CODE B

The AI Database Architecture

How to Design the Data Layer for Production Machine Learning Systems

A practical engineering guide for ML engineers, data architects, and technical leads.

2026 Edition | code-b.dev

Executive Summary

Choosing the right database for a machine learning system is one of the most consequential infrastructure decisions a team can make and one of the most frequently underestimated. Most teams make the choice once, early, based on familiarity rather than fit, and then spend months dealing with the consequences: slow training pipelines, inference latency that breaks user experience, models that perform differently in production than in evaluation, and data that sits in the wrong place at the wrong time.

This playbook exists to change that. It gives ML engineers, data architects, and technical leads a practical, structured framework for designing the database layer of a production AI system — from the first line of raw data ingestion through to real-time model serving.

The core argument is simple: the ML data layer is not a single database decision. It is a series of decisions, one per pipeline stage, each with its own access pattern, latency requirement, and failure mode. Getting each decision right — and understanding how the pieces fit together — is what separates a system that scales cleanly from one that requires a painful re-architecture six months after launch.

What This Playbook Covers

- Why the ML pipeline has five distinct data access patterns, and why no single database handles all five well
- How to design the right database for each stage: ingestion, feature engineering, feature serving, embedding retrieval, and batch training
- Three reusable architecture patterns, Lean Stack, Growth Stack, and Enterprise Stack, are mapped to team size and scale
- A practical vector database selection framework with six criteria that actually matter in production
- When in-database ML eliminates the need for a separate training pipeline entirely
- The six most common production database challenges and how to handle each one
- A 12-question checklist to validate your architecture before you build

Three Common Mistakes This Playbook Helps You Avoid

- Using a data warehouse as a feature store, then wondering why inference latency is 400ms instead of 4ms
- Adding a vector database without a clear embedding refresh strategy, and ending up with a RAG system that returns stale, irrelevant results
- Building on a single database because it handles everything 'well enough' and then hitting a hard scale ceiling at exactly the wrong moment

Who This Guide Is For:

ML engineers are architecting production pipelines. Data architects evaluating infrastructure for an AI product. Technical leads and CTOs are making database decisions that will affect the

system for the next 2 to 3 years.

Chapter 1: The ML Data Layer Problem

The most common database architecture mistake in ML projects is not choosing the wrong database. It is assuming that one database is the right answer at all.

This assumption is understandable. In most software systems, a single database handles everything: user data, transactions, and application state. PostgreSQL or MySQL gets chosen early, and the team builds on top of it.

That works well for conventional applications because the access patterns are similar across the system: reads and writes of moderate size, transactional guarantees, and predictable latency.

ML systems are different. The same data has to serve radically different purposes at different points in the pipeline, and those purposes have conflicting requirements.

The Five Access Patterns in a Production ML System

Every ML pipeline passes data through five distinct stages, each with its own performance contract:

- For data ingestion, you need to accept data fast, from multiple sources, in formats that change over time. The priority is write throughput and schema flexibility. A rigid schema that requires migrations every time a new event type appears will slow down the entire data team.
- Feature engineering; Raw data has to be transformed into model-ready features. This means complex joins, aggregations, and transformations across large datasets. SQL databases excel here because the access pattern is analytical: read large volumes, compute over them, and produce a clean output.
- Feature serving; At inference time, the model needs its input features in under 10 milliseconds. Recomputing features on the fly is too slow. Pre-computed features need to live in a low-latency store that can return a user's profile, recent history, or session context in a single key lookup.
- Embedding retrieval; In retrieval-augmented generation (RAG) applications, the model needs to find the most semantically similar documents from a large collection before generating a response. This is a nearest-neighbor search problem over high-dimensional vectors, which standard databases were not designed for.
- Batch training; Training a model requires scanning enormous datasets, often repeatedly. You need high read throughput on compressed, columnar data, and ideally the ability to run the computation close to where the data lives rather than moving it across a network.

The Cost of Getting It Wrong

These five access patterns have genuinely incompatible requirements. A database optimized for millisecond key-value lookups (Redis) is not designed for petabyte-scale analytical queries. A

columnar warehouse built for batch training (BigQuery) will introduce 300ms of latency if you try to use it as a feature store at inference time.

