weaviate shard (basic unit of storage). Each shard holds an object store (JSON data), an HNSW vector index, and an inverted index for property filtering [15].

- **Replication:** For high availability, shards can have multiple replicas on different nodes. Each replica holds the same data; if one node fails, another can take over. Replication improves fault tolerance and read throughput [14].

- **Vector Index (HNSW):** Weaviate uses **HNSW (Hierarchical Navigable Small World)** graph indexes for fast nearest-neighbor search. Key parameters are:

- `efConstruction`: controls index build time vs. accuracy. Higher values build a more precise index but take longer [16].
- `maxConnections` (M): graph connectivity (higher M = more links, higher accuracy, more memory) [17].

- `ef`: search-time parameter (bigger `ef` yields higher recall at cost of slower search) [16]. The default `ef` is often 64–128; values >512 have diminishing returns [16].

- **Modules:** Weaviate's modular design means additional services run alongside the core:

- *Vectorizer modules* (e.g. `text2vec-transformers`, `text2vec-openai`) either run as separate containers or call external APIs to generate embeddings.
- *Reader/Generator modules* (e.g. `qna-transformers`, `generative-openai`) take retrieved objects and perform LLM tasks.
- *Rerank modules* re-order results using advanced models.

- *Backup modules* handle snapshots to cloud storage. Modules communicate with the core via REST/gRPC.

- **Deployment Models:** Weaviate can be deployed **locally** (single-node Docker/Compose), in a **cluster** (multiple nodes, e.g. via Kubernetes or manual setup), or in the **cloud** (Weaviate Cloud Service).

- *Local:* run a single Weaviate instance (e.g. via Docker Compose) for development or small workloads.
- *Cluster:* multiple Weaviate nodes forming a cluster (sharding/replication). Each node must be configured with a unique `CLUSTER_HOSTNAME` and shared network access. Persistence is achieved via external volumes.
- *Weaviate Cloud Service (WCS):* fully managed clusters provided by SeMI Technologies (see Weaviate Cloud). WCS handles the infrastructure, scaling, and maintenance. It is available in multi-tenant ("Shared Cloud") or single-tenant ("Dedicated Cloud") modes [18] [19].

In summary, Weaviate's architecture separates **storage and search** (vector indexes on shards) from **vectorization and language tasks** (modules), allowing it to scale horizontally. An external vector index (HNSW) ensures fast similarity search, while modules plug in specialized ML capabilities.

Quickstart Guides

Docker Compose Quickstart

For local experimentation, Weaviate provides a Docker Compose setup. Below is an example `docker-compose.yml` that launches Weaviate (v1.35.3) with persistence and no modules enabled:

```
# docker-compose.yml (example)
version: '3.4'
services:
  weaviate:
    image: semitechnologies/weaviate:1.35.3
    ports:
      - "8080:8080"   # GraphQL/REST API
      - "50051:50051" # gRPC API (for vector modules)
    volumes:
      - weaviate_data:/var/lib/weaviate  # persistent storage
    environment:
      CLUSTER_HOSTNAME: "node1"
      PERSISTENCE_DATA_PATH: "/var/lib/weaviate"
      AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED: "true"
      # (Add ENABLE_MODULES=... to enable vectorizer or generative modules)
volumes:
  weaviate_data:
```

This configuration mounts a Docker volume at `/var/lib/weaviate` so data persists across restarts [20]. Replace `ENABLE_MODULES` in the environment to turn on modules (e.g. `text2vec-transformers`, `generative-openai`, etc.). Note: if multiple Weaviate instances (nodes) run, each must have a unique `CLUSTER_HOSTNAME` [20].

Once running (`docker-compose up -d`), Weaviate is accessible at `http://localhost:8080` for REST/GraphQL and port 50051 for gRPC. Use the Weaviate clients (Python, JavaScript, etc.) or HTTP calls to create classes, add objects, and run queries. For example, connect in Python:

```
import weaviate
client = weaviate.Client("http://localhost:8080")
```

Kubernetes Deployment

For production or large-scale setups, Weaviate provides an official Helm chart. First add the repo and install:

```
helm repo add weaviate https://weaviate.github.io/weaviate-helm
helm repo update
helm install my-weaviate weaviate/weaviate --namespace weaviate --create-
namespace
```

This deploys Weaviate with default settings. You should supply a `values.yaml` to customize the deployment. For example, to use version 1.35.3, enable 3 replicas, and turn on API-key auth:

```
weaviate:
  image:
    tag: 1.35.3
replicaCount: 3

authentication:
  apikey:
    enabled: true
    allowedKeys:
      - your-admin-api-key

authorization:
  adminList:
    enabled: true
    users: ["admin-user"]
```

Save this as `weaviate-values.yaml` and run:

```
helm install my-weaviate weaviate/weaviate --namespace weaviate -f weaviate-
values.yaml
```

This sets the image version and enables API-key authentication (by default Helm chart uses random keys). You can also set resource requests/limits and enable modules in `weaviate.modules` blocks. For details, see the Weaviate Helm chart and documentation [21]. The Helm chart will create Kubernetes Services, Deployments, and PersistentVolumeClaims for storage.

