

Building Intelligent Applications

PostgreSQL for AI

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



Table of contents

Preface	1
Who This Book Is For	1
The Running Example	1
How to Read This Book	2
Dedication	3
About the Author	4
Acknowledgments	5
Open Source Projects and Their Creators	5
Technical Reviewers	6
Community and Inspiration	6
Personal	7
SQL Refresher	8
Reading Guide	9
I. Foundations	11
1. Introduction: Why PostgreSQL for AI?	14
1.1. Learning Objectives	14
1.2. Prerequisites	14
1.3. The Problem You Already Have	15
1.4. PostgreSQL in the AI Landscape	15
1.4.1. Where PostgreSQL Genuinely Excels	16
1.4.2. Where PostgreSQL Genuinely Struggles	17
1.4.3. The Consolidation Argument	18
1.5. Who This Book Is For	19
1.5.1. The Backend Engineer Adding AI Features	19
1.5.2. The Data Scientist Exploring PostgreSQL	20
1.5.3. The Tech Lead Evaluating Consolidation	20
1.5.4. Regardless of Your Path	21
1.5.5. Prerequisites	21
1.6. Your First AI Query	22
1.6.1. Step 1: Start the Environment	22
1.6.2. Step 2: Connect	23
1.6.3. Step 3: Generate Embeddings	23
1.6.4. Step 4: Semantic Search — The Payoff	24
1.6.5. What Just Happened	24
1.7. What You’ll Build	25
1.7.1. Part I: Foundations (Chapters 1–2)	25
1.7.2. Part II: Core AI Capabilities (Chapters 3–7)	26

1.7.3.	Part III: Production Patterns (Chapters 8–10)	28
1.7.4.	Part IV: Operations & Beyond (Chapters 11–13)	29
1.7.5.	RecSys: The Running Example Across All Chapters	30
1.8.	When PostgreSQL Is NOT Right for AI	30
1.8.1.	Sub-Millisecond Vector Search at Very Large Scale	31
1.8.2.	GPU-Intensive Model Training and Inference	31
1.8.3.	Teams with Zero SQL Experience	31
1.8.4.	Regulatory Environments Requiring Certified AI Platforms	31
1.8.5.	Cutting-Edge Model Architectures	32
1.8.6.	Quick Decision Framework	32
	Further Reading	33
1.9.	Summary	33
1.10.	Exercises	34
1.10.1.	Exercise 1.1: Verify Your Environment	34
1.10.2.	Exercise 1.2: Explore the Product Catalog with SQL	35
1.10.3.	Exercise 1.3: Semantic Search with a Different Query	36
1.10.4.	Exercise 1.4: Cosine Distance vs. Inner Product	37
1.10.5.	Exercise 1.5: When Would You NOT Use PostgreSQL?	37
2.	Modern PostgreSQL for AI Workloads	40
2.1.	Learning Objectives	40
2.2.	Prerequisites	41
2.3.	Building an AI-Ready Schema	41
2.3.1.	The Problem	42
2.3.2.	Why This Matters	42
2.3.3.	The Base Schema	43
2.3.4.	What’s Coming	45
2.3.5.	Running the Seed Script	45
2.3.6.	The Approach	46
2.4.	JSONB for AI Metadata	46
2.4.1.	Why JSONB for AI	47
2.4.2.	The AI Metadata Column	47
2.4.3.	JSONB Operators	48
2.4.4.	SQL/JSON Path Queries (PostgreSQL 12+)	49
2.4.5.	JSON_TABLE (PostgreSQL 17)	50
2.4.6.	GIN Indexing for JSONB	51
2.4.7.	Performance: JSONB vs. Normalized Tables	52
2.4.8.	Practical Patterns	53
2.5.	Array Types for Embedding Storage	54
2.5.1.	Native Arrays in PostgreSQL	54
2.5.2.	Declaring and Inserting Arrays	55
2.5.3.	Array Operators	55
2.5.4.	Array Functions	56
2.5.5.	Computing Cosine Similarity in Pure SQL	57
2.5.6.	The Performance Wall	58
2.5.7.	Dot Product and Euclidean Distance	59
2.5.8.	Beyond Embeddings: Practical Array Uses	60
2.5.9.	Summary: Arrays for AI	61

2.6.	Custom Types and Domains for AI Data	61
2.6.1.	Composite Types	62
2.6.2.	Enum Types	63
2.6.3.	Domains	64
2.6.4.	Combining Types and Domains	65
2.6.5.	Trade-offs: Types vs JSONB	66
2.6.6.	Summary: Custom Types for AI	67
2.7.	Full-Text Search Fundamentals	67
2.7.1.	The Core Concepts: tsvector and tsquery	67
2.7.2.	GENERATED ALWAYS AS STORED: The Right Pattern	68
2.7.3.	GIN Indexing for Fast Search	69
2.7.4.	Weighted Search with setweight	70
2.7.5.	Ranking Results	70
2.7.6.	Query Types	71
2.7.7.	Phrase Search and Proximity	72
2.7.8.	Configuration and Languages	73
2.7.9.	Clear Boundaries	73
2.7.10.	Summary: Full-Text Search for AI	74
2.8.	Window Functions and CTEs for Feature Engineering	74
2.8.1.	Window Function Fundamentals	75
2.8.2.	Essential Window Functions for AI	75
2.8.3.	Frame Specifications	77
2.8.4.	Named WINDOW Clause	78
2.8.5.	Practical AI Example: Multi-Feature Computation	79
2.8.6.	CTEs: Readable Query Composition	80
2.8.7.	Materialized vs Non-Materialized CTEs	81
2.8.8.	Recursive CTEs	82
2.8.9.	Trade-offs and Performance	83
2.8.10.	Summary: Window Functions and CTEs for AI	84
2.9.	Parallel Query Execution for AI Workloads	84
2.9.1.	How Parallel Query Works	84
2.9.2.	Seeing Parallelism in Action	85
2.9.3.	Configuration Parameters	86
2.9.4.	What Parallelizes (and What Doesn't)	87
2.9.5.	PostgreSQL 17 Improvements	89
2.9.6.	Trade-offs and Practical Advice	89
2.10.	Foreign Data Wrappers for Cross-Database ML	90
2.10.1.	The Setup: Connecting to PostgresML	91
2.10.2.	Querying Across Databases	92
2.10.3.	How FDW Executes Queries	92
2.10.4.	The Statistics Blindness Problem	93
2.10.5.	Controlling Fetch Behavior	94
2.10.6.	Transaction and Consistency Pitfalls	95
2.10.7.	Performance Considerations	95
2.10.8.	Other Foreign Data Wrappers	96
2.10.9.	When NOT to Use FDW	97
2.11.	The Complete AI-Ready Product Catalog	97
2.11.1.	The Evolved Schema	97
2.11.2.	A Query That Uses Everything	99

2.11.3.	How This Schema Feeds Later Chapters	101
2.11.4.	When NOT to Use Standard PostgreSQL for AI	102
Further Reading	104
2.12.	Summary	104
2.13.	Exercises	105
2.13.1.	Exercise 2.1: Query AI Metadata with JSONB	106
2.13.2.	Exercise 2.2: Full-Text Search with Weighted Ranking	106
2.13.3.	Exercise 2.3: Window Functions for Category Analytics	106
2.13.4.	Exercise 2.4: CTE Pipeline: Search, Rank, and Analyze	107