When teams force one database to serve all five patterns, they hit specific, predictable failure modes:

- Training bottleneck – Data is stored in a transactional database (PostgreSQL or MySQL), not designed for full-table analytical scans. Training jobs take 10 times longer than they should because the database was built for row-level reads, not column-level aggregations over billions of records.
- Inference latency spike - Features are retrieved from a data warehouse at serving time. The model makes accurate predictions in evaluation, but the response time in production is 400ms, when users expect under 50ms.
- Training-serving skew Features are computed one way during training and a different way at inference time because two different systems are involved. The model performs well in offline evaluation but degrades in production. This is one of the most common and hardest-to-diagnose ML production problems.
- Stale retrieval: A RAG system uses a vector database that was loaded at launch but never updated. The knowledge base is months out of date. The LLM confidently returns answers based on stale information.

Understanding these failure modes is the first step to avoiding them. The rest of this playbook shows you how to design around each one.

Chapter 2: Designing for Each Stage of the ML Pipeline

Each stage of the ML pipeline has the right database category. This chapter works through all five, covering what each stage requires, what goes wrong if you ignore it, and which database types solve it.

Stage 1 - Raw Data Ingestion

What the stage requires: high write throughput, tolerance for unstructured or semi-structured data, and schema flexibility. Training data arrives from many sources; including application events, sensor streams, third-party APIs, and uploaded documents, and the format changes frequently as the team adds new data sources.

What goes wrong: if ingestion runs through a relational database with a strict schema, every new data source requires a migration. The team spends more time managing the database than analysing the data.

The right database type: NoSQL document or column-family stores. MongoDB handles flexible document structures well and works naturally with JSON event data. Apache Cassandra handles extremely high write volumes with no single point of failure, making it the standard choice for IoT and clickstream ingestion pipelines. Amazon DynamoDB fills the same role for serverless architectures on AWS.

Key principle:

Ingestion is not the place for data quality enforcement. Accept the data, store it flexibly, and validate and transform it downstream. Rigid schemas at ingestion create friction that slows the entire pipeline.

Stage 2 - Feature Engineering

What the stage requires: the ability to run complex SQL queries, joins, and aggregations across large datasets. Feature engineering is an analytical workload, you are reading millions or billions of records, computing statistics, and producing a clean, model-ready output.

What goes wrong: if the data lives in a NoSQL store that does not support complex joins, feature engineering has to happen in external scripts. This creates fragile pipelines, version control problems for feature logic, and difficulty reproducing the exact feature set used to train a specific model.

The right database type: SQL databases and cloud data warehouses. For teams at a moderate scale, PostgreSQL handles feature engineering well, and its extension ecosystem (TimescaleDB and PostGIS) covers a wide range of data types. For a larger scale, Google BigQuery and Snowflake run SQL over petabyte-scale datasets without any infrastructure management.

An important point about dbt: data transformation tools like dbt (data build tool) work alongside your SQL warehouse to manage feature engineering logic as versioned, testable code. If your team is running feature engineering at scale, dbt should be part of the stack.

Stage 3 - Feature Serving

What the stage requires: sub-10ms latency for key-value lookups. At inference time, the model needs its input features immediately. A user submits a request, the serving layer retrieves the user's pre-computed features from a store, passes them to the model, and returns a prediction, all within a tight latency budget.

What goes wrong: teams recompute features at inference time using the same code that was used at training time. This is slow (SQL queries at serving time add hundreds of milliseconds), and it introduces training-serving skew when the serving-time computation differs slightly from the training-time computation.

The right database type: Redis is the near-universal answer for real-time feature serving. It is an in-memory key-value store that returns data in under a millisecond. Pre-computed features are written to Redis after each training run and at regular refresh intervals, and the serving layer reads from Redis at inference time. DynamoDB fills the same role for serverless AWS architectures where you need automatic scaling and zero operational overhead.