Weaviate Cloud Service (WCS)

Weaviate Cloud Service (WCS) is a managed Weaviate by SeMI. It lets you spin up production-ready clusters without manual ops. To use WCS: 1. Sign up at weaviate.io/cloud and log in. 2. Create a new instance (choose region and plan: Shared or Dedicated). 3. For Shared Cloud, you can start a free sandbox instance for testing. Dedicated Cloud offers dedicated resources with compliance (SOC2, HIPAA) [19] . 4. After provisioning, WCS provides an endpoint URL and an API key. Use those to connect:

```
client = weaviate.Client(
    url="https://INSTANCE.wcs.weaviate.cloud",
    api_key="YOUR_WCS_API_KEY"
)
```

5. WCS handles clustering, storage, SSL/TLS, and monitoring automatically.

WCS gives you the same Weaviate API and features, without managing infrastructure [18] . It's ideal for teams who want vector search with minimal setup. For example, the local Quickstart mentions a WCS sandbox with admin key and endpoint for testing [22] .

Schema Design Patterns

Choosing the right schema is crucial for effective search. Here are patterns for common use cases:

- **Document Q&A / Knowledge Base:** Break large documents into smaller objects (e.g. paragraphs) for finer-grained search [7] . A class like `Article` could have properties `title` (Text), `paragraph` (Text), and optional metadata (date, author). Enable a text vectorizer module. Weaviate recommends "paragraph-level" chunks for best precision [7] . Optionally, use cross-reference properties (`Ref` type) to link related paragraphs or articles (at cost of additional storage).

- **Product Search:** Define a `Product` class with textual fields (`name`, `description`), categorical fields (`category`, `brand`), and numeric fields (`price`, `stock`). Use a text vectorizer for the description (and/or name). This lets semantic search for similar products, while exact matches (e.g. filtering by category or price range) are done with GraphQL filters. For example, you could query: nearest neighbors to "wireless headphones" *and* `where: { path: ["category"], operator: Equal, valueText: "Electronics" }`. Hybrid search can blend a keyword search on `name` with vector search on `description`. Embedding product descriptions yields more recall (e.g. catching synonyms) [23] .

- **Knowledge Graph / Ontology:** Weaviate's schema supports references between classes, enabling knowledge graphs. For example, classes `Author` and `Book` where `Book` has a `Ref` property to `Author`. Objects of each class have text fields (e.g. author biography, book synopsis) vectorized for search. This design can support semantic queries like "Find books about quantum physics authored by professors at MIT."

- **Multimodal Retrieval:** For data with images and text (e.g. products with photos, or articles with images), create a class with both image and text properties. Enable both `text2vec-transformers` (or similar) and an image vectorizer like `img2vec-neural` or `multi2vec-clip`. Weaviate can then index images and text. For example, a `Photo` class might have `caption` (Text) and `image` (Blob). With the `multi2vec-clip` module, both fields are embedded into a shared space so you can query by text and match images. More advanced: use the `multi2vec-bind` module (ImageBind) to index images, audio, and video together [8]. This enables “any-to-any” searches (e.g. find images by audio or vice versa).

In all cases, **define schema explicitly** rather than relying on auto-schema. Disable autoschema in production and carefully plan your classes. Use consistent naming and types. Keep schemas small and focused: if a class is too broad, split it. A good rule of thumb: smaller, more specific classes and objects generally yield more precise search [7].

Ingestion Strategies

Efficiently loading data into Weaviate involves batching, chunking, and vectorizing:

- **Chunking:** As noted, large texts should be split into chunks (paragraphs, sections, or fixed-size tokens) before ingesting. Chunking ensures each object fits within an LLM’s context window and improves retrieval relevance [24] [7]. There are strategies like fixed-length chunks, sentence/paragraph splits, or semantic chunking (overlapping windows). The [Weaviate chunking blog] outlines trade-offs [24]: e.g. fixed-length chunks are simple but may split sentences, whereas semantic chunking (using embeddings to group text) is more complex.