II. Core AI Capabilities 109

3.	Vector Search with pgvector	112
3.1.	Learning Objectives	112
3.2.	Prerequisites	113
3.3.	The Keyword Search Problem	113
3.4.	From Words to Vectors	114
3.4.1.	The Algebra of Meaning	116
3.4.2.	Terminology: Vector Search, Similarity Search, and Semantic Search	116
3.5.	The Mathematics of Similarity	117
3.5.1.	Cosine Similarity and Distance	117
3.5.2.	Euclidean Distance (L2)	118
3.5.3.	Inner Product	118
3.5.4.	Manhattan Distance (L1)	118
3.6.	Getting Started with pgvector	119
3.6.1.	Installing pgvector	119
3.6.2.	Vector Data Types	120
3.6.3.	Working with Vectors	121
3.6.4.	Your First Vector Table	121
3.6.5.	From Keyword Failure to Vector Success	122
3.6.6.	Basic Vector Operations in SQL	123
3.6.7.	What We've Built So Far	124
3.7.	Distance Functions in Practice	124
3.7.1.	The Four Operators	124
3.7.2.	Same Query, Different Rankings	125
3.7.3.	When Rankings Diverge	126
3.7.4.	Performance Characteristics	126
3.7.5.	Choosing Your Operator: A Decision Guide	127
3.7.6.	Practical Tips	128
3.8.	Generating Embeddings	129
3.8.1.	Choosing an Embedding Model	129
3.8.2.	pgai – Embeddings Inside SQL	131
3.8.3.	Python Batch Pipeline	132
3.8.4.	The Embedding Model Landscape	134
3.8.5.	Setting Up Dimension Comparison	136
3.9.	HNSW: The Default Choice	137
3.9.1.	How HNSW Works	137
3.9.2.	Creating an HNSW Index	138

3.9.3.	Before and After: EXPLAIN ANALYZE	139
3.9.4.	Search-Time Tuning with ef_search	141
3.9.5.	HNSW Trade-Offs	142
3.10.	IVFFlat: When Simplicity Wins	144
3.10.1.	How IVFFlat Works	144
3.10.2.	Creating and Tuning an IVFFlat Index	144
3.10.3.	When to Prefer IVFFlat Over HNSW	145
3.11.	Scaling Up: DiskANN and Quantization	146
3.11.1.	DiskANN: Disk-Based Approximate Search	147
3.11.2.	Quantization: Shrinking Vectors	148
3.11.3.	Memory Savings at Scale	150
3.11.4.	Choosing Your Index Strategy	151
3.12.	Hybrid Search: The Best of Both Worlds	152
3.12.1.	PostgreSQL Full-Text Search Refresher	152
3.12.2.	Reciprocal Rank Fusion	153
3.12.3.	Implementing RRF in SQL	153
3.12.4.	Production Hybrid Search	155
3.12.5.	Index Strategy for Hybrid Search	157
3.12.6.	When to Use Hybrid Search	158
3.13.	Filtered Vector Search	158
3.13.1.	The Filtering Problem	158
3.13.2.	pgvector 0.8.0 Iterative Index Scans	159
3.13.3.	Practical Filter Patterns	161
3.13.4.	Best Practices	163
3.14.	Production pgvector	164
3.14.1.	Addressing “The Case Against pgvector”	164
3.14.2.	Index Maintenance	165
3.14.3.	Memory Planning	168
3.14.4.	Lock Contention and Concurrent Operations	170
3.14.5.	Troubleshooting Guide	171
3.15.	Sparse Embeddings with sparsevec	173
3.15.1.	Dense vs Sparse: Two Approaches to Meaning	173
3.15.2.	SPLADE: Learning Sparse Representations	174
3.15.3.	The sparsevec Type in pgvector	175
3.15.4.	Indexing and Querying Sparse Vectors	175
3.15.5.	When to Use Sparse Embeddings	176
3.16.	Advanced Quantization Patterns	178
3.16.1.	Expression-Based Indexes for Existing Tables	178
3.16.2.	The binary_quantize() Function	179
3.16.3.	Production Re-Ranking Pattern	179
3.16.4.	Memory Calculations for Product Catalogs	181
3.16.5.	Choosing the Right Quantization Level	182
3.16.6.	Quantization and Filtered Search	182
3.16.7.	Monitoring Quantization in Production	183
3.17.	Embedding Fine-Tuning for Domain Adaptation	184
3.17.1.	When Fine-Tuning Helps	184
3.17.2.	The Fine-Tuning Workflow	185
3.17.3.	Evaluating Fine-Tuning Impact	187
3.17.4.	RecSys Context: Product Catalog Fine-Tuning	188

3.17.5.	Practical Considerations	188
3.18.	pgvector vs Dedicated Vector Databases	189
3.18.1.	The Landscape: Extension vs Purpose-Built	189
3.18.2.	Feature Comparison	191
3.18.3.	Numerical Decision Thresholds	194
3.18.4.	Use-Case Decision Matrix	197
3.18.5.	Making the Decision: A Practical Framework	201
3.18.6.	Summary: Choosing Your Vector Search Solution	202
3.18.7.	The Operational Argument	202
3.18.8.	Benchmark Sources and Caveats	205
3.19.	When NOT to Use pgvector	206
3.19.1.	The Quick Decision Checklist	207
3.19.2.	The Future of pgvector	208
3.19.3.	Migration Path: pgvector to Dedicated	210
3.19.4.	A Note on Intellectual Honesty	210
	Further Reading	211
3.20.	Summary	213
3.21.	Exercises	214
3.21.1.	Exercise 3.1: Basic Vector Similarity Query	214
3.21.2.	Exercise 3.2: Compare Distance Functions	215
3.21.3.	Exercise 3.3: Hybrid Search with RRF	216
3.21.4.	Exercise 3.4: Index Tuning for Recall vs. Latency	217
3.21.5.	Exercise 3.5: Filtered Vector Search	218
4.	RAG Fundamentals	221
4.1.	Learning Objectives	221
4.2.	Prerequisites	222
4.3.	The RAG Pattern	222
4.3.1.	What Is Retrieval-Augmented Generation?	222
4.3.2.	RAG Architecture	223
4.3.3.	Why PostgreSQL for RAG?	224
4.3.4.	The RAG Data Flow	225
4.3.5.	Naive RAG in a Single SQL Query	225
4.3.6.	Limitations of Naive RAG	227
4.4.	The pgai Extension	228
4.4.1.	What pgai Provides	228
4.4.2.	Embedding from SQL	229
4.4.3.	Text Generation	230
4.4.4.	Chat Completion	231
4.4.5.	Complete In-Database RAG Function	233
4.4.6.	Trade-offs of In-Database RAG	235
4.5.	pgai Vectorizer: Automatic Embedding Sync	236
4.5.1.	Vectorizer as a Declarative Index	236
4.5.2.	Architecture: Extension + Worker	237
4.5.3.	Installation and Setup	238
4.5.4.	Creating a Vectorizer for Products	239
4.5.5.	Pipeline Flow	240
4.5.6.	Monitoring Vectorizer Status	240
4.5.7.	When to Use Vectorizer vs. Manual Embedding	241

4.6.	Chunking Strategies	242
4.6.1.	Fixed-Size Character Chunking	242
4.6.2.	Recursive Character Text Splitting	243
4.6.3.	Choosing Chunk Size	244
4.6.4.	Chunk Size vs. Retrieval Quality	245
4.6.5.	Formatting for Context Enrichment	246
4.6.6.	When NOT to Chunk	247
4.7.	Python RAG Pipeline	247
4.7.1.	psycopg (v3) Fundamentals for RAG	247
4.7.2.	Data Classes for Type Safety	248
4.7.3.	Step 1 – Retrieve	249
4.7.4.	Step 2 – Augment	250
4.7.5.	Step 3 – Generate	251
4.7.6.	Complete Pipeline	252
4.7.7.	Error Handling for LLM Calls	253
4.7.8.	When to Use Python vs Pure SQL	254
4.8.	Production RAG	255
4.8.1.	Semantic Caching	256
4.8.2.	Token Management	258
4.8.3.	Monitoring and Observability	259
4.8.4.	Error Handling Patterns	262
4.9.	When NOT to Use In-Database RAG	264
4.9.1.	When RAG Is the Wrong Tool	264
4.9.2.	Signs You’ve Outgrown Basic RAG	264
4.9.3.	The Complexity Trade-off	265
	Further Reading	266
4.10.	Summary	267
4.11.	Exercises	267
4.11.1.	Exercise 4.1: RAG Query Construction	268
4.11.2.	Exercise 4.2: Chunking Strategy Comparison	268
4.11.3.	Exercise 4.3: Build a Complete RAG Pipeline	269
4.11.4.	Exercise 4.4: Semantic Cache Effectiveness	270
5.	Advanced RAG Patterns	272
5.1.	Learning Objectives	272
5.2.	Prerequisites	273
5.3.	Framework Integration: LangChain and LlamaIndex	274
5.3.1.	LangChain + PostgreSQL	274
5.3.2.	LlamaIndex + PostgreSQL	278
5.3.3.	When to Use Frameworks vs. Raw SQL	282
5.3.4.	Running Example: Product Search Across Approaches	283
5.4.	Advanced RAG Patterns	285
5.4.1.	Cross-Encoder Re-ranking	285
5.4.2.	LLM-Based Re-ranking	287
5.4.3.	Re-ranking Comparison	288
5.4.4.	Multi-Step Retrieval	289
5.4.5.	Metadata Filtering	291
5.4.6.	Choosing the Right Pattern	293

