TiDB

GenAI Data Pipelines:

The Engineer's Guide to Database Selection

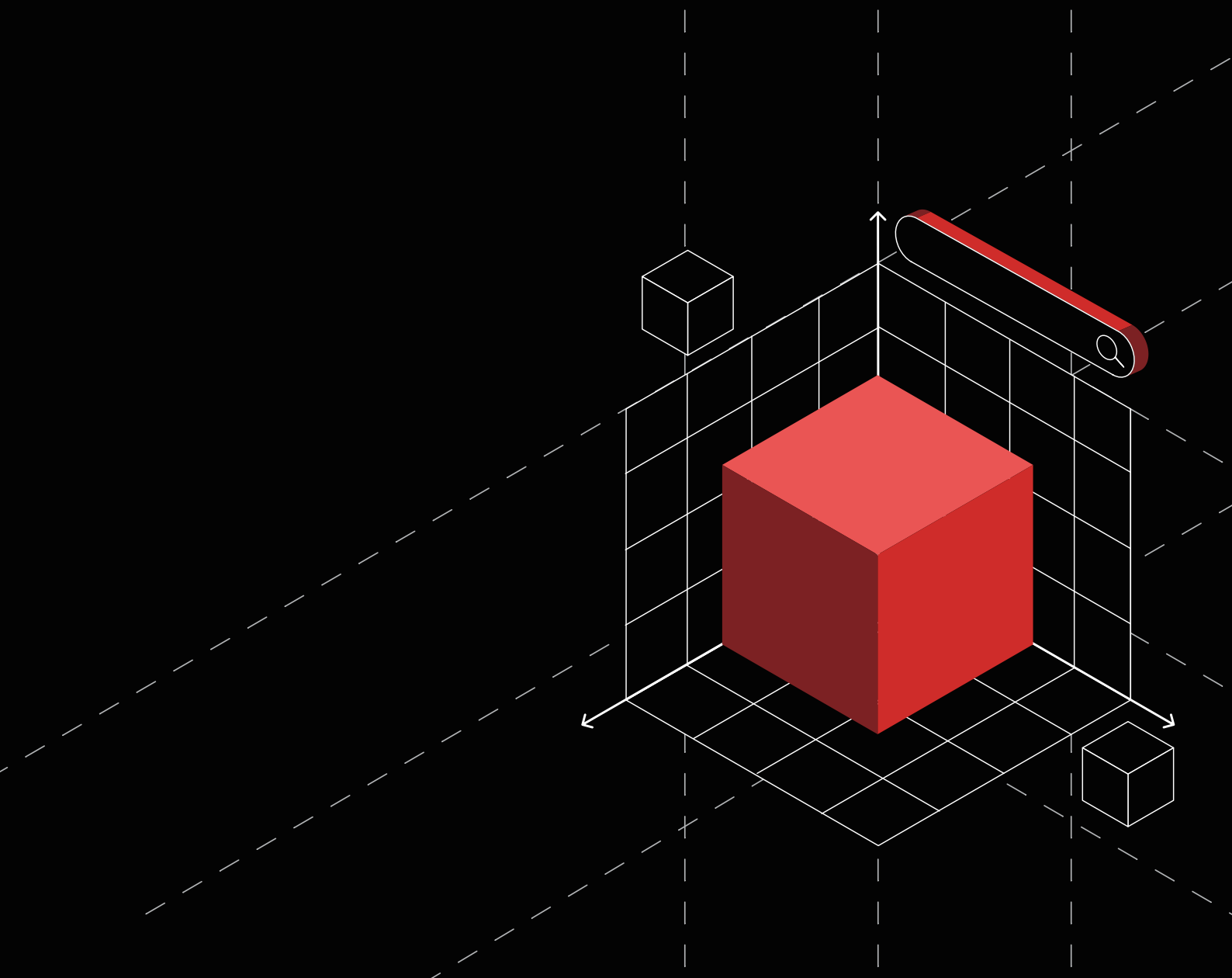


Table of Contents

Prologue

Generative AI presents us with a challenge. It has the potential to be the most transformative new technology of this century so far, while also demanding potentially disruptive changes in how we work with data.

Whether it's creating intelligent search that understands natural language, building virtual assistants that can respond to users conversationally, or analyzing large datasets to extract insights and make predictions, GenAI forces us to look for new tools, rethink our workflows, and navigate a different set of trade-offs.

In this guide, we'll look at the fundamentals of GenAI database use cases, the flow and technologies in a typical GenAI data pipeline, and share insights on selecting the right data tools for your GenAI projects.

Why traditional databases aren't up to the GenAI job

GenAI changes the rules for how we work with and think about data. Up until now, most apps have relied on getting predictable responses from data.

But with GenAI, the rules have changed. AI-based apps need to pull in massive amounts of context, process unstructured information, and generate responses that feel natural and meaningful.

The challenge is that conventional databases were designed for precise, deterministic querying and that makes it harder for them to model, store, and query data in ways that support the needs of GenAI applications.