Stage 4 - Embedding Storage and Retrieval

What the stage requires: the ability to store high-dimensional numerical vectors (embeddings) and find the nearest neighbours of a query vector across millions or billions of stored vectors. This is the core operation in RAG systems, semantic search, and recommendation engines.

What goes wrong: teams try to store embeddings in PostgreSQL as float arrays and compute nearest neighbours with a full table scan. This works for a few thousand vectors but becomes

unusably slow at scale. The mathematical structure of nearest-neighbour search requires specialized index types that standard databases do not have.

The right database type: vector databases. Pinecone is the fully managed option, no infrastructure to operate, but proprietary and expensive at scale. Weaviate and Qdrant are open-source and self-hostable, with strong hybrid search (combining vector similarity with keyword matching).

For teams already on PostgreSQL, the pgvector extension provides vector search without introducing a new system, though it does not match dedicated databases at scale above roughly ten million vectors.

Stage 5 - Batch Training at Scale

What the stage requires: high read throughput on large compressed datasets, columnar storage (so you can efficiently read one column across billions of rows); and, ideally, the ability to run computation close to the data.

What goes wrong: training data lives in an operational database not designed for analytical workloads. Training jobs move enormous amounts of data across networks before processing can begin. The bottleneck is data movement, not compute.

The right database type: cloud data warehouses and distributed processing platforms. BigQuery is serverless and runs SQL-based feature preparation at the petabyte scale. Databricks combines distributed Spark processing with the Delta Lake storage format, giving you both analytics and ML tooling in one platform. Snowflake's compute-storage separation lets you spin up a large cluster for a training run and scale back down when it is done.

Chapter 3: Three Core Architecture Patterns

Most ML systems fall into one of three architectural patterns based on their maturity, team size, and data scale. Each pattern builds on the previous one; you can migrate from Pattern A to B to C as your system grows, without starting over.

These are not rigid prescriptions. They are starting points. Your specific requirements may call for a variation, and the decision framework in Chapter 5 will help you make those calls.

Pattern A; The Lean Stack

Best for: teams of 1 to 5 engineers, early-stage products, validation phases, and under 5 million vectors.

The principle behind Pattern A is using the fewest possible systems while still covering all five pipeline stages. PostgreSQL with the pgvector extension handles both structured training data and vector similarity search. Redis handles real-time feature serving. Nothing else.

This is the stack that gets you from zero to a working production ML system with minimal operational overhead. PostgreSQL is well understood, has a rich ecosystem of tooling, and can be hosted cheaply on any cloud provider. The pgvector extension adds vector search without introducing a new system to monitor, backup, and operate.

The Lean Stack - Components

- PostgreSQL — training data storage, feature tables, experiment metadata, vector embeddings via pgvector
- Redis — pre-computed feature serving at inference time
- Object storage (S3 or GCS) — model artifacts, raw data files

Where it breaks down: pgvector performance degrades above roughly 5 to 10 million vectors. PostgreSQL is not designed for petabyte-scale analytical queries. When training data reaches hundreds of gigabytes and query times start climbing, it is time to move to Pattern B.

Migration signal: When a training data query that took 30 seconds starts taking 8 minutes, or when vector search P99 latency exceeds 100 ms on your production workload, Pattern A has reached its ceiling.

Pattern B - The Growth Stack

Best for: teams of 5 to 20 engineers, scaling products with production traffic, millions of daily events, and production RAG applications.

Pattern B introduces specialization. Each part of the pipeline gets a database designed for its specific access pattern. The Lean Stack's single PostgreSQL instance splits into three systems: a NoSQL store for ingestion, a cloud warehouse for training and feature engineering, and a dedicated vector database for retrieval.

The Growth Stack - Components

- MongoDB or DynamoDB — raw data ingestion, flexible schema for evolving data sources
- BigQuery or Snowflake — feature engineering, SQL, training data preparation, batch analytics
- Weaviate or Qdrant — vector embeddings, semantic search, RAG pipeline
- Redis, a real-time feature serving at inference
- Object storage — raw data lake, model artifacts

At this stage, the data flow looks like this: raw events land in MongoDB or DynamoDB. A scheduled pipeline (Airflow, dbt, or a cloud data pipeline tool) transforms and loads the cleaned data into BigQuery or Snowflake for feature engineering.