5.5.	Agentic RAG	294
5.5.1.	Adaptive RAG: Routing by Query Complexity	294
5.5.2.	Corrective RAG: Scoring Retrieval Confidence	297
5.5.3.	Self-RAG: Reflection Tokens	299
5.5.4.	Comparing the Patterns	301
5.6.	Contextual Retrieval	302
5.6.1.	The Anthropic Approach	303
5.6.2.	PostgreSQL Implementation	303
5.6.3.	Generating Context with pgai	305
5.6.4.	Cost-Quality Tradeoff	306
5.7.	Late Chunking	308
5.7.1.	Traditional vs. Late Chunking	308
5.7.2.	Conceptual Pipeline	308
5.7.3.	PostgreSQL Storage	310
5.7.4.	Comparison with Contextual Retrieval	310
5.8.	Evaluating Your RAG Pipeline	312
5.8.1.	RAG Quality Metrics	312
5.8.2.	Quick Evaluation with RAGAS	313
5.8.3.	Production Evaluation with DeepEval	315
5.8.4.	Building a Golden Dataset in PostgreSQL	317
5.8.5.	CI/CD Integration	319
5.8.6.	PostgreSQL-Specific Considerations	320
5.9.	Knowledge Graph Traversal	321
5.9.1.	Modeling Product Relationships	321
5.9.2.	Recursive CTEs: Walking the Graph	322
5.9.3.	Cycle Prevention	323
5.9.4.	Combining Graph Traversal with Vector Search	324
5.9.5.	Performance and Practical Limits	326
5.10.	GraphRAG	327
5.10.1.	The Microsoft GraphRAG Pipeline	327
5.10.2.	PostgreSQL Schema for GraphRAG	329
5.10.3.	Querying Community Summaries	330
5.10.4.	Graph-Enhanced Retrieval vs GraphRAG	331
5.10.5.	Community Detection in PostgreSQL	333
5.10.6.	Generating Community Summaries	335
5.11.	Text-to-SQL	336
5.11.1.	Manual Approach: Schema Context + Few-Shot Prompting	337
5.11.2.	Safety Rails	339
5.11.3.	pgai Semantic Catalog	341
5.11.4.	Manual vs Semantic Catalog	343
5.11.5.	When NOT to Use Text-to-SQL	343
5.12.	Streaming LLM Responses	344
5.12.1.	Layer 1: Complete Responses with pgai	344
5.12.2.	Layer 2: Python Streaming	345
5.12.3.	Complete Runnable Example	347
5.12.4.	When Not to Stream	348
5.13.	Running Example: Product Q&A Chatbot	349
5.13.1.	What We're Building	349
5.13.2.	Architecture Overview	350

5.13.3.	Database Setup	350
5.13.4.	The Chatbot Code	351
5.13.5.	Sample Interaction	353
5.13.6.	Running the Chatbot	354
5.13.7.	What We've Built	355
5.14.	Multimodal RAG: Text and Image Retrieval	356
5.14.1.	Why Multimodal Matters for Product Search	356
5.14.2.	Image Embedding with CLIP	356
5.14.3.	PostgreSQL Schema for Multimodal Embeddings	358
5.14.4.	Populating Multimodal Embeddings	359
5.14.5.	Combined Search: Text + Image	360
5.14.6.	Python API for Multimodal Search	361
5.14.7.	Use Case: Visual Product Discovery	363
5.14.8.	Performance Considerations	365
5.15.	When Advanced Patterns Add Unnecessary Complexity	366
5.15.1.	When Basic RAG Is Enough	366
5.15.2.	Patterns and Their Complexity Costs	366
5.15.3.	The Incremental Adoption Path	367
	Further Reading	368
5.16.	Summary	370
5.17.	Exercises	371
5.17.1.	Exercise 5.1: RAG Evaluation Setup	371
5.17.2.	Exercise 5.2: Knowledge Graph Construction with CTEs	372
5.17.3.	Exercise 5.3: Text-to-SQL with Safety Guardrails	373
5.17.4.	Exercise 5.4: End-to-End Product Chatbot Enhancement	373
5.17.5.	Exercise 5.5: CRAG Retrieval Evaluator	374
5.17.6.	Exercise 5.6: Contextual Retrieval Comparison	375
5.17.7.	Exercise 5.7: GraphRAG Entity Extraction	376
5.17.8.	Exercise 5.8: Streaming RAG with Response Logging	376
6.	Feature Engineering and Feature Store	379
6.1.	Learning Objectives	379
6.2.	Prerequisites	380
6.3.	The Feature Problem	380
6.3.1.	The Leaky Query	381
6.3.2.	The Fix: Point-in-Time Retrieval	382
6.3.3.	Why LATERAL JOIN Works	384
6.3.4.	Three Problems, One Chapter	384
6.3.5.	Why PostgreSQL?	385
6.4.	Feature Store Concepts	387
6.4.1.	What Is a Feature Store?	387
6.4.2.	Online vs Offline Serving	387
6.4.3.	Point-in-Time Correctness	389
6.4.4.	PostgreSQL as a Feature Store Architecture	391
6.4.5.	What PostgreSQL Handles Well	391
6.4.6.	What PostgreSQL Does Not Handle Well	392
6.4.7.	Setting Up the Seed Data	392
6.5.	Window Functions for Feature Engineering	394
6.5.1.	Rolling Aggregations	394

6.5.2.	ROWS vs RANGE: The Silent Bug	396
6.5.3.	LAG/LEAD for Sequential Patterns	397
6.5.4.	Ranking and Recency	398
6.5.5.	Named Windows for Readability	400
6.5.6.	Combining Patterns: A Complete Feature Vector	402
6.6.	Materialized Views for Feature Caching	404
6.6.1.	Creating a Feature Materialized View	404
6.6.2.	Refresh Strategies	406
6.6.3.	Scheduling with pg_cron	407
6.6.4.	Staleness and Limitations	409
6.6.5.	Online Serving Pattern	410
6.7.	Continuous Aggregates for Time-Series Features	411
6.7.1.	Creating a Continuous Aggregate	412
6.7.2.	Backfilling Historical Data	413
6.7.3.	Refresh Policies	414
6.7.4.	Real-Time vs. Materialized-Only Mode	415
6.7.5.	Hierarchical Continuous Aggregates	416
6.7.6.	The Two-Tier Caching Strategy	417
6.7.7.	Comparison: Materialized Views vs. Continuous Aggregates	418
6.7.8.	Performance Considerations	420
6.8.	Feature Versioning and Registry	421
6.8.1.	The Feature Registry Table	421
6.8.2.	Registering Features	422
6.8.3.	Versioning Workflow	423
6.8.4.	Lightweight Lineage	424
6.8.5.	Feature Documentation as Practice	425
6.8.6.	Registry Maintenance	426
6.9.	Feature Store Anti-Patterns	427
6.9.1.	1. Point-in-Time Violation	427
6.9.2.	2. Training-Serving Skew	428
6.9.3.	3. Stale Feature Cache	429
6.9.4.	4. Feature-Target Leakage	430
6.9.5.	5. Schema Coupling and Naming Chaos	431
6.9.6.	6. Missing Lineage	432
6.9.7.	7. Over-Engineering: The Premature Feature Store	432
6.9.8.	Anti-Pattern Summary	433
6.10.	Putting It All Together: User Behavior Features	434
6.10.1.	The Dataset Recap	434
6.10.2.	Layer 1: Continuous Aggregates (Tier 1 Cache)	435
6.10.3.	Layer 2: Materialized View (Tier 2 Cache)	437
6.10.4.	Layer 3: Feature Registry Entries	439
6.10.5.	Layer 4: Scheduled Refresh	440
6.10.6.	Features by Archetype	441
6.10.7.	Online Serving Query	441
6.10.8.	Offline PIT Extraction	442
6.10.9.	The Complete Architecture	443
6.11.	When NOT to Use PostgreSQL as a Feature Store	444
6.11.1.	What PostgreSQL Gives You That Dedicated Stores Don't	444
6.11.2.	The Decision Framework	445