But relational, document, time series, and other more traditional databases still have a place in GenAI data pipelines. Alongside newcomers that support vector embeddings and semantic search, they form part of a new data ecosystem.

While traditional systems handle structured data efficiently, this new landscape demands capabilities beyond what conventional databases were designed to provide.

As you assemble your GenAI data stack, here are three key areas where your combined tools need to deliver:

- **Context:** Traditional databases are great at finding exact matches, but terrible at understanding meaning. GenAI needs to know that "delivery speed" and "shipping times" are practically the same thing—something standard B-tree indexes simply can't handle. Instead, we need systems built for finding similar concepts, not just identical words.
- **Fluidity:** GenAI can't wait for overnight batch updates. It needs to absorb new information constantly and use it immediately. This requires databases that can ingest, index, and make new data available for queries almost instantly, all without sacrificing performance.
- **Structure:** AI systems need to traverse relationships that traditional data models weren't designed to express. Relational databases normalized data for consistency, NoSQL denormalized for scalability—both requiring explicit schema decisions. GenAI works with emergent structures where relationships aren't manually defined but discovered. This shift from intentional to emergent structure requires systems that can work with the fuzzy, probabilistic nature of AI rather than the deterministic logic that conventional databases were built around.

With so many cloud services and open source projects now claiming to be "GenAI-ready," it's easy to get lost in the marketing noise. What's harder is knowing which ones actually deliver.

In this report, we'll skip the hype and give you the real story—showing exactly which database technologies excel at specific GenAI workloads, where they shine brightest, and where they struggle when put to the test in production environments.

Core GenAI database concepts

Vector embeddings

This is the foundation of how AI "understands" meaning.

You are only responsible for the security of the application you develop and the security of your own business data. Based on the SaaS database service platform of TiDB Cloud, you also need to configure security for the database cluster you create, such as: setting the database cluster user role, setting the network access control of the database, doing any customizing, and selecting the database encryption method.

Semantic search

Using vector embeddings to search by meaning

Lexical search has served us well for decades, but it has clear limitations. Without manual metadata tagging, it can only match literal strings or their fragments. Semantic search brings it closer to how humans think. Instead of matching words, it uses vector embeddings to measure the conceptual distance between queries and the content in vector space. Mathematical tools such as dot product, Euclidean distance, and cosine similarity enable us to get results for “delivery time” even if the user searches for "shipping speed".

Context windows

The working memory of an LLM

LLMs have a short-term memory problem. While their training gives them vast knowledge, conversations are limited by how much new information they can track. Think of it like the difference between your laptop's storage versus its RAM, or solving complex math without paper—you know the concepts but lose track of steps.
This "context window" (typically 8,000-128,000 tokens—where a token is roughly 4 characters) can be a big influence on how you use databases in your GenAI apps. For example, you might need to break documents into smaller pieces ("chunking") so your app can pass only the relevant portions to the LLM.

GenAI database use cases

We're living through a moment of great creativity when it comes to GenAI. Solo developers, indie teams, and larger corporations are racing to build innovative applications on top of AI models. But beneath all this experimentation, how these apps actually use data tends to fall into a few recognizable patterns.

Let's look at four of the most common ways that GenAI applications put their databases to work in the real world.

RAG (Retrieval Augmented Generation)

What it is: A technique for giving an LLM the exact context it needs for your query, on top of its base training.

Why it's needed: LLMs like GPT-4 are trained on a fixed dataset. That leads to two limitations: there's a knowledge cut-off date and they don't have access to your specific data—whether that's company policies, individual customer history, or industry-specific insights. RAG overcomes this by dynamically retrieving chunks of text from your database (this is the R in RAG) and then adding them to your query (the AG). That way, the LLM gets additional context so that it can answer questions that aren't covered by its existing training.

Example: A customer service chatbot that uses individual customer history and similar cases to provide the LLM with data that isn't available in its training.

How it works with databases: RAG depends on databases to store, search, and retrieve relevant context at query time. This often involves vector databases for semantic search, relational databases for more structured data, or both.

A typical RAG workflow looks like this:

- **Analyze the query:** Decide what type of additional data will best support the query. This is where you'll choose what dataset to rely on and, by extension, the type of database you'll be using to retrieve it.
- **Retrieve the data:** Let's say you're looking for context from customer chat histories. You might:
    - **Query a document or relational database directly:** If you need a specific customer's chat history.

    - **Run a semantic search in a vector database:** If you want to retrieve multiple customer conversations that cover a particular topic.
- **Make the augmented query:** Once you have the additional context, add it to the original prompt you submit to the LLM. Typically, you'll structure the prompt to guide the LLM on how to use the additional information provided by the RAG process.

Query speed and efficient indexing are critical—slow retrieval means slow AI responses, directly impacting user experience.