The vector database is loaded from processed documents or embeddings generated from the warehouse data. Redis is populated with pre-computed features on a refresh schedule.

Where it breaks down: when vector collections exceed 100 million entries, when training workloads require distributed Spark processing, or when the organization is running dozens of

models that share overlapping feature sets and training-serving skew becomes a recurring problem.

Pattern C - The Enterprise Stack

Best for: teams of 20 or more engineers, enterprise AI platforms, hundreds of millions of vectors, multi-region deployments, organizations running many models.

Pattern C adds two components that Pattern B lacks: a dedicated feature store and a distributed ML platform. The feature store (Feast, Tecton, or Databricks Feature Store) manages the lifecycle of features centrally, ensuring that every model in the organization uses the same, consistently computed features for both training and serving.

The distributed platform (Databricks or a comparable lakehouse) handles the data engineering and model training at scale.

The Enterprise Stack - Components

- Apache Kafka or a cloud streaming service for event ingestion at high volume
- Delta Lake on Databricks or Snowflake, data warehouse, and feature engineering
- Feature store (Feast, Tecton, or Databricks Feature Store): centralised feature management for training and serving consistency
- Milvus or Pinecone — enterprise-scale vector retrieval (hundreds of millions of vectors and above)
- Redis Cluster, a distributed, high-availability feature serving
- Object storage — raw data, model artifacts, training checkpoint

The enterprise stack is designed for organizations where the cost of training-serving skew, inconsistent feature definitions, or outages in the serving layer is significant. It trades operational complexity for reliability, consistency, and scale.

The three patterns form a progression. The table below summarises when to use each one:

Pattern	Team Size	Vector Scale	When to Move On
Lean Stack	1–5 engineers	Up to 5M vectors	Query times are climbing; pgvector latency > 100ms.
Growth Stack	5–20 engineers	5M–100M vectors	Training-serving skew: need distributed Spark

Enterprise Stack	20+ engineers	100M+ vectors	You are here for the long term

Chapter 4: Vector Database Selection Framework

Vector databases have become a non-negotiable infrastructure for AI teams in 2026. Almost every production LLM application uses some form of retrieval, a vector database retrieves the most relevant context before the model generates a response. Choosing the wrong one, or choosing one without a clear understanding of the tradeoffs, leads to performance problems, vendor lock-in, or expensive re-architectures.

The options in this category market themselves aggressively, and all claim to be the best. This chapter gives you six criteria that reveal the real differences in production, followed by three questions that lead directly to the right choice for your situation.

Six Criteria That Actually Matter in Production

1. Query Latency at Your Scale

Marketing benchmarks always show impressive numbers on a small scale. The question that matters is, 'What is the P99 latency (the slowest 1% of queries) at your specific vector count and query volume?' This number often looks very different from the P50 average. For user-facing AI applications, P99 matters more than average.

2. Metadata Filtering Quality

Pure vector similarity search is rarely enough in production. You almost always need to combine a similarity search with a filter to find documents similar to this query that also belong to this customer, are from the last 30 days, and are of this document type. The quality and speed of metadata filtering vary enormously between vector databases, and it degrades in different ways at scale.

3. Hybrid Search Support

Hybrid search combines vector similarity with traditional keyword (BM25) ranking. It consistently produces better results than pure vector search for queries that contain specific terms, product IDs, or technical keywords where exact matching matters. Not all vector databases support hybrid search natively. For enterprise applications where retrieval precision is critical, this should be a hard requirement.

4. Operational Burden

Self-hosted vector databases (Qdrant, Milvus, and Weaviate self-hosted) require Kubernetes expertise, index configuration, backup management, and monitoring. Managed services (Pinecone, Weaviate Cloud, Qdrant Cloud) take care of all of this at a cost premium. Be honest about your team's operational capacity. A managed service that costs 30% more may be the right call if it saves your team 10 hours per week of infrastructure work.