6.11.3.	Where PostgreSQL Breaks Down	445
6.11.4.	Feast and Tecton: What They Actually Provide	446
6.11.5.	Total Cost of Ownership	448
6.11.6.	Real-World Decision Scenarios	448
6.11.7.	The Migration Path	449
6.11.8.	The Bottom Line	451
	Further Reading	451
6.12.	Summary	452
6.13.	Exercises	453
6.13.1.	Exercise 6.1 — Create a Basic Feature View	453
6.13.2.	Exercise 6.2 — Build User Affinity Features with Window Functions	454
6.13.3.	Exercise 6.3 — Implement Continuous Aggregates with TimescaleDB	455
6.13.4.	Exercise 6.4 — Create Feature Refresh Pipeline with pg_cron	456
6.13.5.	Exercise 6.5 — Design Temporal Feature Engineering for RecSys	457
7.	In-Database Machine Learning	460
7.1.	Learning Objectives	460
7.2.	Prerequisites	461
7.3.	The Case for In-Database ML	461
7.3.1.	ML Concepts for the SQL Practitioner	462
7.3.2.	Data Gravity: Bring the Algorithm to the Data	463
7.3.3.	What PostgresML Provides	464
7.3.4.	The Landscape: Alternatives and Context	465
7.3.5.	What We'll Build	467
7.4.	Setting Up PostgresML	468
7.4.1.	Launching PostgresML	469
7.4.2.	Verifying the Extension	470
7.4.3.	Loading Product Sales Data	470
7.4.4.	Exploring the Dashboard	472
7.4.5.	What We Have So Far	472
7.5.	Training Models from SQL	473
7.5.1.	Your First Model: Predicting Demand	473
7.5.2.	Handling Categorical Features	475
7.5.3.	Comparing Algorithms	476
7.5.4.	Hyperparameter Tuning	478
7.5.5.	Training a Classification Model	479
7.5.6.	What You've Learned	482
7.6.	Making Predictions	483
7.6.1.	Single-Row Predictions	483
7.6.2.	The Feature Array Ordering Pitfall	483
7.6.3.	The Feature Helper View Pattern	484
7.6.4.	Batch Predictions with JOIN	485
7.6.5.	Integration Patterns	486
7.6.6.	Performance Considerations	489
7.6.7.	Choosing Between Real-Time and Batch Predictions	490
7.6.8.	What You've Learned	490
7.7.	PL/Python: The Escape Hatch	490
7.7.1.	Setting Up PL/Python	491
7.7.2.	Your First PL/Python Function	491

7.7.3.	The plpy Module	492
7.7.4.	Package Management Reality	494
7.7.5.	When to Use PL/Python vs. Alternatives	494
7.8.	Anomaly Detection on Product Data	496
7.8.1.	The Statistical Baseline: Z-Score Detection	496
7.8.2.	Building the Isolation Forest Function	497
7.8.3.	Running Anomaly Detection	499
7.8.4.	Interpreting Results	500
7.8.5.	Scheduling Anomaly Checks	501
7.8.6.	Trade-Offs: Unsupervised vs. Supervised Detection	502
7.8.7.	Production Considerations	503
7.9.	Model Versioning and A/B Testing	503
7.9.1.	Deployment Strategies with pgml.deploy	503
7.9.2.	Querying the Model Registry	505
7.9.3.	A/B Testing Models in SQL	506
7.9.4.	Retraining Strategies	510
7.10.	When NOT to Use In-Database ML	513
7.10.1.	When In-Database ML IS the Right Choice	514
7.10.2.	When In-Database ML Is NOT the Right Choice	515
7.10.3.	The Quick Decision Checklist	517
7.10.4.	The Decision Matrix	518
7.10.5.	Reading the Matrix	519
7.10.6.	The Migration Path Out	519
7.10.7.	Comparison with the Vector Search Decision	520
7.10.8.	The Honest Bottom Line	520
7.11.	Running Example: Demand Forecasting and Price Optimization	521
7.11.1.	The Business Scenario	521
7.11.2.	Preparing the Data	522
7.11.3.	Training the Demand Forecasting Model	524
7.11.4.	Comparing Algorithms	525
7.11.5.	Generating Demand Predictions	526
7.11.6.	Price Optimization: Finding Revenue-Maximizing Prices	527
7.11.7.	The Summary Dashboard	530
7.11.8.	Connecting to Prior Chapters	532
7.11.9.	What You've Built	533
7.11.10.	Training a Product Recommender Model	533
7.11.11.	Practice	535
	Further Reading	535
7.12.	Summary	536
7.13.	Exercises	537
7.13.1.	Exercise 7.1 — Train a Basic Regression Model with PostgresML	537
7.13.2.	Exercise 7.2 — Compare Model Algorithms	538
7.13.3.	Exercise 7.3 — Implement Batch Prediction Pipeline	540
7.13.4.	Exercise 7.4 — Build Recommendation Model with Collaborative Filtering Features	542
7.13.5.	Exercise 7.5 — Create Model Monitoring with Drift Detection	543

III. Production AI Systems	547
8. Real-Time AI Pipelines	550
8.1. Learning Objectives	550
8.2. Prerequisites	551
8.3. The Stale Recommendations Problem	552
8.3.1. The Staleness Timeline	552
8.3.2. Quantifying the Cost	553
8.3.3. The Batch Refresh Trap	554
8.3.4. Why This Is Hard	554
8.3.5. What We'll Build	555
8.4. LISTEN/NOTIFY: Simple Event Signaling	556
8.4.1. The Basic Pattern	556
8.4.2. Trigger-Based Notification	557
8.4.3. Python Listener with psycopg (v3)	558
8.4.4. AI Use Case: Real-Time Re-Embedding	560
8.4.5. A Real Production Incident	561
8.4.6. Three More Limitations	562
8.4.7. When to Use LISTEN/NOTIFY — A Decision Framework	562
8.5. SKIP LOCKED: PostgreSQL as a Job Queue	564
8.5.1. How SKIP LOCKED Works	565
8.5.2. The Job Queue Schema	565
8.5.3. The Worker Claim Pattern	566
8.5.4. Complete Python Worker	567
8.5.5. Dead Letter Handling	571
8.5.6. Scaling Workers	572
8.5.7. Enqueuing AI Jobs	572
8.5.8. Limitations and Graduation Criteria	573
8.6. pg_cron Orchestration	574
8.6.1. Scheduling AI Pipeline Steps	575
8.6.2. Job Chaining via Status Checks	576
8.6.3. Monitoring with cron.job_run_details	577
8.6.4. The Cross-Container Problem	578
8.6.5. Practical Schedule Template	579
8.6.6. When pg_cron Isn't Enough	580
8.7. Change Data Capture with Debezium	581
8.7.1. Why CDC Over Triggers?	582
8.7.2. Architecture Overview	583
8.7.3. Starting the CDC Stack	583
8.7.4. Kafka Essentials (Just Enough)	583
8.7.5. Registering the Debezium Connector	584
8.7.6. Understanding CDC Events	585
8.7.7. Python Kafka Consumer	587
8.7.8. Reading Events from the Command Line	589
8.7.9. WAL Management	589
8.7.10. Performance Characteristics	590
8.7.11. Connector Lifecycle	591
8.7.12. When to Use Debezium CDC	592