- **Vector Generation:** Weaviate can generate vectors in two ways:

- *On ingest (server-side)*: If a vectorizer module is enabled, Weaviate will call it for each object and store the resulting vector automatically [25]. For example, with `text2vec-openai` enabled and an OpenAI key provided, Weaviate will send the text to OpenAI to get embeddings upon each object creation [25].
- *Pre-computed (client-side)*: You can compute embeddings outside (e.g. using Hugging Face or OpenAI clients) and then provide them in the object JSON under a `vector` field. This bypasses vectorizer modules.

Choose server-side generation for simplicity, but note external APIs can incur latency/cost. Client-side gives you control (especially offline).

- **Batch Upload:** Always use batch import rather than one-object inserts. Weaviate supports both client-side and server-side batching [26]. Batch mode (default in recent versions) automatically optimizes throughput by parallelizing requests. The documentation notes batch imports can improve speed by 100× by reducing HTTP overhead [26]. Example using Python client:

```
# Create collection 'Article' with text2vec-openai
article_schema = {
    "class": "Article",
    "vectorizer": "text2vec-openai",
    "properties": [{"name": "content", "dataType": ["text"]}]
}
client.schema.create_class(article_schema)

# Batch import a list of article objects
with client.batch(batch_size=50, dynamic=True) as batcher:
    for obj in article_data:
        batcher.add_data_object({"content": obj["text"]}, "Article")
```

This will automatically send objects in batches of 50 with server-side optimization.

- **Metadata and Filtering:** Include metadata fields (strings, numbers, booleans) in your schema for filtering. For example, tags, categories, dates, or any structured data should be stored in properties. Then you can apply GraphQL filters (`where` clauses) to include/exclude objects. For example, `where: {path:["author"], operator:Equal, valueText:"Alice"}` filters by author name. Weaviate supports many filter operators on numeric, text, and geo data. Combining vector search with `where` filters yields powerful hybrid queries.

- **Reference (Relation) Ingestion:** If using cross-references between objects, you must create both objects and then add a reference property linking them. For example:

  ```
  author = client.collections.get("Author")
  author.add_object({"name": "John Doe"})
  author_id = result["id"]

  book = client.collections.get("Book")
  book.add_object({"title": "Deep Learning 101"})
  book_id = result["id"]

  book.add_reference(book_id, "writtenBy", "Author", author_id)
  ```

  Careful: adding references may require consistency (the reference class exists).

Weaviate Modules

Weaviate supports many optional modules. Each serves a different purpose:

- **Text Vectorizers (`text2vec-*`):**
- `text2vec-transformers`: Runs a locally hosted transformer (BERT/DistilBERT) container to embed text. Good for on-premise privacy. Provides high-quality embeddings but needs significant CPU/GPU memory.
- `text2vec-openai`: Calls OpenAI's embedding API at ingest time [25]. Very powerful embeddings (e.g. OpenAI's Ada), easy to use if you have an API key, but involves network calls and cost.
- `text2vec-cohere`: Similar to OpenAI but uses Cohere's API [23]. Offers high-quality domain-tuned models and is cost-effective in some cases.
- `text2vec-contextionary`: (older, context-specific lexicon model) – used to be default, but mostly superseded by transformer modules. Limited to language used to train.

Pros: Transformer modules support many languages, produce high accuracy. `openai`/`cohere` yield state-of-art embeddings.
Cons: Hosted modules (`openai`, `cohere`) send data off-site, may be costly. Local `transformers` require heavy resources.

- **Image/Multimodal Vectorizers:**

- `img2vec-neural`: Uses Neural networks (often ResNet or VGG-based) to embed images. Enables image similarity search.
- `multi2vec-clip`: Uses OpenAI’s CLIP model to embed text and images into a shared space. Great for cross-modal search (image or text queries).
- `multi2vec-bind` (ImageBind): Embeds images, audio, video into one space [8], enabling “any-to-any” multimodal search.
  Use Cases: Product catalogs with images, digital asset libraries, art collections.

- **Reader/Generator Modules:**

- `qna-transformers`: Extractive Q&A. Given a question and a context (retrieved object), it returns an answer span from the context. Useful for factual Q&A.
- `generative-openai`, `generative-huggingface`: Use LLMs (OpenAI GPT models or Hugging Face models) to generate answers or summaries. For example, RAG with `singlePrompt` or `groupedTask`.
  Pros: Automate Q&A, summarization, translation, etc., directly in Weaviate queries (see RAG section).
  Cons: Requires models (external API or heavy local models), potential latency.

- **Reranker Modules:**

- `rerank-cohere`, `rerank-mistral`, etc.: Re-order top search results for higher relevance using a secondary model [27]. Ideal when precision is critical (e.g. re-ranking the top 50 results from vector search).