5. Cost at Your Scale

Vector database pricing models differ dramatically, and the cost curves cross at different scale points. Managed services charge per vector stored and per query. Self-hosted systems charge for compute and storage but require operational labour. For most teams, managed services are cheaper up to roughly 10 to 20 million vectors with moderate query volumes. Above that, self-hosted typically becomes significantly more cost-efficient.

6. Data Residency Requirements

Many enterprise and regulated deployments require that data never leave a specific cloud region or provider. Fully managed cloud services (Pinecone) may not offer the data residency options your compliance team requires. Self-hosted or private cloud deployments (Qdrant, Milvus) give you full control over where data lives.

Three Questions to Find the Right Choice

Answer these three questions, and the right vector database becomes clear for most teams:

Question 1: How many vectors do you need to store now and in 12 months?

- Under 5 million pgvector on your existing PostgreSQL instance is probably sufficient. Do not add a new system before you need to.
- 5 million to 100 million Weaviate or Qdrant, either managed or self-hosted. Both perform well in this range and offer hybrid search.
- Above 100 million Milvus for maximum performance flexibility, or Pinecone if you need zero-ops managed scale.

Question 2: Do you need a hybrid search?

- Yes, Weaviate (strongest native hybrid search) or Qdrant. Both combine a vector and BM25 in a single query.
- No, any of the above options.

Question 3: Can your team operate Kubernetes in production?

- Yes, self-hosted Qdrant or Milvus gives you the best cost-performance ratio at scale.
- No, use a managed service. Pinecone is the simplest. Weaviate Cloud and Qdrant Cloud offer more flexibility.

A Note on RAG-Specific Considerations

Vector database choice interacts with two other RAG architecture decisions that teams often overlook.

Chunk size affects index size and query latency. Larger chunks mean fewer vectors and faster index loads, but lower retrieval precision. Smaller chunks give higher precision but more vectors and longer query times. The right chunk size depends on your document types and query patterns, and it affects the scale tier your vector database needs to handle.

Namespace and tenant isolation matter for multi-tenant AI products where each customer has their own isolated knowledge base. Pinecone handles this well with named namespaces. Weaviate uses named classes. Milvus uses partitions. Each has different isolation guarantees and performance characteristics at high tenant counts.

Chapter 5: In-Database ML - When to Skip the External Pipeline

Most ML pipelines follow the same pattern: extract data from a database, load it into a Python environment, train a model using scikit-learn or PyTorch or XGBoost, then load the results back. This works, but it creates a pipeline with multiple moving parts, a data movement bottleneck, and a class of infrastructure engineering problems that exist solely to move data between systems.

In-database machine learning takes a different approach. You write a SQL statement that creates and trains a model directly against the data already in the warehouse. No extraction. No movement. No separate training environment. The data stays where it lives, and the computation happens there too.

When In-Database ML Is the Right Call

In-database ML is the right choice when three conditions are true:

- Your training data is already in a supported warehouse (BigQuery, Redshift, Snowflake, or Oracle). If the data is already there, in-database ML eliminates the most expensive step of a conventional pipeline.
- Your use case fits a standard model type. In-database ML platforms support logistic regression, linear regression, decision trees, XGBoost, K-means clustering, matrix factorization for recommendations, and time-series forecasting. If you need a custom transformer architecture, you still need an external training environment.
- Data governance or compliance requirements prevent data movement. Financial services and healthcare organizations often cannot allow training data to leave a controlled environment. In-database ML keeps data within the warehouse's security boundary throughout the entire training process.

BigQuery ML - The Most Complete Implementation

BigQuery ML supports the broadest range of model types and provides the most complete SQL-based ML lifecycle of any in-database platform. You can explore data, engineer features, train models, evaluate them, generate predictions, and create explainability reports — all in SQL, all without leaving the BigQuery environment.

The supported model types include logistic regression (binary and multi-class), linear regression, K-means clustering, matrix factorization (product recommendations), ARIMA-Plus for time-series forecasting, XGBoost, TensorFlow-based deep neural networks, and AutoML Tables for automated model selection.