8.8.	Real-Time Recommendations: Putting It All Together	593
8.8.1.	The Recommendations Table	593
8.8.2.	Approach 1: Pure PostgreSQL Pipeline	594
8.8.3.	Approach 2: CDC Pipeline (Debezium + Kafka)	597
8.8.4.	Comparing the Two Approaches	598
8.8.5.	Freshness Improvement	599
8.8.6.	The Lambda Pattern in PostgreSQL	599
8.9.	Choosing Your Pattern — and When NOT to Build Real-Time	600
8.9.1.	Running Example: End-to-End Walkthrough	600
8.9.2.	Decision Framework	602
8.9.3.	When NOT to Use Real-Time Pipelines	603
8.9.4.	Complete Decision Table	604
8.10.	Model Context Protocol (MCP) and PostgreSQL	606
8.11.	AI Agent Frameworks and PostgreSQL	608
8.11.1.	LangGraph and CrewAI	608
8.11.2.	Framework Comparison	609
	Further Reading	610
8.12.	Summary	611
8.13.	Exercises	612
8.13.1.	Exercise 8.1 — LISTEN/NOTIFY for Product Updates	612
8.13.2.	Exercise 8.2 — SKIP LOCKED Queue for Embedding Jobs	612
8.13.3.	Exercise 8.3 — pg_cron Job for Cache Maintenance	613
8.13.4.	Exercise 8.4 — MCP Server for RecSys	613
8.13.5.	Exercise 8.5 — LangGraph Workflow with PostgreSQL State	614
9.	AI Platform Architecture with PostgreSQL	616
9.1.	Learning Objectives	616
9.2.	Prerequisites	617
9.3.	PostgreSQL’s Role in AI Architectures	618
9.3.1.	What PostgreSQL Does Brilliantly	619
9.3.2.	What PostgreSQL Is Not	620
9.3.3.	The Gravitational Pull Pattern	621
9.3.4.	Mapping Prior Chapters to Architecture	622
9.3.5.	A Note on the “PostgreSQL Can’t Scale” Myth	624
9.4.	Multi-Service AI Architecture	625
9.4.1.	Responsibility Matrix	625
9.4.2.	Pattern 1: PostgreSQL + Inference Sidecar	626
9.4.3.	Pattern 2: PostgreSQL + Feature Layer + Model Server	627
9.4.4.	Pattern 3: Full AI Platform	629
9.4.5.	Anti-Pattern: PostgreSQL as Message Bus	631
9.4.6.	The Graduation Path	632
9.5.	Event Sourcing for AI Systems	632
9.5.1.	When Event Sourcing Earns Its Complexity	633
9.5.2.	The Event Store Schema	633
9.5.3.	Projection Patterns: Deriving Current State	634
9.5.4.	Snapshot Pattern: Bounded Replay	635
9.5.5.	Connecting to the CDC Pipeline	636
9.5.6.	Trade-offs	637

9.6.	Agentic AI: PostgreSQL as Agent Memory and Tools	638
9.6.1.	Agent Memory Schema	638
9.6.2.	Tool Registry: PostgreSQL Functions as Agent Tools	640
9.6.3.	Implementing PG Functions as Tools	642
9.6.4.	Agent Orchestration: A Complete Turn	644
9.6.5.	Framework-Agnostic Design	646
9.6.6.	Scaling Considerations	647
9.6.7.	Looking Forward	647
9.7.	Model Context Protocol: PostgreSQL as AI Agent Tool Provider	648
9.7.1.	The Tool Discovery Problem	648
9.7.2.	MCP Protocol Essentials	649
9.7.3.	PostgreSQL as an MCP Tool Provider	650
9.7.4.	Practical Example: AI Agent Queries a Product Catalog	651
9.7.5.	Security and Access Control	653
9.7.6.	The Broader MCP Ecosystem	654
9.7.7.	When NOT to Use MCP	655
9.8.	Multi-Tenant AI Architecture	655
9.8.1.	The tenant_id Pattern	656
9.8.2.	RLS Policies for AI Tables	657
9.8.3.	Embedding Isolation and Index Strategy	659
9.8.4.	Performance Considerations	660
9.8.5.	What This Pattern Gives You	661
9.9.	Data Pipeline Architecture	661
9.9.1.	The ML Lifecycle Loop	662
9.9.2.	Batch Pipeline: pg_cron Orchestration	662
9.9.3.	Real-Time Pipeline: CDC to Stream Processing	664
9.9.4.	Feature Store Integration	665
9.9.5.	PG as Both Source and Sink	665
9.9.6.	Anti-Pattern: ETL Spaghetti	666
9.10.	Putting It All Together: The AI Recommendation Platform	667
9.10.1.	The Complete Architecture	667
9.10.2.	Tracing a Request Through the Architecture	667
9.10.3.	Architecture Decision Record Template	669
9.10.4.	When NOT to Use This Architecture	670
9.10.5.	Scaling the Architecture	672
	Further Reading	672
9.11.	Summary	673
9.12.	Exercises	674
9.12.1.	Exercise 9.1: Document an Architecture Decision Record	675
9.12.2.	Exercise 9.2: Implement Multi-Tenant RLS for RecSys	675
9.12.3.	Exercise 9.3: Design a Hybrid Architecture Diagram	676
9.12.4.	Exercise 9.4: Build an MCP Tool Registry	676
9.12.5.	Exercise 9.5: Implement Connection Routing for Read Replicas	677
10.	Performance Optimization for AI Workloads	679
10.1.	Learning Objectives	679
10.2.	Prerequisites	680
10.3.	Tuning PostgreSQL for AI Workloads	680
10.3.1.	Core Memory Parameters	681

10.3.2.	Parallelism Settings	683
10.3.3.	PG17 and PG18 Performance Improvements	684
10.3.4.	Three Server Profiles	685
10.3.5.	Query-Time Tuning Knobs	686
10.3.6.	Applying Configuration Changes	687
10.3.7.	Verifying Your Configuration	688
10.4.	Benchmarking Vector Search	688
10.4.1.	What to Measure	688
10.4.2.	How to Measure Correctly	690
10.4.3.	Common Mistakes	691
10.4.4.	A Reproducible Benchmark Pattern	692
10.4.5.	Dataset Sizing Guidance	693
10.4.6.	Recording and Reporting Results	694
10.4.7.	From Methodology to Practice	694
10.5.	Query Optimization for AI Queries	695
10.5.1.	EXPLAIN ANALYZE for AI Workloads	695
10.5.2.	Vector Search Query Plans	696
10.5.3.	Feature Query Optimization	700
10.5.4.	RAG Pipeline Query Optimization	701
10.5.5.	Optimization Checklist	702
10.6.	Connection Pooling for AI Workloads	704
10.6.1.	PgBouncer: The Standard Choice	704
10.6.2.	PgCat: The Modern Alternative	706
10.6.3.	PgBouncer vs PgCat	708
10.6.4.	Pool Sizing Guidelines	708
10.6.5.	Docker Setup Pattern	710
10.7.	Partitioning Large Embedding Tables	710
10.7.1.	Three Partitioning Strategies	711
10.7.2.	HNSW Indexes Per Partition	713
10.7.3.	Sizing Guidance	714
10.7.4.	Migrating an Existing Table	715
10.7.5.	Choosing a Strategy	716
10.7.6.	Partition Maintenance	716
10.7.7.	When NOT to Partition	717
10.8.	Read Replicas for AI Inference	718
10.8.1.	Streaming Replication Basics	718
10.8.2.	AI-Specific Routing Rules	718
10.8.3.	Connection Pooler Integration	719
10.8.4.	Replication Lag Monitoring	720
10.8.5.	Scaling Patterns	721
10.8.6.	Limitations	722
10.8.7.	When NOT to Use Replicas	722
10.9.	Real Benchmarks: pgvector vs pgvector scale vs VectorChord	723
10.9.1.	The Contenders	723
10.9.2.	pgvector 0.8.0: The Baseline	724
10.9.3.	pgvector scale: Disk-Friendly Scaling	724
10.9.4.	VectorChord 1.0: The Emerging Contender	725
10.9.5.	The Comparison Table	726
10.9.6.	Decision Framework	727