- **Other Modules:**

- `text-spellcheck`: Corrects spelling in search queries.
- `gcs-backup`, `s3-backup`, `filesystem-backup`: For backup/restore functionality to Google Cloud Storage, AWS S3, or local FS.
- **Custom Modules:** You can build your own (see Weaviate’s [custom modules documentation]).

Each module must be enabled in config (e.g. `ENABLE_MODULES=text2vec-openai,img2vec-neural`) and, where needed, configured with API keys (e.g. `OPENAI_APIKEY`). Consult the [Modules documentation] for specifics. In practice, common combos are: - Text: `text2vec-openai` + optional `qna-transformers` for Q&A. - Images: `multi2vec-clip` (text+image). - Mixed: `text2vec-transformers` + `img2vec-neural` for separate text/image fields. - RAG: include a generative module (e.g. `generative-openai`) and set it as the generative provider.

When choosing, consider: - **Quality vs. Control:** External APIs (OpenAI, Cohere) often give best quality with minimal ops, but you lose data control and incur cost. In-house models (`transformers`, CLIP) are fully under your control but require more resources. - **Use Case Fit:** If doing lots of question answering, the `qna-transformers` module is optimized for that (extractive answers). If doing translations or free-form answers, use a generative module. - **Deployment:** Hosted modules (e.g. OpenAI, Cohere) are allowed on Self-hosted Weaviate but imply sending data out. On WCS, these are managed for you. Dedicated Cloud can be configured to comply with enterprise data governance (SOC2, HIPAA) [19].

Retrieval-Augmented Generation (RAG) Pipelines

A typical RAG pipeline with Weaviate has these steps:

1. **Chunking Source Data:** Prepare your knowledge base by splitting documents into smaller chunks (as in Schema Design). Each chunk is an object.
2. **Vector Indexing:** Ingest chunks into Weaviate so that each has a semantic vector (via a text module). Ensure good coverage so queries find relevant chunks.
3. **Query and Retrieval:** At query time, take the user's question and embed it (using the same vectorizer). Perform a vector search (`nearText`) to retrieve top-K relevant objects. Optionally apply filters (e.g. document type or date).
4. **Reranking (Optional):** If desired, apply a rerank module on the retrieved objects to refine their order. Alternatively, use GraphQL sort on properties (e.g. by timestamp or custom score).
5. **Generative Prompting:** With the top results in hand, formulate prompts for an LLM. Weaviate supports two modes in the `generate.nearText` (or `generate.nearVector`) query:
6. **Single Prompt:** Apply a prompt to each object separately. E.g. `"Answer the question '{question}' given the context {content}"`. Weaviate will return an LLM response for each retrieved object.
7. **Grouped Task:** Aggregate contexts into one prompt (or prompt the model to summarize all at once). E.g. `"Summarize these documents: {doc1} {doc2} ..."`.
8. **Return Answer:** Weaviate returns the generated answers (and/or the original objects). For example, using the Python client:

```
response = client.collections.use("Article").generate.near_text(
    query="What is quantum entanglement?",
    single_prompt="Answer in simple terms: {content}",
    grouped_task="Summarize all contexts",
    limit=3
)
for obj in response.objects:
    print(obj.generative.text)  # The answer generated from that context
print("Combined summary:", response.generative.text)  # from groupedTask
```

In JavaScript/TypeScript:

```
const articles = client.collections.use("Article");
const result = await articles.generate.nearText("Define quantum
entanglement", {
  singlePrompt: "Explain: {content}",
  groupedTask: "Overview of concepts",
}, {
  limit: 3
});
console.log(result.generative?.text);   // groupedTask answer
for (const obj of result.objects) {
  console.log(obj.generative?.text);    // singlePrompt answers
}
```

These examples are adapted from Weaviate docs [28] [29].

1. **Evaluation:** Evaluate the output for correctness. Metrics may include BLEU/ROUGE against ground truth, or manual QA evaluation. Improve by adjusting chunking strategy [24], changing number of retrieved docs, or refining prompts.

In code, Weaviate's client libraries make this easy. For instance, the Python example [28] and JavaScript example [29] above show how to call `generate.nearText()` with prompts. The key is combining vector retrieval (`nearText` or `nearVector`) with the generative options. Behind the scenes, Weaviate handles feeding the retrieved text into the LLM (OpenAI/GPT, etc.) and returns the model's responses.

In summary, a RAG pipeline in Weaviate leverages its vector search for retrieval and a generative module for answer writing, as demonstrated in the code above [12] [28].