Best database types for RAG:

- **Vector databases:** Make the semantic connections needed by RAG. They index embeddings and find content based on similar meaning, enabling AI to discover relevant information even when keywords don't match.
- **Document stores:** Handle the actual content chunks fed into your LLM during augmentation. While great at managing unstructured text at scale, they need vector capabilities layered on top to make that content semantically accessible for RAG.
- **Relational databases:** Add precision when your AI responses require structured data points. This is handy when you need to ground responses in exact facts, such as customer records, product details, or financial data.
- **Time series databases:** Become valuable only when temporal patterns matter to your use case, such as industrial IoT sensor readings. Most RAG implementations won't need this specialized capability unless your domain requires historical trend analysis or time-sensitive context.
- **Databases with integrated vector search:** Support similarity search by offering multiple ways to query existing data, including vector search of document or relational data. This differs from extensions like pgvector for PostgreSQL in that vector search is a first-class part of the software.

Similarity search

What it is: Similarity search is the foundational use case for databases in GenAI applications. Not only does it enable RAG systems to find relevant context, but it powers everything from semantic search to recommendation engines. While techniques like TF-IDF (Term Frequency Inverse Document Frequency) and LSH (Locality Sensitive Hashing) enabled similar functionality with traditional databases, they relied on manual feature engineering and explicit similarity definitions. With vector databases, it's the embedding process that allows those semantic relationships to emerge naturally from the data.

Why it's needed: Lexical search can miss the mark when users expect results that understand intent rather than just matching words. Whether it's powering semantic search, recommendations, or anomaly detection, similarity search helps surface the most relevant results—even when the wording isn't an exact match.

Example: A music streaming service that identifies songs with similar acoustic patterns to your favorites, discovering tracks you'll enjoy based on audio characteristics rather than just metadata tags.

How it works with databases:

1. **Convert content to vectors**: Both your database content and search queries are transformed into vector embeddings—the numerical representations of meaning that we looked at earlier.
2. **Measure distances between vectors:** When a user searches, the system calculates how "close" their query vector is to each data vector in the database. Similar concepts have vectors that are closer together.
3. **Use approximation for speed:** Since exact distance computation across millions of vectors would be too slow for most applications, vector databases use specialized indexing approaches like HNSW (Hierarchical Navigable Small World) to efficiently find the closest matches.
4. **Combine with metadata filtering:** Most real-world applications narrow vector search with filters—limiting results by date range, category, or other metadata attributes before performing similarity calculations.
5. **Return ranked results:** The system gives you back those data points whose vectors are closest to the query, ranked by similarity.

Best database types for similarity search:

- **Vector databases:** Purpose-built for embedding storage and retrieval, providing the performance and specialized indexing needed for large-scale similarity operations that power GenAI applications.
- **Document stores:** House the primary content and metadata, with vector capabilities increasingly built in to enable semantic search directly on the data repository without maintaining separate systems.
- **Relational databases:** Where your data lives primarily in a relational database, you can use a dedicated vector database or vector extensions to support similarity search over existing relational models.
- **Time series databases:** When paired with a vector database to enable similarity search across temporal data, converting time-series segments into embeddings, this allows you to find patterns in sensor data, market movements, or user behavior that are semantically similar despite differences in absolute values or timing.
- **Databases with integrated vector search:** Combine vector search with either the relational or document model in a single system. This simplifies your architecture and can reduce latency.

Fine-tuning and model training

What it is: Pre-trained LLMs are great generalists, but they don't start off knowing the specifics of your domain. Fine-tuning updates a model's internal weights by training it on your proprietary data, making it permanently better at understanding specialized terminology, company policies, or unique workflows.

Unlike RAG, which retrieves external information at query time, fine-tuning bakes knowledge directly into the model, improving its baseline understanding and response quality.

Why it's needed: RAG can be a good first step when it comes to providing the LLM with information it wouldn't have otherwise. But it's not always possible to include all the data you need within a model's context window. Fine-tuning requires some up-front preparation but, in return, it can deliver more precise responses with few hallucinations. That makes it a good fit for applications where precision is important, such as legal, healthcare, and finance.

Example: A radiology GenAI app trained on a hospital's database of labeled scans that learns to detect specific conditions with higher accuracy than generic models, adapting to the facility's equipment and patient demographics.

How it works with databases: Fine-tuning depends on high-quality, well-structured training data, typically pulled from a mix of structured and unstructured sources. The database architectures you choose will directly impact your training pipeline's efficiency and the resulting model quality.

A typical fine-tuning workflow looks like this:

1. **Data collection and preparation:** Extract relevant examples from your databases. This could be question-answer pairs, categorized documents, or specialized domain content.
2. **Data transformation and cleaning:** Convert raw data into training formats, often involving filtering, deduplication, and normalization.
3. **Training data validation:** Analyze your dataset for quality issues, bias, or gaps in coverage. Vector databases can help identify semantic redundancies or underrepresented concepts in your training data.
4. **Batch processing for training:** During the actual fine-tuning process, your database must efficiently deliver batches of examples to the training pipeline. Slow data retrieval becomes the bottleneck that limits GPU utilization and extends training time.
5. **Model evaluation with test data:** Your database needs to efficiently store and retrieve separate test examples that weren't used in training. This allows you to regularly check if your model is actually improving on real-world data, not just the examples it trained on.

You can perform fine-tuning on open source models you host yourself and on public models through dedicated APIs.

