Building Intelligent Applications

PostgreSQL for AI

Vector Search, RAG, Feature Stores, In-Database ML & Real-Time Pipelines

Table of contents

Preface

PostgreSQL has evolved far beyond a traditional relational database. With extensions like pgvector, pgml, and pg_cron, it now serves as a unified platform for AI-powered applications — handling vector search, machine learning inference, RAG pipelines, and real-time AI features alongside your transactional data.

This book shows you how to build intelligent applications using PostgreSQL as the foundation. Through a running example — RecSys, an AI-powered product recommendation platform — you'll learn to implement vector similarity search, integrate large language models, build feature stores, deploy in-database ML models, and architect production-grade AI systems.

Who This Book Is For

- **Backend developers** looking to add AI capabilities to PostgreSQL-backed applications
- **Data engineers** building feature pipelines and ML infrastructure
- **Full-stack developers** who want to understand AI-native database architectures
- **DevOps/SRE teams** deploying and operating AI-enhanced PostgreSQL systems

The Running Example

Throughout this book, we build **RecSys** — an AI-powered product recommendation platform. Each chapter adds capabilities to this system:

- **Vector search** for semantic content discovery
- **RAG pipelines** for AI-generated summaries and answers
- **Feature engineering** for personalized recommendations
- **In-database ML** for real-time scoring
- **Production infrastructure** for reliability at scale

All code runs on a single PostgreSQL instance with Docker Compose, so you can follow along on your laptop.

How to Read This Book

The book is organized in four parts:

Part I: Foundations introduces why PostgreSQL is uniquely suited for AI workloads and covers the modern PostgreSQL features that make it possible.

Part II: Core AI Capabilities dives into the technical building blocks — vector search, LLM integration, feature engineering, and in-database machine learning.

Part III: Production AI Systems covers real-time pipelines, architecture patterns, and performance optimization for AI workloads.

Part IV: Operations & Beyond addresses security, production deployment, and the future of PostgreSQL in the AI ecosystem.

The chapters are presented sequentially and each builds on previous ones, but experienced readers can selectively navigate Parts II–IV based on their needs — provided they have the foundational knowledge from Part I.

Dedication

To everyone who believes that the database can be more than just storage — that it can think, learn, and reason.

And to the PostgreSQL community, whose relentless innovation makes that belief a reality.

About the Author

Ahmet Zeybek has spent the better part of +15 years building things that talk to databases and the better part of 10 of those years arguing that PostgreSQL can handle more than people give it credit for. He studied Computer Engineering, then went on to design and operate over twenty production PostgreSQL deployments across SaaS platforms, e-commerce systems, and online games. Somewhere along the way, he started combining databases with machine learning and never quite stopped.

This book grew out of a recurring frustration: every AI tutorial assumed you'd export your data to somewhere else before doing anything interesting with it. Ahmet wanted to show that PostgreSQL, the database already running in most companies, could handle embeddings, vector search, RAG pipelines, and ML inference without bolting on an entirely separate infrastructure. The result is the book you're holding.

Ahmet contributes to open-source projects, speaks at developer meetups, and lives in Turkey. You can find him at zeybek.dev or on GitHub.

Acknowledgments

This book exists because PostgreSQL's extension ecosystem made something remarkable possible: running AI workloads — embeddings, vector search, machine learning, real-time inference — inside the database itself. That ecosystem is the work of thousands of contributors across decades, and I owe them a tremendous debt.

Open Source Projects and Their Creators

The technical foundation of every chapter rests on open-source projects whose maintainers build in public, answer strangers' questions, and fix bugs at midnight. I want to acknowledge them directly, because without their work this book would have nothing to write about.

Andrew Kane created pgvector and almost single-handedly brought native vector search to PostgreSQL. What started as a modest extension in 2021 has become the backbone of an entire ecosystem. Every chapter in this book that touches embeddings — and that's most of them — depends on his work and his relentless commitment to keeping pgvector simple, correct, and fast.

The Timescale team deserves special recognition. **Matvey Arye** and the engineering team behind pgai and the Vectorizer framework showed that SQL-native AI workflows are not a compromise but a genuine architectural advantage. **Avthar Sewrathan** and the developer relations team produced documentation and examples that helped me understand design decisions I would have otherwise gotten wrong. TimescaleDB's continuous aggregates and hypertables power several of this book's feature engineering examples, and their willingness to discuss internals shaped how I present real-time AI pipelines.

The PostgresML team, led by **Montana Low** and **Lev Kokotov**, proved that in-database machine learning is not just feasible but production-ready. Their work on bringing XGBoost, LightGBM, and transformer models inside PostgreSQL gave Chapter 7 its reason to exist. The PostgresML documentation is among the best I've encountered in any open-source project — clear, honest about limitations, and full of working examples.

The **Ollama project** gave this book its "runs on your laptop" philosophy. By making local LLM inference as easy as `ollama run`, they removed the single biggest barrier to hands-on learning: the need for API keys, cloud accounts, and credit cards. Every code example in this book works offline, and that's largely thanks to Ollama.

Nomic AI deserves mention for `nomic-embed-text`, the embedding model used throughout this book. An open-source, 768-dimensional model with an Apache 2.0 license — it proved that high-quality embeddings don't require proprietary APIs.

To the **PostgreSQL Global Development Group**: you've maintained and evolved a database for over 35 years with extraordinary discipline, backward compatibility, and engineering taste. The extension system, MVCC, logical replication, foreign data wrappers, full-text search — these aren't just features, they're the reason PostgreSQL can serve as a credible AI platform without being rewritten from scratch. This book is, at its core, a love letter to extensibility — and that extensibility is your gift to all of us.

I also want to recognize the teams behind **Debezium**, **Apache Kafka**, **pg_cron**, **pgcrypto**, **pgaudit**, **Patroni**, **Langfuse**, and **dbt** — tools that appear throughout these pages and that make production AI systems possible.

Technical Reviewers

The technical reviewers who read early drafts of this book caught errors I'd missed, challenged assumptions I'd taken for granted, and pushed me to address edge cases I had conveniently ignored. Their feedback on everything from HNSW index parameters to differential privacy implementations made every chapter significantly better. Any remaining errors are entirely my own.

Community and Inspiration

Several people shaped my thinking without necessarily knowing it.

Martin Kleppmann's *Designing Data-Intensive Applications* set the standard for how technical books should discuss trade-offs — honestly, with nuance, and with respect for the reader's intelligence. His influence is visible in every "When NOT to Use This" section in this book. **Simon Willison**'s relentless exploration of AI tooling, his writing about LLMs, and his "build it and show your work" ethos inspired the hands-on approach I've tried to maintain throughout.

The PostgreSQL community on Hacker News, the **pgsql-hackers** mailing list, and the **pgvector GitHub discussions** provided countless insights that found their way into these pages. Many of the "gotchas" and real-world patterns in this book came not from papers or documentation, but from practitioners sharing hard-won lessons in public forums.

The academic papers cited throughout this book represent years of foundational work. I am particularly grateful to the researchers behind HNSW, RAG, Matryoshka embeddings, differential privacy, and the OWASP LLM Top 10 — their work makes the practical applications in this book possible.

Personal

Writing a technical book is an exercise in sustained obsession. For over a year, evenings and weekends disappeared into terminal windows, Docker containers, and endless rounds of "just one more revision."