Production Deployment

In production, consider scalability, reliability, and observability:

- **Clustering:** Run multiple Weaviate nodes (e.g. via Kubernetes). Use shards and replicas to scale. If you have X TB of data, you might shard it across nodes; replication factor >1 for HA. Plan resource requests (CPU, RAM) based on index size. Use the Horizontal scaling guidelines in docs [13]. Ensure each pod/node has a distinct `CLUSTER_HOSTNAME`.

- **Index Tuning (HNSW):** Depending on your accuracy/latency needs, tune HNSW parameters [16]. For larger datasets, you might increase `efConstruction` for a more accurate index at build time. If queries are slow, you can adjust `ef` or increase `maxConnections (M)`. See Vector index configuration for details. Monitor memory usage, since HNSW can be memory-intensive.

- **Persistence:** Always mount persistent volumes for data. In Docker/Kubernetes, use a volume or PVC at `/var/lib/weaviate` and set `PERSISTENCE_DATA_PATH` accordingly [20]. This ensures data survives restarts. (Weaviate also supports an LSM tree, which writes segments to disk.) Configure disk space and I/O carefully: Weaviate tracks disk use and can auto-mark a shard read-only if disks fill [30].

- **Backup and Restore:** Enable Weaviate's backup module for regular snapshots. Set environment variables like `ENABLE_MODULES=backup-filesystem` or `backup-s3` and configure storage paths or S3 buckets. Then use the client API to `backup.create()` and `restore()` [31] [32]. Backups integrate with S3/GCS/Azure for portability [31]. Regular backups (e.g. nightly) are essential, even in a cluster.

- **Monitoring:** Turn on Prometheus metrics by setting `PROMETHEUS_MONITORING_ENABLED=true` in the environment [33]. Weaviate will expose `/

`metrics` (default port 2112) with many metrics (query latencies, cache hits, memory, etc.) [34]. You can use Grafana dashboards (Weaviate provides example dashboards via a GitHub repo [35]) to visualize performance. Important metrics include query timings (`weaviate_query_durations_ms`), batch import times (`batch_durations_ms`), and memory/cpu usage.

- **High Availability:** In Kubernetes, ensure multiple replicas and pod anti-affinity so nodes land on different machines. Use readiness probes (Weaviate has a health endpoint). The Helm chart enables PVCs for data, which helps reschedule pods. For zero-downtime schema changes, use aliases (not covered here, but available).

In short, production deployment requires clustering (shards/replicas) [13], persistent volumes [20], tuned HNSW parameters [16], regular backups [31], and monitoring (Prometheus/Grafana) [34] [33].

Security

Weaviate provides robust access controls and data protection features:

- **Authentication (AuthN):** Weaviate supports API Key authentication, OpenID Connect (OIDC), or anonymous access (discouraged in prod) [36]. For API keys, you can list valid keys via environment variables (`AUTHENTICATION_APIKEY_ALLOWED_KEYS`) or manage users with the built-in user API (v1.30+) [21]. In Kubernetes/Helm, configure under the `authentication` section of `values.yaml`. For OIDC, configure Weaviate to trust an identity provider (e.g. Keycloak or WCD's own OIDC) [37].

- **Authorization (AuthZ):** Beyond verifying identity, Weaviate enforces permissions. You can use the **Admin list** scheme (grant some users admin vs. read-only access) or full **Role-Based Access Control (RBAC)** (since v1.29) [38]. In Admin list, environment variables (`AUTHORIZATION_ADMINLIST_USERS`, `_READONLY_USERS`) define which users are admins/ viewers [39]. For fine-grained RBAC, define roles and permissions via the API. WCS supports pre-defined admin/read roles by default.

- **Data Governance:** If using cloud or hosted modules (OpenAI, Cohere, etc.), be aware that data (or chunks) will be sent externally. Ensure this complies with your privacy requirements. For highly regulated data, use on-prem modules (e.g. `text2vec-transformers`) or the Dedicated Cloud offering, which provides compliance certifications (SOC II, HIPAA) and data isolation [19].

- **Encryption:** In cloud or container environments, secure connections with TLS/SSL. Weaviate Cloud provides HTTPS by default. Self-hosted Weaviate can use an ingress or proxy with TLS termination. For encryption at rest, store data on encrypted volumes.

- **Network Security:** Bind Weaviate to appropriate interfaces (e.g. `--host 0.0.0.0` inside container). Use VPCs or private networks for communication between nodes. Control access to the API ports; consider putting an API gateway or proxy for auth and rate limiting.

- **Audit Logging:** Weaviate can log authentication and authorization events (especially with RBAC) for auditing. Check the logs (info level) for denied requests.