Best database types for similarity search:

- **Vector databases:** Analyze your training data's semantic coverage to identify gaps and redundancies. When fine-tuning a legal AI, vector databases can visualize how your contract examples cluster in embedding space—revealing that you have abundant NDAs but lack partnership agreements entirely. This semantic analysis helps create balanced datasets that cover your domain thoroughly without wasting compute on repetitive examples that don't improve model performance.
- **Document stores:** Manage the unstructured text that forms the foundation of most fine-tuning datasets. A healthcare company fine-tuning on medical literature uses

document stores to organize millions of articles and clinical notes, track which content has been processed, and efficiently deliver training batches to the pipeline. This becomes crucial as datasets grow to terabytes in size where file-based approaches begin to break down.

- **Relational databases:** Maintain factual consistency across training examples by connecting them to their source of truth. Customer service AI training uses relational structures to link examples to specific products and policies they reference. When information changes, this makes it easy to identify and update affected training examples, preventing the model from learning contradictory or outdated information that leads to hallucinations.
- **Time series databases:** Preserve chronological context essential for understanding causality in sequential data. For example, a utility company looking to create forecasting models leverage time series databases to maintain the temporal relationships between weather patterns, consumption rates, and grid demand. This allows models to learn that temperature spikes precede demand surges by specific intervals—causal patterns that would be lost if examples were processed as isolated data points.
- **Databases with integrated vector search:** Simplify training workflows when examples combine different data types. Retail recommendation engines, for example, benefit from databases that combine a more typical operational data model with vector search. They seamlessly integrate product descriptions (text), purchase history (relational), and visual similarity (vector) in a unified system.

Memory for Stateful AI Agents

What it is: A database-driven system that gives LLMs persistent memory across interactions by storing and retrieving conversation history, user preferences, and interaction context. Think of it as a specialized form of RAG focused on user-specific data rather than general knowledge.

Why it's needed: LLMs exist in a world between stateless and stateful. They can maintain context within a single session, up to a point, but even then you can't rely on them to remember things faithfully. Between sessions, LLMs are entirely stateless.

But that doesn't have to limit the scope of your GenAI apps. Instead, you can use databases to store context between sessions and then replay that as part of the prompt you give the model.

This is particularly important for applications requiring continuous interaction or relationship-building, like virtual assistants, customer support, or educational tools.

Example: A healthcare companion chatbot that remembers your specific health history. So, you can ask, “Is my blood pressure lower this month?” and get an accurate answer without having to re-explain your medical situation.

How it works with databases: Think of this as "personal RAG". Instead of retrieving more general knowledge, you're retrieving a specific user's history. Your database becomes the AI's long-term memory, storing conversations and preferences that would otherwise be forgotten. When a user asks a question, your app first checks the database for relevant history before sending the prompt to the LLM.

Best database types for similarity search:

- **Vector databases:** Implement "memory search" where agents need to retrieve relevant past interactions based on meaning rather than exact terms. They excel at answering queries like "What did I tell you about my project timeline?" by finding semantically similar conversations without requiring precise keywords.
- **Document stores:** Ideal for storing complete conversation histories in sequence, preserving the full context of past interactions. Their schema flexibility allows for evolving conversation structures and metadata as the agent's capabilities grow, without requiring database restructuring.
- **Relational databases:** Power the factual knowledge base of AI agents, maintaining structured records of facts about users and their preferences. They particularly shine in long-term relationships where the agent needs to track changing user details across dozens or hundreds of interactions.
- **Databases with integrated vector search:** Solve the challenges of latency and architectural complexity by integrating different data models in one system. For responsive AI agents that need sub-second recall, combining vector search with structured and unstructured storage in a unified database keeps perceived intelligence high by eliminating the delays from coordinating across multiple specialized systems.

Database technologies powering genAI applications

Building generative AI applications requires both specialized databases and familiar systems used in new ways. This means evolving roles for your relational and document databases, and potentially introducing dedicated vector databases.

For many teams, that means choosing between adding vector capabilities to your existing operational data or adding a dedicated vector database to your stack. Each option involves different tradeoffs in terms of performance, operational complexity, and development workflow

Vector databases

What they are: Purpose-built engines for storing, indexing, and querying vector embeddings—the numerical representations of text, images, or audio that capture semantic meaning in high-dimensional space.

Key strengths:

- **Get results even with imprecise queries:** Deliver relevant results based on semantic matching rather than lexical matching.
- **Data structure is emergent:** Feeding your data through an embedding algorithm automatically creates a semantic map of your data in vector space.
- **Improve with metadata**: You can add to the vector search with metadata to filter results by category, date range, or other attributes, giving you more control over the results.
- **Query performance at scale:** Using specialized algorithms and indexing, most vector databases can maintain millisecond response times even with millions of vectors.

Notable limitations:

- **Limited data manipulation:** Most standalone vector databases act as an index to your data, helping you retrieve data, but they lack support for complex transactions, updates, and joins that traditional databases provide.
- **Trade-offs between accuracy and performance:** Algorithms like Approximate Nearest Neighbor sacrifice some precision for speed, which can impact result quality.
- **Operational overhead:** Running specialized vector databases alongside operational databases adds monitoring, scaling, and maintenance complexity.
- **Evolving ecosystem:** Standards, best practices, and tooling are still maturing compared to established database technologies.