To my friends and colleagues who listened to me talk about embeddings at dinner parties, offered encouragement when progress felt slow, and asked "how's the book going?" without flinching at the answer: I owe you more than a mention on this page.

To Gamze: you endured a year of late nights, distracted weekends, and conversations that inevitably circled back to PostgreSQL. You never once asked me to stop. Instead, you brought tea, cleared space, and quietly made sure everything else in our life kept running while I disappeared into this manuscript. This book is yours as much as it is mine.

Writing is rewriting. This book went through more drafts than I care to admit. If you find an error, an unclear explanation, or a code example that doesn't work, I'd genuinely appreciate hearing about it — it means someone is actually reading, and that's the best outcome an author can hope for.

SQL Refresher

This book assumes you're comfortable with SQL fundamentals: `SELECT`, `JOIN`, `WHERE`, and `GROUP BY`. If your SQL skills have gotten rusty, or you've never worked with PostgreSQL specifically, the resources below will get you up to speed quickly. None of them cost money, and most are interactive so you can learn by doing.

Resource	Focus	Best For
PostgreSQL Official Tutorial	Comprehensive coverage from basics to advanced	Complete learners and reference lookups
Mode SQL Tutorial	Interactive walkthroughs, especially JOINs and aggregation	Refreshing rusty skills with immediate feedback
SQLBolt	Hands-on exercises with a visual approach	Learning by doing, step-by-step practice
pgexercises	PostgreSQL-specific problems and drills	PostgreSQL syntax and idioms

Choose the one that matches your learning style. If you're completely new to SQL, start with SQLBolt. If you know SQL but want to deepen your PostgreSQL knowledge, pgexercises is your best bet. The PostgreSQL official tutorial is always worth keeping open in a browser tab — it's detailed, honest about limitations, and you'll reference it repeatedly as you work through this book.

Worth noting: Chapter 2 (Chapter 2) covers window functions, CTEs, and other modern PostgreSQL features from scratch. No prior experience with these advanced features is needed — I'll introduce them when they matter.

Reading Guide

The diagram below shows how chapters build on each other. The book is organized into four parts, and while reading cover-to-cover works well, you don't have to.

Start here: Chapters 1 and 2 are recommended for all readers. Chapter 1 frames the problem and introduces the RecSys running example. Chapter 2 establishes the modern PostgreSQL features — JSONB, full-text search, window functions, CTEs — that every later chapter assumes.

Then follow your path. After the foundations, your route depends on your role. Data scientists and ML engineers should follow Part II straight through to master vector search, RAG, feature engineering, and in-database ML. Full-stack developers building AI features will benefit from combining Part II with Part III's real-time pipelines and architecture patterns. Platform and DevOps engineers can jump to Part III and Part IV; they cover architecture, performance, security, and deployment.

All paths converge at Chapter 9 (Architecture) — the point where every piece assembles into a production-ready system.

> Non-Linear Reading
>
> Each chapter opens with a schema checkpoint that loads all prerequisite state. If you want to jump directly to Chapter 8 (Real-Time AI), the checkpoint script will set up everything Chapters 2–7 would have built. You won't have the context of why each piece exists, but the code will run.

Chapter 2 feeds into Chapters 3, 6, and 7 — the three pillars of core AI capability. Chapter 3 (Vector Search) is itself foundational for Chapters 4, 8, and 10. Chapters 4 and 5 form a RAG learning sequence. Chapters 6 and 7 (Feature Engineering and In-Database ML) work as a pair. Chapter 8 (Real-Time AI) pulls from vectors and features. The final sequence — Chapters 10 through 13 — covers performance, security, deployment, and future directions.

Whether you read sequentially or jump to what matters most, the dependency graph above will keep you oriented.

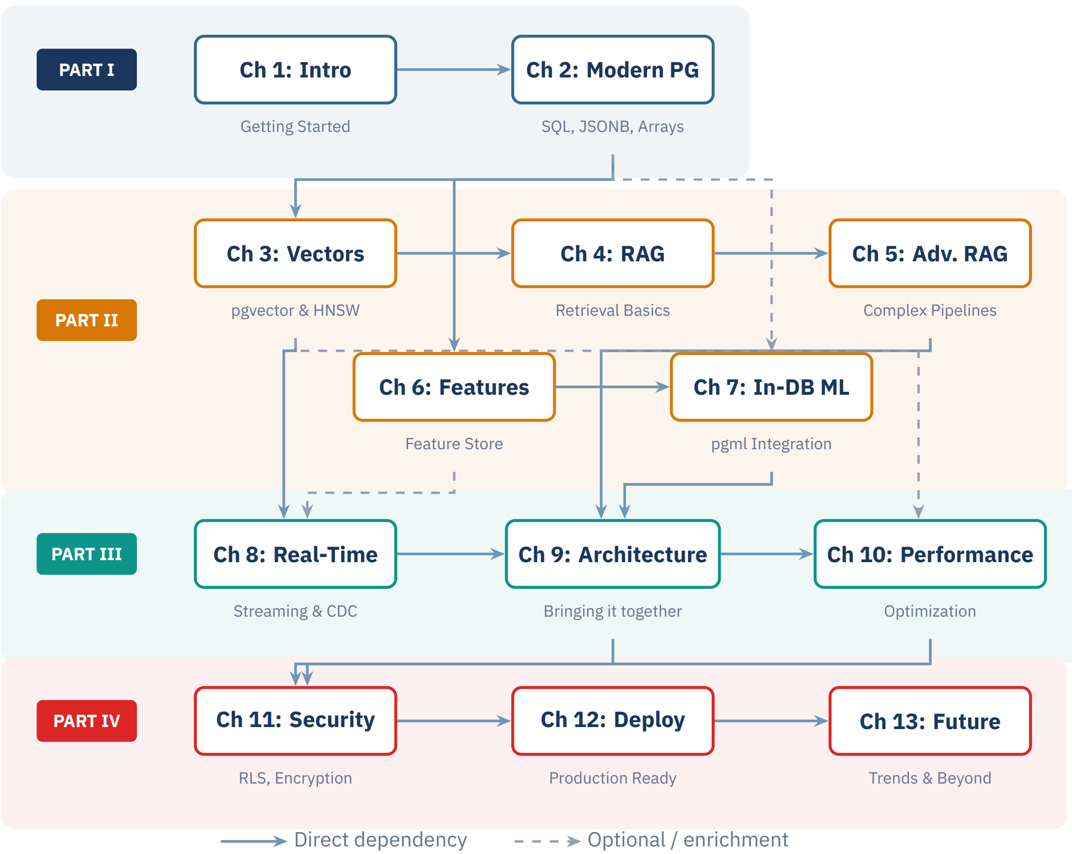

Figure 1.: Chapter dependency graph organized by book parts. Solid arrows show direct dependencies; dashed arrows show optional enrichment paths.

PART

I

Foundations

Every AI system needs a solid foundation. These two chapters establish yours: why PostgreSQL is a serious contender for AI workloads, what it offers out of the box, and the schema patterns that make everything else in this book possible.

The best tool is usually the one you already have.

— A pragmatic engineer’s proverb

1 Introduction: Why PostgreSQL for AI?

Your database already handles your most critical data. This chapter makes the case for why it should handle your AI workloads too, and when it shouldn't.

> RecSys Progress
>
> In this chapter, RecSys is born: you set up the Docker environment, seed the database with 1,000 products, and run your first AI-powered semantic search query.