The workflow is worth understanding in detail because it is significantly simpler than a conventional training pipeline. You create a model with a SELECT statement as the training data, evaluate it against a held-out dataset, and run predictions with a function call. The entire cycle takes four SQL statements.

Other In-Database ML Platforms

Amazon Redshift ML

Redshift ML connects to SageMaker Autopilot. You write a CREATE MODEL statement specifying the training table and the target column. Redshift exports the data to S3, SageMaker runs automated model selection and hyperparameter tuning, and the winning model is registered as a prediction function in your Redshift cluster. You call it with a SELECT statement.

This is particularly useful for AWS-centric teams that want AutoML without managing ML infrastructure. The tradeoff is that the training step depends on SageMaker and S3, which adds some complexity compared to BigQuery ML's fully unified approach.

MySQL HeatWave AutoML

HeatWave brings AutoML capabilities directly into MySQL with Oracle Cloud acceleration. For teams whose application already runs on MySQL, this is a compelling path to machine learning because the training data is already in the operational database. No warehouse, no ETL, no migration, you run AutoML against the tables your application writes to every day.

Snowflake Snowpark ML

Snowflake takes a different approach to in-database ML. Rather than a SQL interface, Snowpark ML brings Python directly into the Snowflake compute layer. Data scientists write Python DataFrames and scikit-learn-compatible pipelines that execute inside Snowflake against data that never moves.

For teams that are comfortable in Python and already use Snowflake as their warehouse, this combines the flexibility of Python ML with the data governance advantages of in-database processing.

When to Skip In-Database ML

In-database ML is not the right tool for every situation. Skip it when:

- You need custom deep learning architectures, transformers, diffusion models, and multi-modal networks. Warehouse ML platforms support standard algorithms but not arbitrary model architectures.

- You need fine-grained control over training hardware, specific GPU types, distributed multi-node training, and custom CUDA kernels. In-database platforms run on the warehouse's managed compute and do not expose hardware controls.
- Your team already has a mature Python ML pipeline with robust tooling, versioning, and experiment tracking. Replacing a working pipeline with in-database ML disrupts it without adding meaningful value.
- You need a framework-specific library not available in the platform, specialized NLP libraries, computer vision frameworks, and custom loss functions.

Chapter 6: Common Database Challenges in Production AI

The database decisions you make at the start of an ML project shape the system for months or years. But the harder problems typically emerge after launch. Training data drifts. Embeddings go stale. Indexes grow slow. Schemas evolve. Compliance requirements tighten.

This chapter covers the six most common production database challenges in AI systems and what to do about each one.

1. Data Drift

Data drift happens when the real-world data your model sees in production begins to look different from the data it was trained on. User behaviour changes. New products are launched. Market conditions shift. The features that were predictive six months ago may no longer carry the same signal.

The database layer plays a specific role in drift management: it is where the historical signal lives. An effective drift response requires storing feature distribution statistics over time (in a warehouse or time-series database), monitoring for statistical divergence between recent data and training data, and triggering retraining when the divergence exceeds a threshold.

The practical implication: design your feature storage with drift monitoring in mind from the start. Store not just feature values but feature statistics, mean, standard deviation, and distribution histograms; at regular intervals. This historical record is what makes drift detectable before it causes meaningful model degradation.

2. Embedding Refresh

In RAG and semantic search systems, embeddings are the bridge between your data and the model's ability to retrieve it. When the source data changes, new documents are added, existing documents are updated, and old documents are removed; the embeddings in your vector database become stale. The retrieval system returns outdated or irrelevant results. The LLM generates responses based on information that no longer reflects the actual knowledge base.

The right approach is an incremental embedding refresh pipeline: detect content changes (via timestamps, change data capture, or explicit versioning), generate new embeddings only for

changed documents, and update the vector index without a full rebuild. A full rebuild of a large vector index can take hours and creates a window where the retrieval system is unavailable or degraded.

For large-scale systems, consider keeping a separate staging index where new embeddings are loaded and tested before being promoted to production. This lets you validate retrieval quality before the change goes live.