Example databases:

- **Pinecone:** A proprietary vector database service that focuses on simplicity and scalability. Provides serverless deployment options, specialized indexing algorithms, and REST APIs for integrating vector search into applications without managing infrastructure.
- **Milvus:** An open-source vector database with strong community support. Offers flexible deployment options (self-hosted or cloud), multiple indexing algorithms, and scalar filtering capabilities. Designed for high throughput and horizontal scaling.
- **Qdrant:** Open-source vector database with an emphasis on extended filtering capabilities and flexible payload storage alongside vectors. Features include binary quantization for storage optimization and a lightweight footprint suitable for both cloud and edge deployments.

Document stores

What they are: Schema-free and built for flexibility, these databases handle JSON and unstructured data, making them ideal for storing both the text that models train on and the context they retrieve.

Key strengths:

- **RAG-optimized storage:** Excel at storing and retrieving the full text chunks that LLMs need for context windows, with flexible schemas for both content and metadata.
- **Semantic chunking management:** Maintain relationships between chunks, parent documents, and their vector representations, crucial for effective retrieval augmentation.
- **Hybrid search capabilities:** Many document stores now integrate vector search alongside traditional text search, enabling combined lexical and semantic matching in a single query.
- **Adaptation without migration:** Support evolving AI application needs without disruptive schema changes or migrations.

Best for these AI use cases:

- **Chunk optimization challenges:** Requires careful tuning of chunk sizes to balance context relevance with retrieval precision
- **Complex relationship modeling:** May struggle with maintaining the complex relationships between entities that LLMs reference
- **Consistency trade-offs:** Some document stores sacrifice strong consistency for performance, complicating certain AI workflows
- **Read/write balance:** RAG workloads tend to be read-heavy, requiring different optimization strategies than traditional applications

Notable limitations:

- **Limited data manipulation:** Most standalone vector databases act as an index to your data, helping you retrieve data, but they lack support for complex transactions, updates, and joins that traditional databases provide.
- **Trade-offs between accuracy and performance:** Algorithms like Approximate Nearest Neighbor sacrifice some precision for speed, which can impact result quality.
- **Operational overhead:** Running specialized vector databases alongside operational databases adds monitoring, scaling, and maintenance complexity.
- **Evolving ecosystem:** Standards, best practices, and tooling are still maturing compared to established database technologies.

Relational databases

What they are: Your app's system of record also has a role in supporting GenAI operations. Whether through RAG or finetuning, the data in your relational database can augment your LLM prompts to give you more accurate responses grounded in the factual data of your application. The backbone of application development for the past few decades, relational databases have a new role in GenAI pipelines, particularly where consistency, transactions, and integration with existing systems are critical.

Key strengths:

- **Structured knowledge bases:** Excel at providing factual, structured information to ground LLM outputs, reducing hallucination through reliable data retrieval.
- **Strong consistency and transactions:** Ensures data integrity for AI operations where accuracy is non-negotiable.
- **Precise querying:** Complex SQL queries can retrieve exactly the structured data needed for AI context, improving relevance.

Best for these AI use cases:

- **Fine-tuning:** Managing structured training datasets with high quality requirements, tracking data provenance and annotations.
- **Memory for AI agents:** Tracking user profiles and preferences reliably.
- **RAG for structured data:** Retrieving structured information like product catalogs or financial data.
- **Orchestration and tracking:** Logging prompt execution history, model performance statistics, and usage patterns.

Notable limitations:

- **Query complexity for semantic contexts:** Constructing SQL queries that effectively retrieve relevant context for LLMs often requires complex joins and query logic.
- **Structured vs. unstructured tension:** Relational schemas are great for structured data but can become unwieldy when handling the variable-length text chunks needed for RAG.
- **Semantic understanding gaps:** Relational models represent explicit relationships well but struggle with the implicit semantic connections that embedding models capture naturally.

Databases with integrated vector search

What they are: The practical default for most GenAI applications: familiar relational or document databases that have added vector embedding capabilities. Rather than introducing an entirely new database technology, these solutions let you use your existing operational database systems with extensions that add vector search.

Key strengths:

- **Easy to adopt:** Add GenAI capabilities to your stack without learning, integrating, and operating an entirely new database system.
- **Unified data management:** Keep your application data and vector embeddings in sync automatically without complex ETL processes.
- **Simplified infrastructure:** Maintain one database system instead of separate operational and vector databases.
- **Reduced end-to-end latency:** Eliminate network hops between separate operational and vector databases, keeping response times low.

Notable limitations:

- **Performance ceiling:** May not match specialized vector databases for applications requiring extreme scale (billions of vectors).
- **Feature depth trade-offs:** Might lack some advanced vector indexing algorithms or distance metrics available in dedicated solutions.