1.1. Learning Objectives

By the end of this chapter, you will:

- Run your first AI-powered query against PostgreSQL in under five minutes
- Understand PostgreSQL's genuine strengths and limitations as an AI platform
- Identify which of three reader personas matches your background and choose your path through the book
- Know when PostgreSQL is NOT the right choice for your AI workload

> Schema Checkpoint
>
> This is the starting chapter: no prior database setup required. By the end, you'll have a running PostgreSQL instance with pgvector, TimescaleDB, and 1,000 seed products ready for the chapters ahead.
> To start fresh: `docker compose up -d`

1.2. Prerequisites

- Docker and Docker Compose installed on your machine
- Basic familiarity with SQL (SELECT, INSERT, CREATE TABLE)
- A terminal or command-line interface
- Approximately 4 GB of disk space for containers and sample data

1.3. The Problem You Already Have

Your project has pgvector in one container, Redis for caching in another, a Python embedding service in a third, and Pinecone for "real" vector search because someone said pgvector doesn't scale. That's four moving pieces for what should be one query.

And it's not just the architecture diagram that's getting complicated. Every service needs its own monitoring, its own backup strategy, its own security model, its own on-call runbook.

When the embedding service goes down at 2 AM, you're debugging network hops between containers instead of looking at query plans. When your vector database vendor changes their pricing tier, you're rewriting integration code instead of shipping features. You've been here before. You know how this story ends: with a Terraform module nobody wants to touch and a Slack channel called `#infra-fires`.

Here's what makes this particular version of infrastructure sprawl so frustrating: PostgreSQL already handles your relational data. It already handles your full-text search. If you're using TimescaleDB, it handles your time-series data. If you're using pg_cron, it handles your job scheduling. PostgreSQL has been quietly accumulating capabilities for decades, and in the last three years, it's accumulated the ones that matter for AI.

pgvector gives you vector similarity search [1]. pgai gives you in-database embedding generation and LLM (Large Language Model) calls [2]. pgvectorscale gives you DiskANN indexing for large-scale workloads [3]. PostgresML gives you in-database machine learning training and inference [4]. These are production extensions used by companies processing millions of queries daily [5].

So the question isn't "Can PostgreSQL do AI?" It can. The question is: "Should *your* project consolidate its AI workloads into PostgreSQL, and if so, how far should you go?"

If that sounds too good to be true, good; skepticism is the right starting point. This book won't tell you PostgreSQL is the best choice for every AI workload. It isn't.

What this book *will* do is give you practical, verified patterns for the workloads where PostgreSQL genuinely excels, honest benchmarks for where it falls short, and clear decision frameworks for knowing which situation you're in. Every code example runs against a real database. Every performance claim cites a source. Every trade-off gets its own section.

Let's start with where PostgreSQL actually fits in the AI landscape.

1.4. PostgreSQL in the AI Landscape

PostgreSQL occupies a unique position in the AI ecosystem [6], [7]. It's not the fastest vector database. It's not the most sophisticated ML platform. But it's the only system that can handle relational data, vector search, full-text search, time-series analytics, and machine learning inference in a single process, with one backup strategy, one security model, one monitoring stack, and one team that already knows how to run it.

That's not a marketing claim. That's an architectural reality with measurable trade-offs.

To understand PostgreSQL's position, it helps to look at the landscape it sits in. On one side, you have purpose-built vector databases (Pinecone, Weaviate, Milvus, Qdrant, Chroma), each optimized for similarity search as their primary use case. On the other side, you have ML platforms (SageMaker, Vertex AI, Databricks) designed for the full model lifecycle. PostgreSQL sits in the middle, doing none of these things as its *primary* purpose, but doing all of them *well enough* that for most teams, it replaces the need for dedicated tools.

1.4.1. Where PostgreSQL Genuinely Excels

Data proximity. Your features, your embeddings, and your application data live in the same database. A recommendation query that needs user history, product metadata, and vector similarity doesn't cross a network boundary — it's a single SQL statement with a JOIN.

When you fetch recommendations from a separate vector database, you need to do this:

1. Query the vector DB for similar item IDs
2. Send those IDs back to your application
3. Query PostgreSQL for the item metadata
4. Join the results in application code
5. Handle the inevitable consistency issues when items exist in one system but not the other

With pgvector, that's one query; the similarity score and the product details come back in the same result set, with the same transactional guarantees you already rely on. In Chapter 9, you'll see how this eliminates an entire class of consistency bugs that plague multi-system architectures.

Operational simplicity. One system to back up, monitor, secure, and upgrade. If your team already runs PostgreSQL (and statistically, they do), adding AI capabilities means learning new extensions, not new infrastructure. Your existing connection pooling, replication, and disaster recovery all carry over.

This is the argument that doesn't show up in benchmark charts but dominates real-world total cost of ownership. Running Pinecone alongside PostgreSQL means two billing systems, two uptime SLAs, two security audits, and two sets of credentials to rotate; for a startup with three backend engineers, that overhead can easily consume more engineering time than any performance difference saves.

SQL as universal interface. Every developer on your team already speaks SQL. Vector search, full-text ranking, ML inference, and feature engineering are all SQL queries. No new SDKs, no new query languages, no new deployment pipelines.

You'll see this throughout the book: the most useful patterns are often surprisingly short SQL statements. A hybrid search combining vector similarity with full-text relevance and business rule filtering is a single query with a CTE; a feature engineering pipeline that would require pandas, Redis, and a scheduler in Python is a materialized view with pg_cron.

Extension ecosystem. The PostgreSQL AI stack isn't a single monolithic product; it's a composable set of extensions, each doing one thing well:

- **pgvector** — Vector similarity search with HNSW [8], [9] and IVFFlat indexes [1]
- **pgai** — In-database embedding generation and LLM function calls [2]
- **pgvectorscale** — DiskANN indexing for larger-than-memory vector workloads [3]
- **PostgresML** — In-database model training and inference [4]
- **TimescaleDB** — Time-series data management and continuous aggregates
- **pg_cron** — Scheduled batch jobs without external schedulers

These extensions compose naturally. You can use pgvector for search, pgai for embeddings, TimescaleDB for time-series features, and pg_cron for scheduled refreshes — all in the same database, all accessible via SQL. Chapter 12 provides the full compatibility matrix showing which extensions work together and which have conflicts.

1.4.2. Where PostgreSQL Genuinely Struggles

Honest positioning means acknowledging the limitations as clearly as the strengths. PostgreSQL has real constraints for AI workloads, and pretending otherwise would waste your time.

Raw vector throughput at extreme scale. Purpose-built vector databases like Pinecone, Weaviate, Milvus, and Qdrant are engineered for one thing: serving vector queries as fast as possible. At scales beyond pgvector's comfortable range (see Section 3.19 for specific thresholds), they achieve lower latencies through GPU acceleration, distributed sharding, and indexing algorithms optimized exclusively for vectors.

pgvector and pgvectorscale are excellent up to that scale (and improving rapidly), but they carry the overhead of being general-purpose database extensions. When your vector index needs more RAM than your largest available instance, purpose-built databases that shard transparently across nodes have a genuine architectural advantage.

Chapter 3 covers the specific benchmarks with real numbers and methodology. Chapter 10 details tuning strategies — partitioning, quantization, read replicas — that push PostgreSQL's boundary further than you might expect.