10.9.7.	Running Your Own Comparison	729
10.9.8.	What the Benchmarks Don't Tell You	729
10.9.9.	Extension Installation Reference	730
10.9.10.	Summary: The Landscape in Early 2026	731
10.10.	Running Example: Scaling to 1M+ Products	731
10.10.1.	Step 0: The Baseline	732
10.10.2.	Step 1: Apply the Medium Tuning Profile	732
10.10.3.	Step 2: Optimize Queries with EXPLAIN ANALYZE	733
10.10.4.	Step 3: Partition Embeddings	734
10.10.5.	Step 4: PgBouncer Connection Pooling	735
10.10.6.	Step 5: Read Replica Routing	736
10.10.7.	Results Summary	737
10.10.8.	Scaling Projections	737
10.10.9.	When NOT to Optimize	738
	Further Reading	738
10.11.	Summary	739
10.12.	Exercises	741
10.12.1.	Exercise 10.1: Configure Memory Settings for Vector Workloads	741
10.12.2.	Exercise 10.2: Analyze EXPLAIN Output for Vector Queries	741
10.12.3.	Exercise 10.3: Tune HNSW Index Parameters	742
10.12.4.	Exercise 10.4: Design Connection Pooling for RecSys	743
10.12.5.	Exercise 10.5: Implement Hash Partitioning for Large Vector Tables	743

IV. Operations & Beyond

745

11. Security, Privacy, and Governance

748

11.1.	Learning Objectives	749
11.2.	Prerequisites	749
11.3.	The AI Threat Landscape	750
11.3.1.	Defense-in-Depth for AI Systems	750
11.3.2.	AI-Specific Threat Taxonomy	750
11.3.3.	The Maturity Gap	753
11.4.	RLS Hardening for AI Workloads	753
11.4.1.	FORCE ROW LEVEL SECURITY – The Owner Bypass	754
11.4.2.	RLS with Vector Similarity Search	754
11.4.3.	Materialized Views Bypass RLS	755
11.4.4.	RLS Testing Scripts	756
11.4.5.	Leak Detection: Finding Unprotected Tables	757
11.4.6.	Performance Trade-offs	758
11.5.	Encrypting Sensitive AI Data	758
11.5.1.	The Pattern: Encrypt Text, Search Vectors	759
11.5.2.	PGP Symmetric Encryption Walkthrough	759
11.5.3.	Key Management: The Hard Part	760
11.5.4.	Pitfall: Encrypted Data in Logs	762
11.5.5.	Trade-Offs	762
11.6.	Differential Privacy in SQL	763
11.6.1.	The Laplace Mechanism	763
11.6.2.	Epsilon Trade-Offs	764

11.6.3.	k-Anonymity as a Guardrail	765
11.6.4.	Anti-Pattern: Differential Privacy Theater	767
11.6.5.	Production Alternatives	767
11.7.	GDPR Compliance for AI Data	767
11.7.1.	Embedding Lineage Tracking	768
11.7.2.	The Erasure Function	769
11.7.3.	Consent Tracking	772
11.7.4.	Data Lineage Beyond Embeddings	773
11.7.5.	Regulatory Context: Beyond GDPR	773
11.8.	Audit Logging for AI Operations	774
11.8.1.	pgaudit: Compliance-Grade Database Logging	774
11.8.2.	Custom AI Audit Table	775
11.8.3.	The Combined Approach	777
11.8.4.	Querying the Audit Trail	778
11.9.	AI-Specific Security Threats	779
11.9.1.	Embedding Inversion: Your Vectors Are Not Hashes	779
11.9.2.	Prompt Injection Beyond Text-to-SQL	781
11.9.3.	Data Poisoning	782
11.9.4.	Model Extraction	784
11.10.	Securing RecSys	785
11.10.1.	Layer 1: Access Control with Row-Level Security	785
11.10.2.	Layer 2: Encryption for PII	786
11.10.3.	Layer 3: Differential Privacy for Analytics	787
11.10.4.	Layer 4: Compliance and Data Lineage	788
11.10.5.	Layer 5: Audit Logging	790
11.10.6.	Layer 6: AI-Specific Defenses	791
11.10.7.	Verification Walkthrough	793
11.10.8.	What This Doesn't Cover	795
11.11.	When NOT to Use In-Database Security	795
11.11.1.	1. Network-Level Attacks	795
11.11.2.	2. Application-Layer Authentication	796
11.11.3.	3. Key Management at Scale	796
11.11.4.	4. Advanced Differential Privacy	797
11.11.5.	5. Regulatory Compliance Beyond Technical Controls	798
11.11.6.	6. Multi-Database and Distributed Architectures	798
11.11.7.	Decision Framework	799
11.12.	OWASP LLM Top 10	800
11.12.1.	The 2025 Risk Landscape	800
11.12.2.	PostgreSQL Mitigations Summary	808
11.12.3.	RecSys Example: Securing the Recommendation Pipeline	809
11.13.	LLM Guardrails	811
11.13.1.	What Are LLM Guardrails?	811
11.13.2.	NeMo Guardrails	811
11.13.3.	Guardrails AI	814
11.13.4.	PostgreSQL Integration: Audit and Analytics	816
11.13.5.	When to Use Guardrails vs Database Security	817
11.13.6.	Production Considerations	818
11.13.7.	Resources	819

11.14. EU AI Act Compliance	820
11.14.1. Risk-Based Classification	820
11.14.2. Key Requirements for AI Applications	821
11.14.3. PostgreSQL Compliance Patterns	821
11.14.4. Implementation Example: Decision Logging	821
11.14.5. Right to Explanation	823
11.14.6. Human Oversight Workflow	824
11.14.7. Data Governance with Schema Versioning	825
11.14.8. Compliance Checklist	826
11.14.9. Security Checklist for AI-Powered PostgreSQL Systems	827
Further Reading	829
11.15. Summary	830
11.15.1. What's Next	831
11.16. Exercises	831
11.16.1. Exercise 11.1 — Implement Basic RLS Policies	831
11.16.2. Exercise 11.2 — Create Audit Logging Triggers	832
11.16.3. Exercise 11.3 — Encrypt PII with pgcrypto	832
11.16.4. Exercise 11.4 — Build Tenant Isolation Bypass Tests	833
11.16.5. Exercise 11.5 — Implement GDPR Data Erasure with Cascade	834
12. Production Deployment and Operations	836
12.1. Learning Objectives	837
12.2. Prerequisites	838
12.3. Cloud Deployment Options	838
12.3.1. Amazon RDS for PostgreSQL	838
12.3.2. Amazon Aurora PostgreSQL	839
12.3.3. Google AlloyDB	839
12.3.4. Neon	840
12.3.5. Supabase	840
12.3.6. Self-Hosted PostgreSQL	841
12.3.7. Decision Tree: Choosing Your Deployment	842
12.4. Extension Compatibility Matrix	843
12.4.1. The Matrix	843
12.4.2. Reading the Matrix	844
12.4.3. Annotated Notes	844
12.4.4. Cross-Reference: Where Each Extension Was Introduced	847
12.5. Monitoring and Observability	848
12.5.1. Standard PG Monitoring: The Foundation	848
12.5.2. Five AI Metrics Standard Monitoring Misses	848
12.5.3. Recall Quality: The Hardest Metric Done Simply	849
12.5.4. The ai_metrics View	849
12.5.5. Prometheus + postgres_exporter Setup	851
12.5.6. Custom Metrics Export	852
12.5.7. Alerting Rules	853
12.5.8. Managed Provider Monitoring	855
12.5.9. Trade-Offs	855
12.6. Backup Considerations for AI Workloads	856
12.7. High Availability	857
12.7.1. Managed HA	858