Example databases:

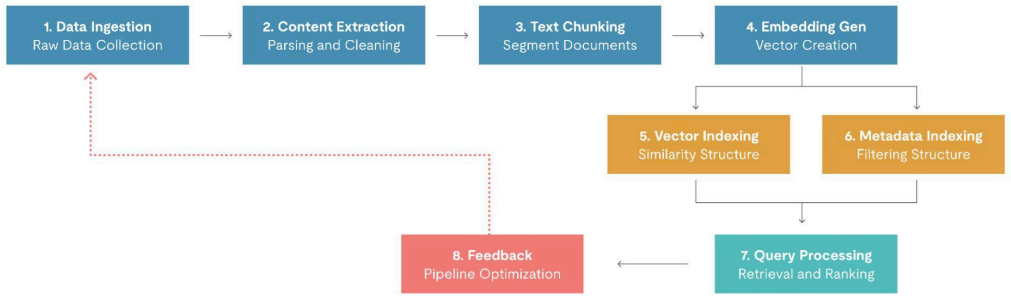
- **TiDB Serverless:** A cloud-native SQL database that combines OLTP, OLAP and vector search capabilities with automatic scaling and pay-as-you-go pricing.
- **PostgreSQL + pgvector:** A popular option for teams with existing PostgreSQL deployments, providing vector search capabilities integrated with a familiar relational engine.
- **MongoDB Atlas with vector search:** Combines document flexibility with integrated vector operations for teams already in the MongoDB ecosystem.

Typical ETL (Extract, Transform, Load) pipelines are mostly about moving data so it can be used in a different context. The GenAI data pipeline not only moves data but finds the meaning within it. The result is that, while traditional ETL focuses on data correctness, GenAI pipelines need to preserve meaning with ongoing improvements.

How the GenAI data pipeline differs from ETL

Traditional ETL	GenAI Pipeline	Why It Matters
Data correctness	Context preservation	GenAI needs narrative flow, not just accurate values
Schema-driven	Recursive improvement	Systems learn from usage patterns and feedback
Schema-driven	Meaning-aware	Semantic relationships emerge rather than being defined

The stages of the GenAI data pipeline



Data ingestion

The source and quality of your data fundamentally determine what your AI can understand. This first stage collects raw data from various sources and prepares it for processing in the pipeline.

- **Common pitfall:** Ingesting low-quality data that later propagates through the entire pipeline
- **Tools and technologies:**
  - **Connectors:** Airbyte, Fivetran, custom API clients
  - **Data lakes:** S3, Azure Blob Storage, Google Cloud Storage
  - **Streaming platforms:** Kafka, Kinesis for real-time data ingestion
- **Considerations:**
  - Source reliability and data freshness
  - Data quality validation at ingestion time
  - Handling different file formats and encodings
  - Real-time vs. batch ingestion requirements

Content extraction and cleaning

Structured, clean data is essential for effective downstream processing.
This stage parses raw content, extracts text and metadata, and standardizes formats for further processing.

- **Common pitfall:** Losing valuable context during cleaning processes
- **Tools and technologies:**
  - **Document parsers:** Apache Tika, Unstructured.io, BeautifulSoup
  - **ETL tools:** dbt, Apache NiFi
  - **Cloud services:** AWS Textract, Google Document AI
- **Considerations:**
  - Text normalization and noise removal
  - Maintaining document structure during extraction
  - Handling multi-modal content (text, tables, images)
  - Balancing processing depth with throughput requirements

Text chunking and segmentation

How you divide content directly impacts retrieval effectiveness.

This stage breaks down documents into optimally sized pieces that can be embedded and later retrieved.

- **Common pitfall:** Creating chunks that are either too large (diluting relevance) or too small (losing context)
- **Tools and technologies:**
    - **Text processors:** LangChain text splitters, NLTK sentence tokenizers
    - **NLP libraries:** spaCy, Stanza for linguistic-aware splitting
    - **Custom processors:** Domain-specific chunking algorithms
- **Considerations:**
    - Chunk size optimization for context windows
    - Semantic vs. fixed-length chunking approaches
    - Document structure preservation across processing
    - Handling structured data and non-text elements

Embedding generation

Converting text to vector representations enables semantic understanding.

This stage transforms text chunks into numerical vectors that capture semantic meaning.

- **Common pitfall:** Using generic embedding models for domain-specific content
- **Tools and technologies:**
    - **Embedding models:** OpenAI's text-embedding-ada-002, BERT, BGE
    - **Batch processors:** Ray, Apache Spark for distributed processing
    - **Vector optimization:** Dimensionality reduction, quantization tools
- **Considerations:**
    - Chunk size optimization for context windows
    - Semantic vs. fixed-length chunking approaches
    - Document structure preservation across processing
    - Handling structured data and non-text elements

Vector indexing and storage

Efficient vector storage and indexing directly impacts retrieval performance.

This stage organizes vectors in specialized data structures optimized for similarity search.