GPU acceleration. PostgreSQL runs on CPUs (full stop). For workloads that benefit from GPU-accelerated inference or training (large language model fine-tuning, real-time image generation, continuous model retraining on millions of samples), you need external compute. There is no pgvector GPU mode, and there is no way to attach a GPU to a PostgreSQL query executor.

This isn't a temporary limitation waiting for a patch; it's a fundamental architectural constraint. PostgreSQL's shared-memory, process-per-connection model wasn't designed for GPU workloads, and retrofitting GPU support would require changes to the core query executor that aren't on any roadmap.

Chapter 7 shows how far in-database ML goes with CPU-based training and inference, and where it stops. For many practical workloads (classification, regression, anomaly detection on tabular data), CPU is more than sufficient. For anything involving deep learning at scale, you'll need infrastructure outside PostgreSQL.

Cutting-edge ML serving. If you need A/B tested model deployments with traffic splitting, automatic rollback on metric degradation, canary releases with statistical significance testing, and GPU-backed inference endpoints, dedicated ML platforms (SageMaker, Vertex AI, MLflow and KServe) are purpose-built for that complexity.

PostgreSQL can serve models and even do basic A/B comparisons. But the gap between PostgreSQL's ML serving capabilities and dedicated platforms is wider than the gap in vector search (and this book says so explicitly in Chapter 7). Chapter 12 covers where PostgreSQL-centric MLOps fits in the broader landscape and provides concrete criteria for when to reach for external tools.

1.4.3. The Consolidation Argument

The purpose-built tools are excellent at their niches (nobody disputes that). But every additional system in your architecture adds operational cost that compounds over time:

- Another monitoring dashboard to check during incidents
- Another security audit surface for compliance reviews
- Another failure mode to handle in your runbooks
- Another credential rotation cycle
- Another vendor relationship to manage
- Another team member who needs to understand the system

The question isn't whether Pinecone can serve vectors faster than pgvector (at large enough scale, it can). The question is whether that performance difference justifies the operational complexity of running two systems instead of one.

For most teams, at most scales, the answer is no. Your 500K-product catalog doesn't need a dedicated vector database [1]; your RAG (Retrieval-Augmented Generation) pipeline doesn't need a separate embedding service; your feature store doesn't need Redis.

PostgreSQL can handle all of these, and this book shows you how, with code you can run today against the Docker environment we'll set up in Section 1.6.

For the teams and scales where the answer is yes, this book helps you make that decision with data instead of hype. Chapter 3 provides detailed comparison benchmarks; Chapter 9 gives you architecture decision frameworks; Chapter 13 tracks where the ecosystem is heading because the line between "PostgreSQL can't do this" and "PostgreSQL can do this" moves every few months.

Now that you know where PostgreSQL fits, let's figure out where *you* fit — and which path through this book will give you the most value.

1.5. Who This Book Is For

This book serves three distinct audiences. You'll recognize yourself in one of these personas; each comes with a recommended path through the chapters.

1.5.1. The Backend Engineer Adding AI Features

You have PostgreSQL experience. You've written migrations, optimized queries, maybe even set up replication. Now your team wants vector search, RAG-powered features, or ML-based recommendations, and you'd rather not introduce three new services to get there.

You know how to `EXPLAIN ANALYZE` a slow query, but you've never built an HNSW index. You've used `tsvector` for search, but you've never combined it with vector similarity. You want practical patterns that fit into your existing PostgreSQL deployment, not a research paper about embedding theory.

Your path: Start with Chapter 2 for a refresher on the PostgreSQL features that matter most for AI; you'll be surprised how many you haven't used (recursive CTEs, window functions with frame clauses, LATERAL joins). Then work through Chapter 3 and Chapter 4 for vector search and RAG (these are your bread and butter). When you're ready to architect a complete system, Chapter 9 and Chapter 10 show you production patterns and performance tuning.

You'll get the most value from following the running example sequentially, since each chapter builds on the last.

1.5.2. The Data Scientist Exploring PostgreSQL

You're comfortable with Python, pandas, and scikit-learn. You've trained models and built pipelines. But you're curious about running ML closer to the data: in-database feature engineering, SQL-based model serving, or using PostgreSQL as a feature store instead of maintaining a separate one.

You've hit the pain point where your feature pipeline has a `pandas.merge()` that takes 20 minutes because it's pulling millions of rows across a network. Or where your "real-time" features are actually 4 hours stale because the batch job that computes them runs on a schedule. You suspect there's a better way.

Your path: Start with Chapter 6 and Chapter 7. Feature engineering and in-database ML are where PostgreSQL will challenge your assumptions about what belongs in a database. Window functions and materialized views can replace significant chunks of your pandas pipeline, and they run where the data lives.

Then explore Chapter 3 for vector search; you'll see how SQL-based similarity search compares to your usual FAISS or Annoy workflows. Chapter 8 covers real-time AI patterns including streaming inference, CDC pipelines, and keeping external systems in sync with PostgreSQL.

You'll likely skim Chapter 2, but don't skip the window functions and CTEs sections. They're the foundation of everything in Chapter 6 and Chapter 7, and SQL window functions work differently than pandas `.rolling()` in ways that matter.

1.5.3. The Tech Lead Evaluating Consolidation

You're making build-vs-buy decisions. Your team is either considering PostgreSQL for new AI features or evaluating whether to migrate away from a multi-system architecture. You need real trade-offs, not vendor advocacy.

You've sat through vendor demos where everything looks easy. You've read blog posts that cherry-pick benchmarks. What you actually need is someone to tell you: "Here's what works, here's what doesn't, here's where the line is, and here's how to measure it for your specific situation."

Your path: Start with Chapter 9 for architecture patterns and decision frameworks; it directly addresses when PostgreSQL is and isn't the right choice, with concrete criteria instead of hand-waving. Chapter 11 covers the security model, which matters for compliance evaluations. Chapter 12 covers production operations, cost analysis with relative ratios, and the extension compatibility matrix you'll need for planning.

Then skim Chapter 3 for the candid pgvector-vs-dedicated-databases comparison; it's the section your team will argue about, and you'll want the actual numbers.

You probably won't read every code example. But the "When NOT to Use This" sections at the end of each chapter are written specifically for you; they're the clear-eyed assessment of where each approach breaks down.

1.5.4. Regardless of Your Path

No matter which persona you identify with, every chapter follows the same structure: a problem-focused opening, working code you can run against the book's Docker environment, trade-off analysis, and a "When NOT to Use This" section. Cross-references connect related topics across chapters, so you can follow threads without reading linearly. And the running example (an e-commerce product catalog with search, recommendations, and analytics) provides a consistent context that makes patterns concrete rather than abstract.

1.5.5. Prerequisites

This book assumes you have:

- **SQL fundamentals.** You can write SELECT, JOIN, WHERE, and GROUP BY without looking them up. If CTEs or window functions are new to you, Chapter 2 covers them from scratch.
- **Docker basics.** You know what `docker compose up` does. That's about it; the book's Docker Compose setup handles PostgreSQL, all extensions, and supporting services automatically.
- **Command line comfort.** You can run `psql`, navigate directories, and read terminal output without anxiety. Nothing in this book requires complex shell scripting.
- **Python basics** for the embedding generation and LLM integration scripts in Chapter 3 and Chapter 4. You need to read Python, install packages with pip, and run scripts. If you're focused on SQL-only chapters (Chapter 2, Chapter 6, Chapter 7, Chapter 9, Chapter 10), Python isn't required.