In summary, enforce authentication (API keys or OIDC), enable authorization (admin list or RBAC) [36] [38], and use network/storage security best practices. For hosted Weaviate Cloud, take advantage of its built-in identity and compliance features [19].

Advanced Topics

- **Fine-Tuning Embeddings:** When generic embeddings underperform on your domain, fine-tune a model on domain-specific text. For example, retrain a sentence-transformer on your company's docs. A Weaviate blog recommends fine-tuning if off-the-shelf embeddings "do not sufficiently capture domain- or company-specific semantic relationships" [40]. The process is external (e.g. using Hugging Face libraries) – once you have a fine-tuned model, you can deploy it in `text2vec-transformers` or generate embeddings offline.

- **Custom Vector Pipelines:** Weaviate allows custom flows. You could preprocess text (remove stopwords, apply stemming) before ingestion, or chain multiple vectorizers. For example, use text2vec to vectorize, then apply UMAP for dimensionality reduction before storing. These pipelines would be implemented outside Weaviate or as custom modules.

- **Multimodal Retrieval:** For advanced multimodal use, use modules like `multi2vec-bind` [8] or CLIP-based models. Weaviate can store different data types in one object and index them together. For example, a product with image and description: you can search by an image to find relevant products, or by text to find matching images.

- **Weaviate Agents:** Weaviate 1.35 introduced **Weaviate Agents**, a suite of intelligent agents (built on Weaviate APIs and LLMs) that automate data tasks [41] [42]. For example:

- *Query Agent:* Interprets natural language queries, constructs appropriate searches/aggregations, and fetches results [43]. It lets users query the database without writing GraphQL.
- *Transformation Agent:* Performs data transformations (translation, labeling, augmentation) via simple prompts [44].
- *Personalization Agent:* Recommends or re-ranks items based on user preferences [45].

These agents are "pre-trained" on Weaviate's APIs and reduce the need to build custom pipelines. They exemplify the next generation of AI-driven data management [46] [42].

- **Multi-Tenancy:** Weaviate supports multi-tenancy (tenants). Ensure you configure data tenancy if needed; note that backup and monitoring may have special modes for multi-tenant setups.

- **Performance Optimization:** Use the cache (Weaviate has an internal cache of recent vectors). Adjust Go GC or OS-level memory settings for large datasets. For extremely large index sizes, consider vector quantization (Weaviate supports PQ) to reduce memory footprint, though this may slightly affect accuracy (see Vector Quantization).

Troubleshooting

Common issues and remedies:

- **"OIDC auth not configured" (401 error):** This means Weaviate expected OpenID auth but none was set up. Check your `AUTHENTICATION_OIDC_*` environment settings. For local dev, you can disable auth or use API keys. See Authentication guide [36].

- **Backup stuck/hanging:** Ensure the backup module is enabled (`ENABLE_MODULES=backup-*`) and that the storage path or cloud bucket is writable. There were bugs in old versions (upgrade Weaviate if needed). Monitor logs for disk issues.

- **Cluster misconfiguration:** If shards are not joining, check that every node has a unique `CLUSTER_HOSTNAME` and proper network visibility. The `status` endpoint shows shard allocation. Use the Shards API to move shards if needed.

- **Low search quality:** If results are poor, try re-tuning HNSW (`ef`, `maxConnections`) or re-vectorize with a different model. Use hybrid search to incorporate keyword filtering. Check that chunking is appropriate [24].

- **Slow ingestion or queries:** Use batch import (not single-object inserts) [26]. Increase resources (CPU/memory) for modules. Scale out nodes if under heavy load.

- **Garbage/leftover data:** If you change the schema, you may need to delete and recreate the collection to purge old data. Or use the `clear` or `migrate` features cautiously.

- **Module loading errors:** Verify module availability. For example, if `text2vec-openai` gives an error, ensure `OPENAI_APIKEY` is set and that Weaviate can reach the internet (or the OpenAI API). Check logs for missing dependencies.

In general, consult the Weaviate forum or GitHub issues for specific error messages. The official docs and community provide solutions for most common issues.

Learning Path

A structured plan can help teams and individuals learn Weaviate in stages:

- **Days 1–30 (Foundations):**
- Study the **basic concepts** (vector search, schema, objects) via the Weaviate docs and tutorials.
- Install Weaviate locally (Docker Compose) and run the quickstart. Ingest a small dataset (e.g. a Wikipedia dump or product list).
- Practice simple **GraphQL queries**: nearText, hybrid searches, filters. Build familiarity with the client libraries (Python/JS).

- Learn schema design: experiment with defining classes and references. Test simple semantic search.