- **Common pitfall:** Choosing a standalone vector database when an operational database with integrated vector search would be more efficient.
- **Tools and technologies:**
  - **Vector databases:** Pinecone, Milvus, Qdrant, Weaviate
  - **Operational databases with integrated vector search:** TiDB Serverless, PostgreSQL with pgvector, MongoDB Atlas Vector Search
  - **Indexing algorithms:** HNSW, IVF, FAISS indexes
- **Considerations:**
  - Index type selection based on query patterns
  - Query latency requirements at production scale
  - Index build time vs. query performance tradeoffs
  - Storage requirements and scaling strategy

Metadata indexing

Combining vector search with metadata filtering enables precise retrieval.

This parallel stage indexes document metadata to support hybrid search strategies.

- **Common pitfall:** Overlooking metadata's role in filtering and enhancing retrieval
- **Tools and technologies:**
  - **Search engines:** Elasticsearch, OpenSearch, Solr
  - **Relational databases:** TiDB Serverless, PostgreSQL, MySQL for structured metadata
  - **Vector search solutions:** Vector search with metadata filtering
- **Considerations:**
  - Schema design for filtering efficiency
  - Integration between vector and metadata search
  - Metadata schema flexibility for evolving requirements
  - Index optimization for common query patterns

Query processing and retrieval

Transforming user questions into effective queries determines result quality.

This stage converts input questions into search queries that use both vector and metadata indexes.

- **Common pitfall:** Relying solely on vector similarity without metadata filtering
- **Tools and technologies:**
    - **Retrieval frameworks:** LangChain, LlamaIndex
    - **Rerankers:** Cohere Rerank, Cross-encoders
    - **Query routers:** Multi-index routing systems
- **Considerations:**
    - Query expansion and hybrid search strategies
    - Balancing recall vs. precision in results
    - Performance optimization for complex queries
    - Re-ranking and result diversity approaches

Feedback integration and pipeline optimization

Continuous learning from query patterns and user feedback improves performance.

This final stage captures performance metrics and user feedback to optimize earlier pipeline stages.

- **Common pitfall:** Building a static pipeline without feedback incorporation
- **Tools and technologies:**
    - **Analytics platforms:** Weights & Biases, MLflow
    - **Feedback collectors:** Argilla, custom feedback loops
    - **A/B testing frameworks:** Optimizely, LaunchDarkly
- **Considerations:**
    - Feedback collection methodologies
    - Metrics definition for pipeline performance
    - Identifying bottlenecks across pipeline stages
    - Balancing stability with continuous improvement

Making the right GenAI database choice in 3 steps

Step 1: What is your primary use case?

- **RAG (Retrieval Augmented Generation):** Dynamically retrieving relevant context from your data to augment LLM prompts.
- **Similarity search:** Using vector embeddings to find semantically related content rather than exact keyword matches.
- **Fine-tuning and model training:** Creating, managing, and processing datasets for specialized model training.
- **Memory for stateful AI agents:** Maintaining persistent context and user-specific information across interactions.

Step 2: Assess your operational reality

Practical factor	Questions to ask
Existing database investment	What databases already power your core applications?
Team expertise	What database skills does your team already have?
Operational overhead	Do you have the capacity to integrate and manage another system?
Data consistency	How important is it that AI features use the latest operational data?
Total cost of ownership	What are the full costs of additional systems versus extending existing ones?

Step 3: Make your technology decision

Approach	Best when	Not ideal when
Add vector capabilities to existing DB	• Your app already uses relational/document DB• You need AI features that use operational data• Consistency between AI and app data is critical• You want to minimize infrastructure complexity	• You need specialized vector algorithms beyond basic nearest-neighbor search• You're processing billions of vectors with strict latency requirements
Specialized vector database	• You're building a pure AI application• You need sub-10ms search across billions of vectors• Your vectors rarely change and don't need strong consistency with operational data	• Your AI features need to join with operational data• You want to minimize infrastructure complexity• Your team lacks bandwidth to maintain another system

Unifying your GenAI data infrastructure with TiDB Serverless

We've seen how apps built around generative AI have complex data storage and processing needs. That can mean bringing together multiple database tools, with the learning curve, integration, and operational headaches associated.

The challenge

Most GenAI applications require:

- Vector storage for embeddings and semantic search
- Relational data for operational needs
- Document stores for unstructured content

Managing these separately can lead to data synchronization issues, operational complexity, and consistency problems.

TiDB Serverless: Simplifying your GenAI data architecture

Integrated vector search and SQL

- Store and query embeddings alongside relational data
- Combine semantic similarity with precise SQL filtering
- Apply business logic without crossing system boundaries

Developer-friendly experience

```
# Search for similar products within a category
results = session.execute(
select(Product.name, Product.description)
.where(Product.category == "electronics")
.order_by(Product.embedding.cosine_distance([0.1, 0.2, ...]))
.limit(5)
).all()
```

Operational advantages

- Automatic scaling with zero cost when idle
- No capacity planning or provisioning required
- High availability across availability zones
- Pay only for actual usage (storage and request units)

Real-world impact: Dify.AI

Dify.AI, a leading LLMOps platform serving hundreds of thousands of users, unified their vector and relational data with TiDB Serverless, achieving:

- Consistent performance across 100M+ records
- Simplified infrastructure management
- Reduced engineering resources
- Predictable scaling with variable workloads

"With TiDB, our users can concentrate on building their GenAI apps rather than worrying about setup. The scale-to-zero capability lets us provide dedicated databases without the burden of idle resource costs."