What you do *not* need:

- **No machine learning prerequisites.** This book teaches ML concepts from scratch, in the context of PostgreSQL. You don't need to know what an embedding is, how a neural network works, or what cosine similarity measures; we'll get there.
- **No AI/LLM experience.** If you've never called an LLM API or built a RAG pipeline, Chapter 4 starts from zero and builds up.

- **No PostgreSQL extension experience.** Every extension used in this book (pgvector, pgai, pgvectorscale, PostgresML, TimescaleDB, pg_cron) is installed via Docker, configured, and explained as it's introduced. You don't need to know how `shared_preload_libraries` works before you start.

1.6. Your First AI Query

Enough context: let's run something. In the next few minutes, you'll perform a semantic search that finds products by *meaning*, not keywords. No Python. No external vector database. Just SQL.

1.6.1. Step 1: Start the Environment

Bash

```bash
git clone https://github.com/pgsql-ai-book/postgresql-ai-book && cd
  ↪ postgresql-ai-book
docker compose up -d
```

The `docker-compose.yml` defines a PostgreSQL service with pgvector and pgai extensions; Ollama runs directly on your host machine:

YAML

```yaml
services:
  db:
    build: ./recsys/docker                # PG17 + pgvector + pgai +
  ↪ TimescaleDB
    ports:
      - "5432:5432"
    environment:
      POSTGRES_DB: ai_db
      POSTGRES_USER: postgres
      # ...
    volumes:
      - ./recsys/docker/init:/docker-entrypoint-initdb.d  # seeds schema + data
      - ./recsys:/recsys
```

⚠ Linux Users: Network Configuration

On Linux, `host.docker.internal` is not available by default. You have two options:

Option 1: Add to docker-compose.yml

YAML

```yaml
extra_hosts:
  - "host.docker.internal:host-gateway"
```

Option 2: Use the host network

YAML

```yaml
network_mode: "host"
```

This affects connections from Docker containers to services running on the host machine (like Ollama). Mac and Windows users don't need this configuration.

Wait about 30 seconds for PostgreSQL to initialize extensions and seed 1,000 products. Meanwhile, ensure Ollama (a tool for running open-source AI models locally on your machine) is installed, and pull the embedding model:

Bash

```bash
ollama pull nomic-embed-text
```

1.6.2. Step 2: Connect

Bash

```bash
psql -h localhost -p 5432 -U postgres -d ai_db
```

1.6.3. Step 3: Generate Embeddings

The product catalog is loaded but doesn't have embeddings yet — those get generated in Chapter 3. The Docker init script creates a `products` table with an `embedding vector(768)` column, matching the 768-dimensional output of the `nomic-embed-text` model [10]. For this taste, let's embed a small batch:

SQL

```sql
UPDATE products
SET embedding = ai.ollama_embed(
    'nomic-embed-text',
    name || ' ' || description,
    host => 'http://host.docker.internal:11434'
)
WHERE id <= 100;  -- just the first 100 for a quick taste
```

We embed just the first 100 products to keep this quick taste under a minute — Chapter 3 covers batch strategies for the full catalog.

1.6.4. Step 4: Semantic Search — The Payoff

SQL

```sql
WITH query AS (
    SELECT ai.ollama_embed(
        'nomic-embed-text',
        'comfortable running shoes for beginners',
        host => 'http://host.docker.internal:11434'
    ) AS embedding
)
SELECT p.name, p.category,
       1 - (p.embedding <=> q.embedding) AS similarity
FROM products p, query q
WHERE p.embedding IS NOT NULL
ORDER BY p.embedding <=> q.embedding
LIMIT 5;
```

You'll see results like lightweight running shoes, shock-absorbing sneakers, and breathable athletic footwear — even though none of those product names contain the word "comfortable" or "beginners." The search understood what you *meant*.

1.6.5. What Just Happened

You searched products using meaning, not keywords. pgvector's `<=>` operator computed cosine distance between your query embedding and every product embedding [11]. pgai called the `nomic-embed-text` model via Ollama to generate the query embedding right inside

SQL. No application code. No external vector database. No data leaving PostgreSQL. This is the foundation of Retrieval-Augmented Generation [12].

> **If This Looks Like Magic**
>
> It's not — it's linear algebra and a well-trained neural network. Chapter 3 breaks down exactly how embeddings represent meaning as numbers, why cosine distance measures semantic similarity, and how HNSW indexes make this fast enough for production.

1.7. What You'll Build

Over thirteen chapters, you'll build **RecSys**: a complete AI-powered product recommendation system. Each chapter adds a capability to the same product catalog; by the end, RecSys handles semantic search, natural language Q&A, demand forecasting, real-time recommendations, and production deployment.

The book is organized in four parts that mirror the way real AI projects evolve: foundations first, then core capabilities, then production hardening, then deployment. Figure 1.1 shows how the pieces fit together; PostgreSQL sits at the center, with each chapter adding a capability to RecSys.

Table 1.1.: Book organization and the extensions each part introduces

Part	Chapters	What You'll Learn	Key Extensions
I: Foundations	1–2	PostgreSQL's built-in AI features	Core PostgreSQL
II: Core AI	3–7	Vector search, RAG, features, ML	pgvector, pgai, PostgresML
III: Production	8–10	Real-time, architecture, performance	TimescaleDB, Debezium
IV: Operations & Beyond	11–13	Security, cloud deploy, operations, future	All extensions

1.7.1. Part I: Foundations (Chapters 1–2)

Chapter 2: Modern PostgreSQL for AI Workloads (Chapter 2)

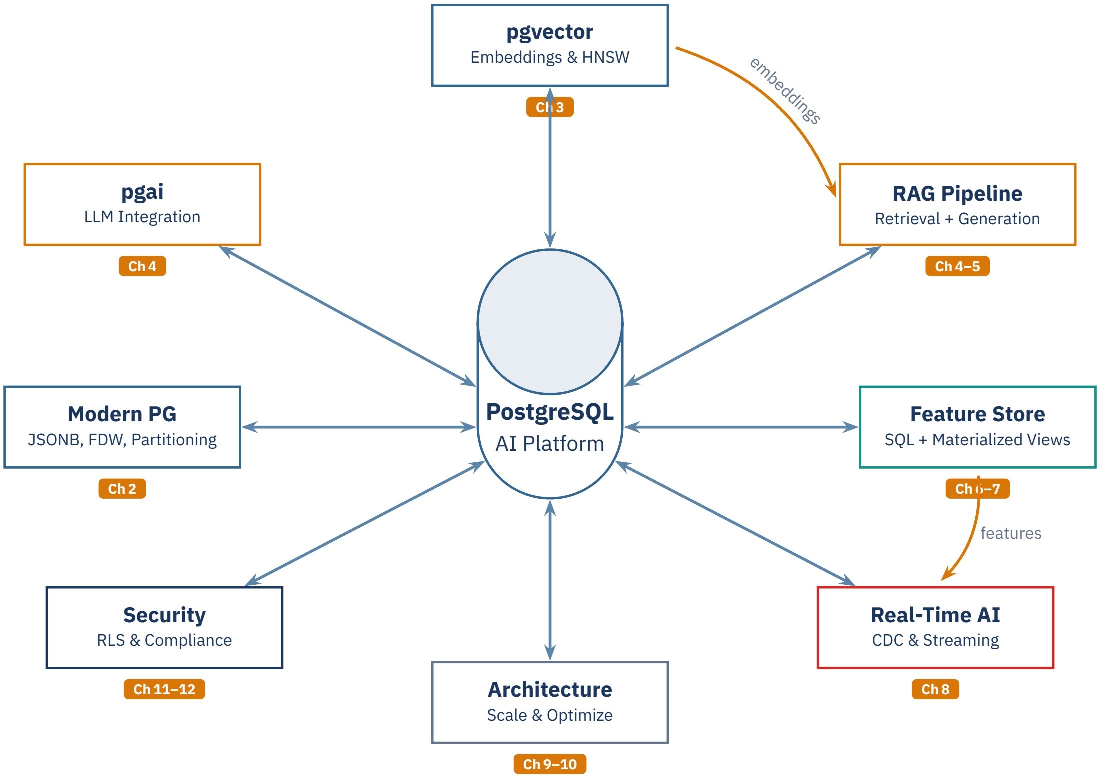

Figure 1.1.: RecSys system architecture: PostgreSQL at the center with surrounding AI capabilities and their chapter references.