3. Index Performance Degradation

Vector database indexes, particularly HNSW (Hierarchical Navigable Small World) indexes, can degrade in performance over time as data is added, deleted, and updated. Deleted vectors leave holes in the index graph that increase search path lengths. High insertion rates can produce suboptimal graph structures. The result is higher query latency and, in some cases, lower recall accuracy.

The operational response is periodic index rebuilding or optimisation during low-traffic windows. Most production vector databases support background optimisation that compacts the index without taking it offline. Schedule these operations, monitor index health metrics (graph connectivity, entry point quality), and validate recall accuracy after rebuilds to confirm that retrieval quality has not degraded.

SQL databases and warehouses have analogous maintenance requirements — table vacuuming in PostgreSQL, partition pruning in BigQuery, clustering key maintenance in Snowflake. These should be part of standard database maintenance schedules, not reactive fire-fighting.

4. Schema Evolution

ML projects rarely operate with stable schemas. New data sources require new columns. Model iterations change which features are relevant. Business requirements add new entities and relationships. The question is not whether schemas will change but how gracefully the system handles those changes.

The approaches that work in practice: use additive-only schema changes where possible (add columns, do not remove or rename them) to preserve backward compatibility. Version your feature definitions explicitly – a feature named 'user_activity_7d_v2' is clearer and safer than one that silently changes its computation. Use a feature store (Feast, Tecton, or Databricks Feature Store) to abstract feature definitions away from model code so that changing how a feature is computed does not require changes to every model that uses it.

5. Training-Serving Skew

Training-serving skew is the condition where the features a model received during training are computed differently from the features it receives during inference. The model's predictions degrade not because of data drift but because the input it is evaluated on at serving time is subtly different from what it learned on. This is one of the most common and most frustrating

causes of production model underperformance.

The root cause is almost always organizational: different code computes features at training time than at serving time, and no mechanism enforces consistency. The structural solution is a feature store with both offline (batch training) and online (real-time serving) backends. The same feature definition produces the training data and the serving data, guaranteed by the feature store, not by convention or code review.

If a feature store is not an option, a practical alternative is to serialize the exact feature computation logic (not just the feature values) at training time and use that same serialized logic in the serving pipeline. Any divergence between training and serving should fail loudly, not silently produce wrong predictions.

6. Security and Governance

AI systems process sensitive data. Training datasets may contain personally identifiable information, financial records, or healthcare data. Vector databases store document embeddings that can sometimes be used to reconstruct source text. Model outputs may reveal information about individual training examples.

The database layer is a significant part of the security perimeter. Implement role-based access controls so that only the systems and users that need to read training data can do so. Encrypt data at rest and in transit. Maintain audit logs of data access for regulated environments. For vector databases, be aware that embeddings are not inherently anonymous — reconstruction attacks are possible on some embedding models.

For teams working with healthcare (HIPAA), financial (SOX, PCI-DSS), or European (GDPR) data, the in-database ML platforms covered in Chapter 5 deserve serious consideration precisely because they minimize data movement and keep sensitive data within a controlled, auditable environment.

Chapter 7: Building Your Database Roadmap

The goal of this playbook is not to get you to the enterprise stack as fast as possible. It is to get you to the right stack for your current stage, and to do it in a way that allows you to grow without starting over.

This chapter gives you a three-phase migration path and a 12-question checklist to validate your architecture before you build it.

The Three-Phase Roadmap

Phase 1 - Validate (Pattern A, Lean Stack)

The goal in Phase 1 is to prove the product works, not to build scalable infrastructure. Use the fewest systems possible. Accept that you will migrate later. PostgreSQL with pgvector is a

perfectly reasonable starting point for a production RAG application with hundreds of thousands of vectors. Redis for feature serving. Object storage for raw data.

What to do in Phase 1: establish clear scale thresholds that will trigger a move to Phase 2. Write these down. A common set: move when the vector collection exceeds 5 million, when training data queries exceed 5 minutes, or when inference latency consistently exceeds 100ms. When you hit these thresholds, it is time to specialize.

Phase 2 - Scale (Pattern B, Growth Stack)

In Phase 2, you are separating the concerns that were consolidated in Phase 1. Ingestion moves to a NoSQL store. Training and feature engineering move to a cloud warehouse. Vector retrieval moves to a dedicated vector database. Redis stays.