— Luyu Zhang, Founder & CEO, Dify.AI

When TiDB Serverless makes sense

Consider TiDB Serverless when your GenAI project:

- Needs both operational data and vector capabilities
- Requires consistent data between vectors and structured records
- Has variable workload demands
- Would benefit from simplified infrastructure management

By unifying your GenAI data infrastructure, you can focus on building intelligent applications rather than managing the complexity of multiple specialized databases.

Appendix

Your guide to GenAI database terminology

Generative AI comes with its own terminology. While much of it will be familiar or easy to decode, there's a glossary of the terms you'd like to encounter when working with data for GenAI applications.

Fundamentals

- **Embedding:** A numerical vector that represents the meaning of text, images, or other data. Similar content has similar embeddings, making it possible to find similar content.
- **Vector database:** A database optimized for storing, indexing, and querying embedding vectors using specialized algorithms that enable similarity search at scale.
- **RAG (Retrieval Augmented Generation):** A technique that pulls relevant information from a database to give an LLM the context it needs for a specific query, rather than relying solely on its training data.

Vector storage and retrieval

- **Vector indexing:** Specialized data structures that organize vectors for efficient similarity search, unlike traditional B-tree **indexes used in relational databases.**
- **ANN (Approximate Nearest Neighbor):** Algorithms that find similar vectors quickly by trading perfect accuracy for speed—essential for production GenAI applications where milliseconds matter.
- **Distance metrics:** Methods for measuring how similar two vectors are, like cosine similarity (which compares the angle between vectors) or Euclidean distance (which measures straight-line distance). For example, cosine similarity is useful in text search to find documents with similar meaning, while Euclidean distance is often used in image retrieval to find visually similar items.
- **Vector filtering:** Adding metadata constraints to vector searches, such as "find content similar to this query, but only from this year's documents."

RAG database concepts

- **Chunking strategies:** Ways to break documents into smaller pieces for embedding and retrieval, balancing context preservation with precision. For example, chunking a legal

contract into small 100-token pieces helps pinpoint specific clauses precisely, but loses surrounding context. Larger 1000-token chunks preserve context but may dilute relevance scoring by including too much extraneous text.

- **Retrieval context window:** The maximum amount of retrieved text that can be included in a prompt to an LLM before hitting token limits. This constraint directly impacts how your database should chunk and retrieve content—smaller chunks allow for more diverse context but risk fragmenting important information, while larger chunks preserve more context but limit how many distinct sources you can include
- **Hybrid search:** A technique combining traditional keyword search (finding exact text matches) with vector similarity search (finding semantic similarities) to deliver more comprehensive results. For example, searching for "database migration" would match both exact keyword occurrences and conceptually related content about "moving data between systems" that keyword search alone might miss. Not to be confused with hybrid operational/analytical database architectures.
- **Document metadata:** Additional information stored with embeddings that enables filtering, attribution, and versioning in GenAI applications.

Integration patterns

- **Embedding pipeline:** The workflow for converting raw content (like text or images) into vector embeddings and storing them in your database. This typically includes steps for cleaning and preprocessing content, sending it through an embedding model to generate vectors, and then saving those vectors with appropriate metadata for later retrieval.
- **Vector synchronization:** How to keep embeddings up-to-date when source content changes, without rebuilding your entire index.
- **Semantic caching:** Storing results of similar previous queries to speed up responses and reduce database load.
- **Cross-encoder reranking:** A technique that improves search quality by using a second model to score and reorder initial database results. While databases quickly retrieve candidates using vector similarity, the cross-encoder evaluates each (query, result) pair more thoroughly to provide more accurate relevance rankings—trading speed for precision in the final results shown to users.

Performance considerations

- **Recall@k:** The percentage of truly relevant documents your database retrieves within the top-k results. For example, if there are 10 relevant documents for a query and your system finds 8 of them in the top-20 results, your recall@20 is 80%. This metric directly impacts how comprehensively your GenAI application can access relevant information.
- **Vector cardinality:** The total number of vector embeddings stored in your database. A production RAG system might contain millions of vectors, each representing a document chunk. As cardinality increases, query performance typically degrades unless your database uses specialized indexing techniques to maintain speed at scale.
- **Dimensionality:** The number of numerical values in each vector (typically 768-1536 for today's embedding models). Higher dimensionality vectors capture more nuance but take up more storage and processing. For example, a database with 1 million chunks using 1536-dimensional vectors requires at least 6GB of raw vector storage before indexing overhead.
- **Quantization:** Techniques for compressing vectors by reducing numerical precision while preserving similarity relationships. For instance, converting 32-bit floating-point numbers to 8-bit integers can reduce storage requirements by 75% with minimal impact on retrieval quality, allowing databases to handle larger document collections with less impact on resources.

TiDB

tidb.io/ai