Before you install a single extension, PostgreSQL already speaks AI. This chapter covers the features you have but might not be using for AI workloads: JSONB for storing model metadata and experiment configs, native arrays for embedding storage, full-text search with weighted fields and linguistic stemming, window functions for time-aware feature computation, and foreign data wrappers for cross-database ML pipelines. (No extensions required.)

Running example: The product catalog gains an AI-ready schema with JSONB specs, full-text search vectors, and array-based embedding columns — no extensions required.

1.7.2. Part II: Core AI Capabilities (Chapters 3–7)

Chapter 3: Vector Search with pgvector (Chapter 3)

This is where the quick taste becomes a full tutorial. You'll learn the math behind embeddings (what they represent, why cosine distance measures semantic similarity, and how the geometry actually works), then generate embeddings with pgai (SQL) and Python batch pipelines using nomic-embed-text. You'll choose between HNSW, IVFFlat, and DiskANN indexes based on dataset size and query patterns, and implement hybrid search combining vector similarity with full-text search using Reciprocal Rank Fusion.

Running example: The product catalog gets production-grade semantic search. A query for "comfortable shoes for standing all day" returns relevant products regardless of keyword overlap.

Chapter 4: RAG Fundamentals (Chapter 4)

Vector search returns relevant results, but users want answers, not ranked lists. This chapter builds retrieval-augmented generation (RAG) pipelines that combine PostgreSQL's search with large language models. You'll call LLMs directly from SQL with pgai, automate embedding generation with Vectorizer, apply chunking strategies for different data types, and build your first RAG pipeline end-to-end.

Running example: The product catalog becomes a conversational Q&A chatbot. Users ask "What's a good gift for a runner under $50?" and get grounded, accurate recommendations.

Chapter 5: Advanced RAG Patterns (Section 5.4)

Production RAG pipelines need more than basic retrieval. This chapter covers the patterns that separate demo-quality RAG from production-quality RAG: re-ranking retrieved results for precision, contextual compression to reduce noise, multi-query retrieval for complex questions, parent-child document strategies, and agentic RAG with tool-calling LLMs that can query multiple data sources. You'll also build text-to-SQL pipelines (Section 5.11) that convert natural language questions into database queries with five-layer safety guarantees, plus evaluation frameworks to measure RAG quality systematically.

Running example: The product Q&A chatbot gets smarter — handling multi-step questions, filtering by attributes, and providing grounded answers with citations.

Chapter 6: Feature Engineering and Feature Store (Chapter 6)

What if a single SQL JOIN could silently destroy a model that took weeks to build? This chapter opens with a data leakage cautionary tale [13], [14] (94% accuracy in development, 61% in production), then systematically builds the SQL patterns that prevent it. Feature engineering pipelines must be treated as first-class infrastructure [15], [16]. You'll compute rolling aggregations with window functions, cache features with materialized views and TimescaleDB continuous aggregates, build a lightweight feature registry for versioning and lineage, and learn the seven most common feature store anti-patterns.

Running example: 200 users across four behavioral archetypes generate events that become eight features: purchase counts, cart values, activity acceleration, and recency signals. These feed directly into Chapter 7's ML models.

Chapter 7: In-Database Machine Learning (Chapter 7)

What if training a model was a SQL query? Using PostgresML, you'll train regression and classification models with pgml.train(), compare scikit-learn, XGBoost, and LightGBM with a single function call, and serve predictions via pgml.predict() in standard SQL queries. When PostgresML doesn't cover your use case, PL/Python provides the escape hatch; we use it for anomaly detection with Isolation Forest and custom pipelines.

Running example: Demand forecasting predicts which products will sell next week. Price optimization uses scenario generation across 13 price points per product. Both models live inside PostgreSQL, serving predictions in the same transaction that reads data.

1.7.3. Part III: Production Patterns (Chapters 8–10)

Chapter 8: Real-Time AI Pipelines (Chapter 8)

Your recommendation model is accurate (on yesterday's data). This chapter closes the freshness gap with three progressively capable patterns: LISTEN/NOTIFY for lightweight event-driven triggers, SELECT ... FOR UPDATE SKIP LOCKED for durable job queues at moderate scale, and full Change Data Capture with Debezium streaming through Kafka for high-throughput production pipelines.

Running example: When a user adds a product to their cart, the recommendation model re-scores related products within seconds — not hours.

Chapter 9: AI Platform Architecture (Chapter 9)

The most common AI architecture mistake isn't choosing the wrong database; it's expecting one database to do everything. This chapter assembles everything from Chapters 2–8 into coherent system designs. You'll implement event sourcing for AI workflows, build agentic AI infrastructure with tool registries and function-calling format, apply multi-tenant isolation with Row-Level Security, and design data pipelines that move features between PostgreSQL and external systems.

Running example: A complete platform architecture diagram shows how every component connects — from product catalog to embedding pipeline to recommendation engine to security layer. Six "When NOT to Use" criteria define clear boundaries.

Chapter 10: Performance Optimization (Chapter 10)

Default PostgreSQL settings assume traditional OLTP workloads. AI workloads (large embedding scans, HNSW index builds consuming gigabytes of memory, parallel similarity computations) need fundamentally different configuration. You'll tune server settings using concrete formulas and three server profiles, establish rigorous benchmarking methodology (five metrics, five common mistakes), optimize vector search query plans with EXPLAIN ANALYZE, configure connection pooling for vector workloads, and partition large datasets.

Running example: The product catalog scales to 100K vectors with benchmarks comparing pgvector, pgvectorscale, and VectorChord. Every performance claim cites methodology and numbers.

1.7.4. Part IV: Operations & Beyond (Chapters 11–13)

Chapter 11: Security, Privacy, and Governance (Chapter 11)

AI systems create attack surfaces that traditional database security doesn't cover. Embeddings can be inverted to reconstruct source text; prompt injection can bypass application logic; feature stores can leak training data across tenant boundaries. This chapter builds six-layer defense-in-depth: Row-Level Security, encryption (encrypting source text while leaving embeddings searchable), differential privacy, GDPR compliance with erasure chains, audit logging, and AI-specific threat defense.

Running example: The product platform gets a complete security hardening with a 15-item checklist organized by defense layer.