- **Days 31–60 (Building Use Cases):**

- Explore **modules**: enable a text vectorizer (e.g. text2vec-transformers) and try embedding your data. If possible, get API keys for OpenAI/Cohere and test those modules.
- Build a prototype application, such as:
  - A **document Q&A**: ingest FAQ articles, implement a basic Q&A using nearText.
  - A **product search**: use a sample catalog, support semantic “customer search” plus filters by category/price.
  - A **knowledge base**: ingest company wiki pages, allow FAQ search.
- Test **hybrid search** combining keyword and vector.

- Experiment with **RAG**: use Weaviate's `generate.nearText` with an OpenAI/Llama model to answer queries from your data (see code examples above). Evaluate results, adjust prompts.
- Get familiar with running in **Kubernetes** (even minikube) and use the Helm chart.

- **Days 61–90 (Advanced & Production Skills):**

- Dive into **architecture and scaling**: deploy a multi-node Weaviate cluster (K8s or VMs). Configure shards/replicas.
- Tweak **HNSW parameters** for your use case (trade memory vs recall).
- Set up **monitoring**: enable Prometheus metrics and Grafana. Observe query patterns and latency.
- Implement **security**: turn on API keys or OIDC, configure roles. Practice secure deployment (TLS, secrets for keys).
- Explore **advanced modules**: try QnA or personalization modules, or test multi-modal search with sample images.
- Learn backup/restore: do a simulated backup to S3 or local filesystem and restore it to a fresh cluster.
- (Optional) Experiment with fine-tuning: pick an embedding model to fine-tune on your domain, then use it in Weaviate.
- Document the entire process and update the schema. Create a runbook for your deployment (incl. troubleshooting common issues).

Hands-on exercises along the way (e.g., "Implement a semantic filter to answer user questions", "Add image search to your app") will cement skills. Leverage Weaviate's Academy tutorials and community for guided exercises.

Appendix

This appendix provides example code and config snippets.

Code Snippets

Python (weaviate-client): Creating a collection and adding objects.

```python
import weaviate

client = weaviate.Client("http://localhost:8080")

# 1. Create a collection/class
schema = {
    "class": "Article",
    "vectorizer": "text2vec-openai",
    "properties": [
        {"name": "title", "dataType": ["text"]},
        {"name": "content", "dataType": ["text"]},
    ]
}
client.schema.create_class(schema)
```

```
# 2. Batch import some objects
batch_data = [
    {"title": "AI in 2026", "content": "The state of AI is advanced..."},
    {"title": "Quantum Computing", "content": "Quantum computers use
qubits..."}
]
with client.batch(batch_size=10) as batch:
    for obj in batch_data:
        batch.add_data_object(obj, "Article")

# 3. Semantic search query (GraphQL)
response = client.query.get("Article", ["title", "content"])\
            .with_near_text({"concepts": ["artificial intelligence"]})\
            .with_limit(5).do()
for article in response["data"]["Get"]["Article"]:
    print(article["title"])
```

JavaScript (weaviate-client): Connecting and querying.

```
import weaviate from "weaviate-client";

const client = weaviate.client({
  scheme: 'http',
  host:   'localhost:8080',
});

// Create a collection (REST since JS client v3)
await client.schema.classCreator()
  .withClass({
    class: 'Product',
    vectorizer: 'text2vec-transformers',
    properties: [
      { name: 'name', dataType: ['text'] },
      { name: 'description', dataType: ['text'] },
    ],
  })
  .do();

// Add an object
await client.data.creator()
  .withClassName('Product')
  .withProperties({
    name: 'Wireless Mouse',
    description: 'A mouse with Bluetooth connectivity.',
  })
  .do();

// Query by vector similarity
const result = await client.graphql.get()
  .withClassName('Product')
  .withFields('name', 'description')
  .withNearText({ concepts: ['bluetooth mouse'] })
  .withLimit(3)
  .do();
console.log(JSON.stringify(result, null, 2));
```

GraphQL Templates

Example GraphQL queries:

- **Semantic search (nearText):**

```
{
  Get {
    Article(
      nearText: { concepts: ["climate change effects"] },
      limit: 3
    ) {
      title
      content
    }
  }
}
```

- **Hybrid search (text+filter):**

```
{
  Get {
    Product(
      hybrid: { query: "running shoes", alpha: 0.5 },
      where: {
        path: ["category"], operator: Equal, valueText: "Footwear"
      }
    ) {
      name
      description
      price
    }
  }
}
```

- **GraphQL mutation (add object):**

  Note: Weaviate uses REST or clients for schema/objects. The GraphQL API is read-only except for meta operations.

- **Filtering and Aggregation:** Weaviate supports `where` filters and `Aggregate` queries to get stats (count, min, etc.) on classes.