12.7.2.	Self-Hosted HA with Patroni	858
12.7.3.	Failover Impact on AI Workloads	860
12.7.4.	Decision Framework: Managed vs. Self-Hosted HA	861
12.8.	CI/CD for AI Database Systems	862
12.8.1.	What Makes AI Database CI/CD Different	862
12.8.2.	Migration Tools	863
12.8.3.	CI/CD Pipeline	863
12.8.4.	Blue-Green Database Deployments	865
12.8.5.	Rollback Considerations for AI Workloads	865
12.9.	PG-Centric MLOps	866
12.9.1.	Building a Model Registry in PostgreSQL	866
12.9.2.	Promotion Workflow	867
12.9.3.	Scheduled Retraining with pg_cron	868
12.9.4.	Embedding Refresh Tracking	869
12.9.5.	Signals to Graduate	870
12.10.	External MLOps Integration	871
12.10.1.	MLflow with a PostgreSQL Backend	871
12.10.2.	Airflow for Pipeline Orchestration	872
12.10.3.	dbt for Feature Transforms	873
12.10.4.	The Integration Pattern	874
12.10.5.	Honest Assessment	875
12.10.6.	Framework Deployment Patterns	875
12.11.	LLM Observability Tools	876
12.11.1.	Why LLM Observability Matters	876
12.11.2.	Tool Comparison	876
12.11.3.	Langfuse Integration	877
12.11.4.	PostgreSQL-Native Observability	880
12.11.5.	Connecting Traces to PostgreSQL Data	882
12.12.	AI Quality Monitoring	883
12.12.1.	RAG Retrieval Quality Metrics	883
12.12.2.	Embedding Drift Detection	886
12.12.3.	Prediction Distribution Monitoring	888
12.12.4.	RecSys Quality Dashboard	889
12.13.	Cost Analysis Framework	891
12.13.1.	Five Cost Dimensions	891
12.13.2.	Cost Calculator Template	892
12.13.3.	When Cloud AI Services Win	893
12.13.4.	When PostgreSQL-Based Wins	894
12.13.5.	Scale-Based Scenarios	894
12.13.6.	Making the Decision	895
12.14.	Running Example: Deploying RecSys	896
12.14.1.	Architecture Recap	896
12.14.2.	1. Provider Selection	897
12.14.3.	2. Monitoring Setup	898
12.14.4.	3. Backup Strategy	898
12.14.5.	4. High Availability Configuration	899
12.14.6.	5. CI/CD Pipeline	899
12.14.7.	6. Cost Estimation	899

12.15. Zero-Downtime Deployments	901
12.15.1. Why Zero-Downtime Matters for AI Applications	901
12.15.2. Logical Replication for Blue-Green Databases	901
12.15.3. Embedding Model Version Migration	902
12.15.4. CREATE INDEX CONCURRENTLY for Non-Blocking Rebuilds	904
12.15.5. Index Swap Strategy	905
12.15.6. Cutover Checklist	906
12.16. Cost Optimization for AI Workloads	906
12.16.1. Understanding AI Cost Drivers	906
12.16.2. Embedding Caching to Avoid Recomputation	907
12.16.3. Batch Processing for API Cost Efficiency	909
12.16.4. Local vs. Cloud Embedding Trade-offs	911
12.16.5. Storage Optimization Techniques	912
12.16.6. Cost Monitoring Dashboard	914
12.17. Disaster Recovery for Vector Data	915
12.17.1. Backup Strategies for Vector Tables	915
12.17.2. Embedding Regeneration Strategy	917
12.17.3. Index Recovery Considerations	918
12.17.4. RTO/RPO Planning for AI Systems	919
12.17.5. Recovery Drills	920
12.17.6. Backup Validation	921
12.18. Multi-Region Patterns	922
12.18.1. Why Multi-Region for AI Applications	922
12.18.2. Read Replicas for Vector Search Scaling	923
12.18.3. Citus for Distributed Vector Storage	925
12.18.4. Patroni for High-Availability Orchestration	927
12.18.5. Multi-Region Embedding Generation	929
12.18.6. Deployment Topology Summary	930
12.19. Production Case Studies	930
12.19.1. Case Study 1: E-Commerce Product Discovery	931
12.19.2. Case Study 2: News Recommendation Platform	934
12.19.3. Case Study 3: Fintech Document Intelligence	937
12.19.4. Common Patterns Across Case Studies	940
12.20. When NOT to Use PostgreSQL for AI in Production	941
12.20.1. Scenario 1: Sub-Millisecond Vector Search at 100M+ Scale	941
12.20.2. Scenario 2: GPU-Heavy Model Serving	941
12.20.3. Scenario 3: Complex Multi-Framework ML Pipelines	942
12.20.4. Scenario 4: Ephemeral and Serverless AI Workloads	942
12.20.5. Scenario 5: Teams Without PostgreSQL Expertise	942
12.20.6. Decision Framework	943
12.20.7. Production Readiness Checklist	944
Further Reading	945
12.21. Summary	946
12.21.1. Looking Ahead	947
12.22. Exercises	948
12.22.1. Exercise 12.1 — Verify Extension Compatibility Matrix	948
12.22.2. Exercise 12.2 — Set Up Embedding Freshness Monitoring	948
12.22.3. Exercise 12.3 — Configure Backup for Vector Data	949

12.22.4. Exercise 12.4 – Implement Blue-Green Deployment for Embedding Model Upgrade	950
12.22.5. Exercise 12.5 – Design Multi-Region Failover Strategy	950
13. The Future of PostgreSQL and AI	953
13.1. Learning Objectives	953
13.2. Prerequisites	953
13.3. Signal vs. Noise	954
13.4. PostgreSQL 18: What Matters for AI Workloads	955
13.4.1. Asynchronous I/O Subsystem	955
13.4.2. Virtual Generated Columns	956
13.4.3. Parallel GIN Index Builds	957
13.4.4. UUIDv7 – Time-Ordered Identifiers	957
13.4.5. OAuth 2.0 Authentication	958
13.4.6. pg_upgrade Retains Optimizer Statistics	958
13.4.7. Skip Scan for Multicolumn B-tree Indexes	959
13.4.8. Data Checksums by Default	959
13.4.9. The Bigger Picture	959
13.5. pgvector: Where It Is and Where It’s Going	960
13.5.1. Current State: pgvector 0.8.x	960
13.5.2. What’s Being Discussed	961
13.5.3. What’s Speculative	962
13.5.4. The Conservative Release Cadence Is a Feature	963
13.5.5. Planning Around pgvector’s Trajectory	963
13.6. The Emerging Extension Ecosystem	964
13.6.1. EXISTS TODAY (Young but Usable)	964
13.6.2. IN DEVELOPMENT	966
13.6.3. SPECULATIVE	967
13.6.4. Evaluating New Extensions	968
13.7. Multi-Modal Embeddings: Beyond Text	969
13.8. Your Next Steps	973
13.8.1. This Week	973
13.8.2. This Month	974
13.8.3. This Quarter	974
13.8.4. Building Your Knowledge Network	975
13.8.5. Staying Current	975
13.9. Honest Assessment	976
13.9.1. When NOT to Bet on PostgreSQL for AI	976
13.9.2. What PostgreSQL Actually Gives You	977
13.9.3. The Consolidation Math	978
13.9.4. The Database You Already Know	979
13.9.5. Learning Objectives Check	979
Further Reading	980
13.10. Summary	980
13.10.1. What You’ve Built	981
13.10.2. The RecSys Journey	982
13.10.3. The Path Forward	982
13.11. Exercises	982
13.11.1. Exercise 13.1 – Research PostgreSQL 18 AI Features	983

13.11.2.	Exercise 13.2 – Implement Multimodal Search Schema	983
13.11.3.	Exercise 13.3 – Design Edge Embedding Sync Pattern	984
13.11.4.	Exercise 13.4 – Build Cross-Modal Search (Text-to-Image)	984
13.11.5.	Exercise 13.5 – Create Upgrade Plan for pgvector Major Version	985