Chapter 12: Production Deployment (Chapter 12)

The demo worked perfectly (then someone opened the AWS console). Three of four core extensions didn't exist on RDS. This chapter's centerpiece is an extension compatibility matrix showing exactly which cloud providers support which extensions. Beyond provider selection, you'll set up AI-specific monitoring, backup strategies that account for HNSW rebuild times, high availability with Patroni, CI/CD pipelines with soak-period rollback, and cost analysis using relative ratios that won't age.

Running example: A six-concern production walkthrough: provider selection, monitoring, backup, HA, CI/CD, and cost — applied to RecSys.

Chapter 13: The Future of PostgreSQL and AI (Chapter 13)

The ecosystem moves fast; pgvector went from its first release in April 2021 [17] to production standard in under three years. This chapter uses a structured tier-label framework: EXISTS TODAY, IN DEVELOPMENT, and SPECULATIVE. You'll evaluate PostgreSQL 18 features (async I/O, virtual columns, parallel GIN, UUIDv7), the pgvector roadmap, emerging extensions, and practical next steps for your own projects.

Running example: The chapter closes with a clear-eyed final assessment of where PostgreSQL stands in the AI landscape — and a concrete plan for what to build next.

1.7.5. RecSys: The Running Example Across All Chapters

Throughout this book, we build **RecSys** — a product recommendation system that showcases every PostgreSQL AI capability. RecSys isn't just a teaching device — it's the connective tissue of the entire book. Here's how it evolves:

1. **Chapter 2:** RecSys starts — 1,000 products with JSONB specs, full-text search vectors, and array columns
2. **Chapter 3:** RecSys gains 768-dimensional embeddings and semantic search capability
3. **Chapter 4:** RecSys becomes conversational — users ask questions, get grounded answers via RAG
4. **Section 5.4:** RecSys gets advanced retrieval with re-ranking, compression, and evaluation
5. **Chapter 6:** RecSys builds user features — 200 users generate purchase patterns and activity signals
6. **Chapter 7:** RecSys trains ML models to predict demand and optimize pricing
7. **Chapter 8:** RecSys goes real-time — pipelines update recommendations as users browse and buy
8. **Chapter 9:** RecSys assembles into a platform with clear architectural boundaries
9. **Chapter 10:** RecSys gets performance-tuned — scaling to 100K vectors with benchmarks
10. **Chapter 11:** RecSys implements security — RLS, encryption, privacy, and audit
11. **Chapter 12:** RecSys deploys to production with monitoring, HA, and CI/CD
12. **Chapter 13:** RecSys looks ahead — PostgreSQL 18, multimodal, and ecosystem evolution

By the final chapter, you'll have built something that would pass a serious architecture review — not because it's complex, but because every component earns its place.

> Reading Order
>
> The book is designed for sequential reading — each chapter builds on the previous. If you're experienced with PostgreSQL, you can skim Chapter 2. If you only care about vector search, Chapters 3–5 are self-contained after the setup steps. But the full value comes from seeing how the pieces connect across all thirteen chapters.

1.8. When PostgreSQL Is NOT Right for AI

This book argues that PostgreSQL handles more AI workloads than most teams realize. But intellectual honesty requires acknowledging the workloads where it's the wrong choice. Here are the signals that should send you elsewhere.

1.8.1. Sub-Millisecond Vector Search at Very Large Scale

If you need sub-millisecond latency over 10+ million vectors with high query concurrency [8], [9], purpose-built vector databases like Pinecone, Weaviate, or Milvus offer optimizations PostgreSQL can't match: GPU-accelerated indexing, purpose-built memory management, and query routing designed exclusively for similarity search.

pgvector and pgvectorscale are fast for most workloads. Chapter 3 covers the exact performance boundaries with reproducible benchmarks so you can make this decision with data, not marketing claims.

1.8.2. GPU-Intensive Model Training and Inference

PostgreSQL runs on CPU. If your workload requires fine-tuning large language models, training deep neural networks, or serving models that need GPU acceleration for acceptable latency, you need dedicated ML infrastructure: SageMaker, Vertex AI, or self-hosted GPU clusters.

PostgresML handles classical ML algorithms well (XGBoost [18], LightGBM [19], scikit-learn), but for anything requiring GPU compute, Chapter 7 is clear about where its capabilities end and when to reach for external tools.

1.8.3. Teams with Zero SQL Experience

PostgreSQL rewards SQL fluency. Every pattern in this book (from window functions for feature engineering to recursive CTEs for knowledge graphs) assumes you're comfortable writing SQL. If your team is entirely Python-native and the overhead of learning SQL patterns outweighs the architectural benefits, a Python-native stack (LangChain + a managed vector database) will get you to production faster.

The trade-off: you'll pay for that speed with operational complexity later. But that's a valid choice if time-to-market matters more than infrastructure simplicity.

1.8.4. Regulatory Environments Requiring Certified AI Platforms

Some industries require AI systems built on platforms with formal certification — SOC 2 Type II for the AI layer specifically, FDA validation for medical AI, or EU AI Act compliance tooling with built-in audit trails and model cards. PostgreSQL extensions don't carry these certifications.

Specialized vendors like Dataiku, H2O.ai, or Azure ML offer the compliance documentation your auditors need. You can still use PostgreSQL as the data layer underneath — Chapter 9 discusses hybrid architectures where PostgreSQL handles storage and certified platforms handle the regulated AI operations.

1.8.5. Cutting-Edge Model Architectures

PyTorch and TensorFlow add new architectures weekly. PostgresML and pgai support a specific set of models and algorithms. If you need the latest transformer variant or custom model architecture the day it's published, work directly with the ML frameworks.

That said, most production AI systems don't need cutting-edge architectures — they need reliable ones. If your use case is served by established models (embedding generation, classification, regression, anomaly detection), PostgreSQL covers it well.

1.8.6. Quick Decision Framework

Ask yourself these five questions:

1. **Scale:** Will you exceed pgvector's scale ceiling with sub-millisecond latency requirements (see Section 3.19)?
2. **Compute:** Do you need GPU for training or inference?
3. **Team:** Does your team have basic SQL proficiency (or willingness to learn)?
4. **Compliance:** Do you need formally certified AI platforms for regulatory audits?
5. **Models:** Do you need the very latest model architectures on release day?

If you answered "yes" to any of these, the relevant chapter's "When NOT to Use" section will help you decide whether PostgreSQL handles your specific variant of the concern — or whether you should delegate that piece to a purpose-built tool.

If you answered "no" to all five, you're in PostgreSQL's sweet spot.

Each chapter includes its own "When NOT to Use" section with specific thresholds and decision criteria for that chapter's topic: vector search scale limits in Chapter 3, in-database ML boundaries in Chapter 7, architectural boundaries in Chapter 9, security delegation in Chapter 11, and deployment constraints in Chapter 12. The pattern throughout this book is: use PostgreSQL where it excels, delegate where it doesn't, and know exactly where the boundary is.

Further Reading

Bracketed numbers refer to entries in Appendix B: References.