Configuration Examples

- **Docker Compose snippet (persistence):** From [57†L105-L113]:

```yaml
services:
  weaviate:
    # ...
    volumes:
      - /var/weaviate:/var/lib/weaviate
    environment:
      CLUSTER_HOSTNAME: 'node1'
      PERSISTENCE_DATA_PATH: '/var/lib/weaviate'
```

This mounts host folder `/var/weaviate` into the container for storage.

- **Helm values (excerpt):** (in a `values.yaml` for Kubernetes/Helm)

```yaml
weaviate:
  image:
    tag: 1.35.3
  resources:
    weaviate:
      requests:
        cpu: "1"
        memory: "2Gi"
      limits:
        cpu: "2"
        memory: "4Gi"
replicas: 2

authentication:
  apikey:
    enabled: true
    allowed_keys: ["admin-secret-key"]
  anonymous_access:
    enabled: false

modules:
  text2vec-transformers:
    enabled: true
  generative-openai:
    enabled: true
```

- **Weaviate Cloud (WCS):** No config needed; follow the WCS console. Ensure you note the `cluster_url` and API key for your instance.

- **Backup config (env vars):** To enable filesystem backup:

```
ENABLE_MODULES=backup-filesystem
BACKUP_FILESYSTEM_PATH=/var/lib/weaviate/backups
```

- **Monitoring (env var):** To enable Prometheus metrics:

```
PROMETHEUS_MONITORING_ENABLED=true
PROMETHEUS_MONITORING_PORT=2112
```

References: Weaviate official documentation and blogs [1] [12] [15] [7] [25] [28] [8] [34] [36] [38] provide detailed information on concepts, architecture, modules, and best practices. This handbook summarizes and consolidates that information for easy reference.

---

[1] [2] [3] Weaviate Database | Weaviate Documentation
https://docs.weaviate.io/weaviate

[4] Release Notes | Weaviate Documentation
https://docs.weaviate.io/weaviate/release-notes

[5] Data structure | Weaviate Documentation
https://docs.weaviate.io/weaviate/concepts/data

[6] [16] [17] Vector index | Weaviate Documentation
https://docs.weaviate.io/weaviate/config-refs/indexing/vector-index

[7] FAQ | Weaviate Documentation
https://docs.weaviate.io/weaviate/more-resources/faq

[8] Multimodal Retrieval-Augmented Generation (RAG) | Weaviate
https://weaviate.io/blog/multimodal-rag

[9] [10] Modules | Weaviate Documentation
https://docs.weaviate.io/weaviate/concepts/modules

[11] Hybrid search | Weaviate Documentation
https://docs.weaviate.io/weaviate/search/hybrid

[12] [28] [29] Retrieval Augmented Generation (RAG) | Weaviate Documentation
https://docs.weaviate.io/weaviate/search/generative

[13] [14] [15] Horizontal Scaling | Weaviate Documentation
https://docs.weaviate.io/weaviate/concepts/cluster

[18] [19] Weaviate Cloud | Weaviate Documentation
https://docs.weaviate.io/cloud

[20] [30] Persistence | Weaviate Documentation
https://docs.weaviate.io/deploy/configuration/persistence

[21] [36] [37] Authentication | Weaviate Documentation
https://docs.weaviate.io/deploy/configuration/authentication

[22] Quickstart: With Cloud resources | Weaviate Documentation
https://docs.weaviate.io/weaviate/quickstart

[23] [27] Cohere + Weaviate | Weaviate Documentation
https://docs.weaviate.io/weaviate/model-providers/cohere

[24] Chunking Strategies to Improve Your RAG Performance | Weaviate
https://weaviate.io/blog/chunking-strategies-for-rag

[25] Text Embeddings | Weaviate Documentation
https://docs.weaviate.io/weaviate/model-providers/openai/embeddings

[26] Batch data import | Weaviate Documentation
https://docs.weaviate.io/weaviate/tutorials/import

[31] [32] Backups | Weaviate Documentation
https://docs.weaviate.io/deploy/configuration/backups

[33] [34] [35] Monitoring | Weaviate Documentation
https://docs.weaviate.io/deploy/configuration/monitoring

[38] [39] Authorization | Weaviate Documentation
https://docs.weaviate.io/deploy/configuration/authorization

[40] Why, When and How to Fine-Tune a Custom Embedding Model | Weaviate
https://weaviate.io/blog/fine-tune-embedding-model

[41] [42] [43] [44] [45] [46] Welcome to the Next Era of Data and AI: Meet Weaviate Agents | Weaviate
https://weaviate.io/blog/weaviate-agents