Appendices

988

A. Bonus Project: Ask the Book

988

A.1.	Learning Objectives	988
A.2.	Prerequisites	988
A.3.	Why RAG Over a Book?	989
A.3.1.	The Full-Circle Moment	990
A.3.2.	What Makes This Different from Chapter 4?	990
A.4.	Architecture Overview	991
A.4.1.	Ingestion Pipeline	991
A.4.2.	Query Pipeline	992
A.4.3.	Schema	993
A.4.4.	Module Dependency Graph	993
A.5.	Design Decisions	994
A.5.1.	PDF Input, Not Source Files	994
A.5.2.	Font-Based Structure Detection	994
A.5.3.	Heading-Aware Chunking	995
A.5.4.	Hybrid Search with RRF	996
A.5.5.	Cross-Encoder Re-ranking as Optional	996
A.5.6.	Streaming Generation	997
A.5.7.	Local-Only Evaluation	997
A.5.8.	Why Not LangChain or LlamaIndex?	998
A.6.	Code Walkthrough	998
A.6.1.	Setting Up the Database	998
A.6.2.	Parsing a PDF	999
A.6.3.	Chunking for Retrieval Quality	1000
A.6.4.	Embedding and Ingestion	1002
A.6.5.	Hybrid Retrieval	1002
A.6.6.	Streaming Generation	1004
A.6.7.	The CLI Layer	1005
A.7.	Evaluation	1006
A.7.1.	The Test Set	1006
A.7.2.	The Four Metrics	1007
A.7.3.	Running the Evaluation	1008
A.7.4.	Interpreting Results Honestly	1009
A.7.5.	What the Scores Mean for the Pipeline	1010
A.8.	Exercises	1011
A.8.1.	Exercise 1: Tune Chunk Size and Measure Impact	1011
A.8.2.	Exercise 2: Add a Different Embedding Model	1011
A.8.3.	Exercise 3: Implement Agentic Retrieval	1011
A.8.4.	Exercise 4: Add GraphRAG for Cross-Chapter Questions	1012
A.8.5.	Exercise 5: Build a Quality Dashboard	1012
A.8.6.	Exercise 6: Support EPUB Input	1012

A.9. Feature-to-Chapter Mapping	1013
A.9.1. Reading the Map	1014
A.10. Summary	1015
Further Reading	1016
B. References	1020
C. Glossary	1030
D. Index	1040

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](https://github.com).

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.

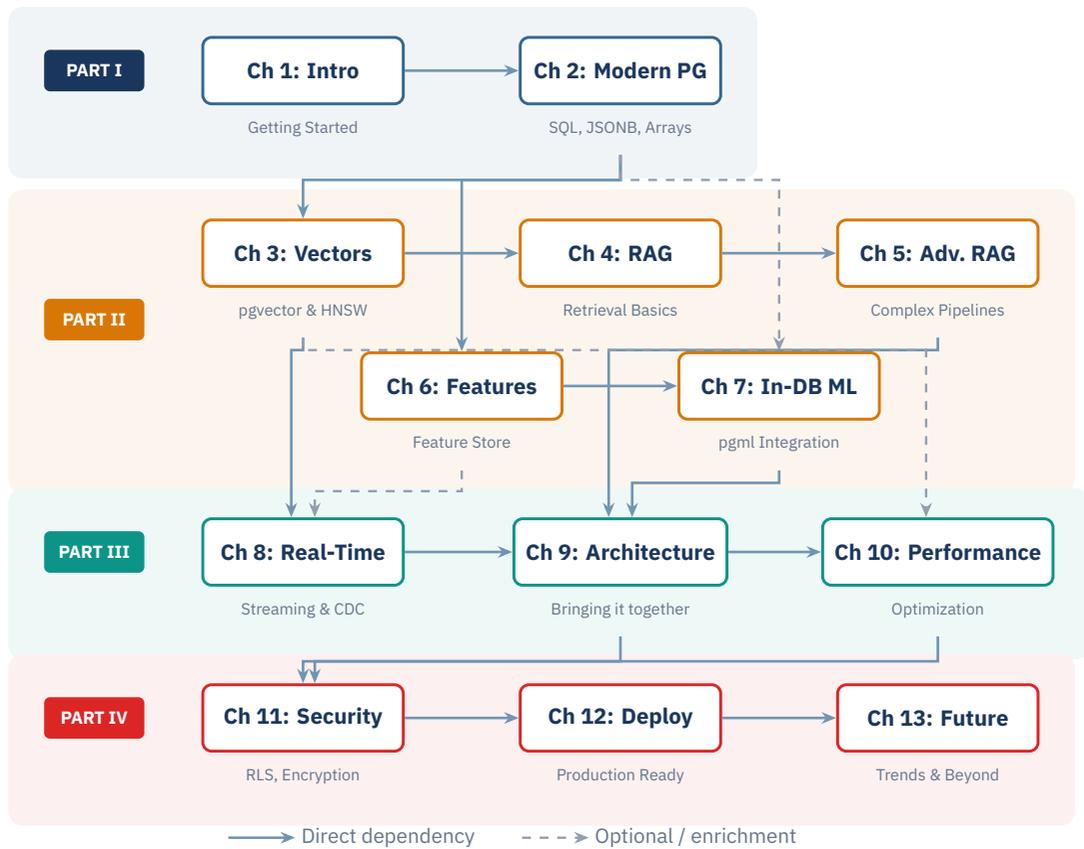


Figure 1.: Chapter dependency graph organized by book parts. Solid arrows show direct dependencies; dashed arrows show optional enrichment paths.

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

i 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]. `pgvector` 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]
- **pgvector scale** – 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 `pgvector scale` 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, pgvector_scale, 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

```
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

```
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

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

Option 2: Use the host network

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

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

1.6.2. Step 2: Connect

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

```

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

```

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)

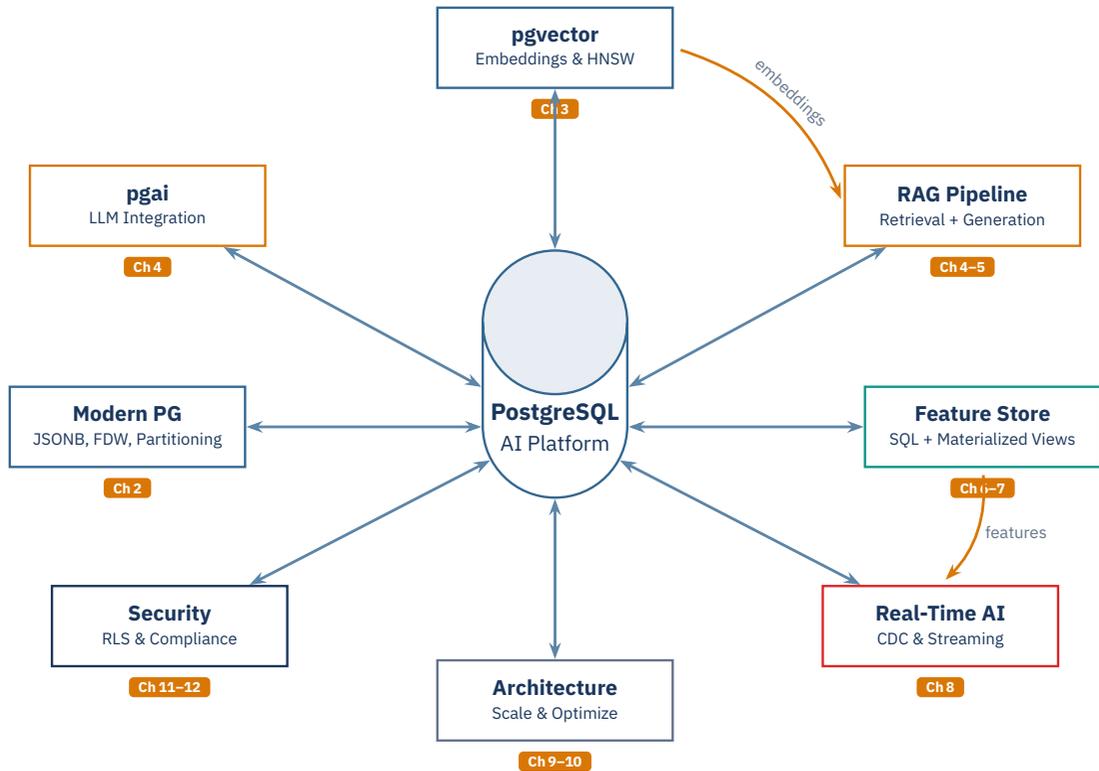


Figure 1.1.: RecSys system architecture: PostgreSQL at the center with surrounding AI capabilities and their chapter references.

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, pgvector scale, 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.

i 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 pgvector scale 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

```
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

```
-- 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

```
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

```
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

```
-- 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).