The migration priority: start with the component that is causing the most pain. If training is the bottleneck, migrate to BigQuery or Snowflake first. If vector search latency is the issue, migrate to Weaviate or Qdrant first. You do not have to migrate all five stages at once.

What to do in Phase 2: implement the feature store pattern even if you are not using a dedicated feature store product yet. At a minimum, create a clear boundary between the code that computes features and the code that serves them. This boundary makes the Phase 3 migration to a formal feature store significantly easier.

Phase 3 - Operate at Scale (Pattern C, Enterprise Stack)

Phase 3 is for organizations that are running multiple models, serving significant traffic, and where inconsistencies in the data layer are creating measurable problems. The additions in Phase 3, a feature store, a distributed ML platform, a high-availability serving layer, are organizational as much as technical. They require buy-in from data engineering, ML, and platform teams.

Do not rush to Phase 3. The overhead of operating a Databricks cluster, a Feast feature store, and a Milvus cluster is real. Teams that adopt the enterprise stack before they need it spend engineering time managing infrastructure instead of building product.

Architecture Validation Checklist

Before finalizing your database architecture, work through these 12 questions. Each one surfaces a common design gap.

1. Have you identified the access pattern for each of the five pipeline stages: ingestion, feature engineering, feature serving, embedding retrieval, and batch training?
2. Does your feature serving database return features in under 10ms at your expected peak query rate?
3. If you are using a vector database, have you estimated your vector count at 12 months, not just at launch?
4. Do you have a documented plan for refreshing embeddings when source documents change?

5. Is feature computation logic shared between training and serving, or could training-serving skew emerge?
6. If you are using a managed vector database, have you calculated the cost at your 12-month projected vector count and query volume?
7. Does your chosen vector database support hybrid search if your use case requires matching specific terms?
8. Have you identified the compliance requirements for your training data, and does your architecture keep sensitive data within the required security boundary?
9. Do you have a plan for re-indexing your vector database during low-traffic windows as the collection grows?
10. Are your schema change processes additive-only, or could a breaking schema change disable a model in production?
11. Have you defined the scale thresholds that will trigger a migration from your current architecture pattern to the next?
12. Is your feature store or feature serving layer monitored for data drift, or would drift go undetected until model accuracy degrades visibly?

Re-evaluation cadence:

Database requirements change as your product scales. Set a calendar reminder every six months to re-run this checklist against your current architecture. The right answer at 1 million vectors is often the wrong answer at 50 million.

Conclusion

The data layer is not a detail of ML system architecture. It is the foundation. The decisions made here – which database serves ingestion, which handles feature engineering, which stores embeddings, and which serves features at inference- shape the performance ceiling, operational complexity, and long-term scalability of every model that runs on top of them.

The core principle of this playbook is that ML systems have five distinct data access patterns, and no single database handles all five well. The teams that build the most reliable, scalable AI systems are the ones that match database technology to access patterns rather than reaching for the familiar tool regardless of fit.

The three architecture patterns in Chapter 3 give you a starting point matched to your current scale: the Lean Stack for validation and early-stage products, the Growth Stack for scaling applications, and the Enterprise Stack for large organizations with complex multi-model environments. Each is designed to be migrated to the next when scale demands it.

The most important single piece of advice this playbook can offer is to set your migration thresholds before you need them. Decide in advance: At what vector count do we add a dedicated vector database? At what training data volume do we move to a cloud warehouse? At

what inference latency do we add a feature store? Teams that define these thresholds ahead of time migrate smoothly. Teams that wait for a crisis re-architect under pressure.

Work with Code-B's ML engineering team.

Code-B's engineering team designs and builds production AI systems for companies at every stage, from choosing the right database stack to architecting and implementing complete ML pipelines.

If you are designing a new ML data layer or auditing an existing one, we offer a free 30-minute architecture review. We will look at your current or planned stack, flag the gaps, and give you specific, actionable recommendations – no sales pitch, just engineering.

Book your free architecture review at code-b.dev/contact-us