- **[20]** — The original POSTGRES design paper by Stonebraker and Rowe. Essential for understanding why PostgreSQL's extensible type system (the foundation that makes pgvector, pgai, and every other AI extension possible) was a deliberate architectural choice, not an afterthought.
- **[6]** — The official PostgreSQL 17 documentation remains the single most authoritative reference for every feature discussed in this book. Bookmark the chapters on extensions, configuration, and SQL syntax; you'll return to them constantly.
- **[1]** — The pgvector GitHub repository doubles as its documentation. Read the README for installation, operator reference, and index tuning parameters. The issue tracker is also valuable — real-world production questions get detailed answers from the maintainer.
- **[2]** — Timescale's pgai extension documentation covers both the SQL extension (in-database LLM calls) and the Vectorizer Python library (automated embedding pipelines). Start with the quickstart guide to understand the two-component architecture.
- **[7]** — Zaharia et al.'s blog post on compound AI systems articulates why the industry is shifting from monolithic models to systems that combine retrieval, generation, and traditional software. This frames the entire premise of this book: PostgreSQL as the data backbone of compound AI systems.

1.9. Summary

This chapter made the case for PostgreSQL as an AI platform: not as a replacement for every specialized tool, but as a consolidation opportunity for teams already running PostgreSQL.

What we covered:

- **The problem** (Section 1.3): Infrastructure sprawl from separate vector databases, caching layers, and embedding services; the operational tax of monitoring, backup, and security for each moving piece
- **PostgreSQL's position** (Section 1.4): pgvector for similarity search, pgai for embedding generation and LLM integration, pgvectorscale for large-scale indexing, PostgresML for in-database model training, and TimescaleDB for time-series feature computation

- **Reader personas** (Section 1.5): Three paths through the book—backend engineers adding AI to existing PostgreSQL deployments, data scientists exploring in-database ML and feature engineering, and tech leads evaluating consolidation from multi-system architectures
- **Quick taste** (Section 1.6): A working semantic search query in under five minutes, demonstrating the core value proposition
- **Book roadmap** (Section 1.7): Chapter-by-chapter progression from foundations through production deployment
- **When NOT to use** (Section 1.8): Five decision criteria where PostgreSQL is the wrong choice—sub-millisecond search on 10+ million vectors, GPU-intensive training and inference, teams with zero SQL experience, regulatory certifications requiring specialized vendors, and cutting-edge model architectures needing latest frameworks

Key takeaways:

1. PostgreSQL has quietly accumulated production-grade AI capabilities; these aren't toys, they're used by companies processing millions of queries daily
2. The core argument is architectural simplicity; every external service adds monitoring, backup, security, and operational overhead
3. Consolidation has limits; every chapter adds specific thresholds where dedicated tools outperform
4. The goal isn't to use PostgreSQL for everything; it's to use it for everything it's good at, and know exactly when to reach for something else

If you're still reading, PostgreSQL is probably right for your use case. Let's get started with the foundation: the PostgreSQL features you already have that make everything else possible.

1.10. Exercises

These exercises reinforce the key concepts from this chapter. They use only the Docker environment and tools introduced above; no additional setup required.

1.10.1. Exercise 1.1: Verify Your Environment

Difficulty: Beginner | **Time:** ~5 min

Start the book's Docker environment and connect to the database:

Bash

```
docker compose up -d
psql -h localhost -p 5432 -U postgres -d ai_db
```

Run the following queries and confirm the output:

SQL

```sql
SELECT extname, extversion FROM pg_extension ORDER BY extname;
SELECT count(*) FROM products;
```

Questions:

1. Which AI-related extensions are installed? (Look for `vector`, `ai`, and `vectorscale`.)
2. How many products are in the seed dataset?
3. What happens if you run `SELECT embedding FROM products LIMIT 1;`? Is the column populated or `NULL`?

1.10.2. Exercise 1.2: Explore the Product Catalog with SQL

Difficulty: Beginner | **Time:** ~10 min

Using basic SQL, answer the following about the product catalog:

SQL

```sql
-- 1. How many distinct categories exist?
SELECT count(DISTINCT category) FROM products;

-- 2. Which category has the most products?
SELECT category, count(*) AS n
FROM products
GROUP BY category
ORDER BY n DESC
LIMIT 5;

-- 3. What is the price range (min, max, average)?
SELECT min(price), max(price), round(avg(price), 2) AS avg_price
FROM products;
```

Now write your own query: find the 5 most expensive products and display their name, category, and price. Sort by price descending.

1.10.3. Exercise 1.3: Semantic Search with a Different Query

Difficulty: Intermediate | **Time:** ~10 min

In Section 1.6 you searched for *"comfortable running shoes for beginners."* Now try a different query to see how semantic search handles meaning.

First, ensure embeddings exist for the first 100 products (if you haven't already):

SQL

```sql
UPDATE products
SET embedding = ai.ollama_embed(
    'nomic-embed-text',
    name || ' ' || description,
    host => 'http://host.docker.internal:11434'
)
WHERE id <= 100 AND embedding IS NULL;
```

Then run a semantic search with a query of your choice. Some suggestions:

- *"eco-friendly kitchen gadgets"*
- *"gift for a teenager who likes music"*
- *"durable outdoor gear for camping"*

SQL

```sql
WITH query AS (
    SELECT ai.ollama_embed(
        'nomic-embed-text',
        'your query here',  -- Replace with your query
        host => 'http://host.docker.internal:11434'
    ) AS embedding
)
SELECT p.name, p.category,
       round((1 - (p.embedding <=> q.embedding))::numeric, 4) AS similarity
FROM products p, query q
WHERE p.embedding IS NOT NULL
ORDER BY p.embedding <=> q.embedding
LIMIT 5;
```

Questions:

1. Do the results make sense even when none of the returned product names contain words from your query?

2. What happens if you search for something completely unrelated to the catalog (e.g., *"quantum physics textbook"*)? Are the similarity scores noticeably lower?

1.10.4. Exercise 1.4: Cosine Distance vs. Inner Product

Difficulty: Intermediate | **Time:** ~15 min

pgvector provides three distance operators:

Operator	Distance Metric	Lower = More Similar?
`<=>`	Cosine distance	Yes
`<#>`	Negative inner product	Yes
`<->`	L2 (Euclidean) distance	Yes

Using the same query embedding from Exercise 1.3, run three versions of the search: one with each operator. Compare the top-5 results:

SQL

```sql
-- Cosine distance (used in the chapter)
ORDER BY p.embedding <=> q.embedding

-- Negative inner product
ORDER BY p.embedding <#> q.embedding

-- L2 distance
ORDER BY p.embedding <-> q.embedding
```

Questions:

1. Do all three operators return the same top-5 products, or do the rankings differ?
2. Based on the results, why do you think cosine distance is the default choice for text embeddings? (Hint: think about what happens when embedding magnitudes vary.)
3. Chapter 3 covers this in depth. Form a hypothesis now and check it when you get there.

1.10.5. Exercise 1.5: When Would You NOT Use PostgreSQL?

Difficulty: Advanced | **Time:** ~15 min

This is a thought exercise; no SQL required.

Re-read Section 1.8 and the Quick Decision Framework's five questions. Then evaluate the following three scenarios and decide: **PostgreSQL only, hybrid architecture, or external tool?**

1. **Scenario A:** A startup with 50,000 products, 3 engineers, and a PostgreSQL database already in production. They want to add semantic search to their product catalog.
2. **Scenario B:** A medical imaging company processing 10 million CT scans. They need GPU-accelerated inference for a deep learning model that classifies tumors, with FDA audit trail requirements.
3. **Scenario C:** A social media platform with 500 million user-generated posts. They need sub-millisecond vector search for a "similar posts" feature serving 100,000 queries per second.

For each scenario, justify your answer by referencing the specific criteria from the chapter (scale, compute, team, compliance, models).