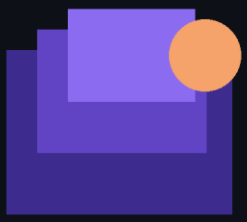

THE COMPLETE GUIDE TO
EF CORE ON POSTGRESQL

The Entity Framework Core & PostgreSQL Handbook

Mastering EF Core 10 with PostgreSQL 18 for Production-Grade Applications

```
// Semantic search, from LINQ, in the database you already run
var hits = await db.Documents
    .Where(d => d.Embedding != null)
    .OrderBy(d => d.Embedding!.CosineDistance(query))
    .Take(5)
    .ToListAsync();

// ORDER BY s.embedding <=> $1 LIMIT 5

// The column, and the index that makes it fast
builder.Property(d => d.Embedding)
    .HasColumnType("vector(384)");

builder.HasIndex(d => d.Embedding)
    .HasMethod("hnsw")
    .HasOperators("vector_cosine_ops");
```

RACHID DAHIR

EF CORE 10 · POSTGRESQL 18

The Entity Framework Core & PostgreSQL Handbook

Mastering EF Core 10 with PostgreSQL 18

for Production-Grade Applications

FIRST EDITION — 2026

Rachid Dahir

AviSoft

Technical Publishing

FIRST EDITION — 2026

Produced against .NET 10 with C# 14, EF Core 10.0.11, the Npgsql provider 10.0.3, and PostgreSQL 18.6 with PostGIS 3.6.4 and pgvector 0.8.6 in a digest-pinned container image. Later patch releases of any of these may change the output of a listing; the companion repository records the exact versions every printed result was captured from.

avisoft.ma

Preface

Who This Book Is For

This book is written for .NET developers who already use Entity Framework Core and have started to feel where it ends. You can model an entity, write a LINQ query, and add a migration. Your application works. What stays opaque is the other half of the system: which SQL your query became, why the same LINQ is fast against one table and slow against its neighbour, what your provider does with a JSON column or a UUID key, and which of PostgreSQL's capabilities you are already paying to run and never using. If that describes you, this book was written for you.

Some EF Core experience is assumed, and not much. Comfort with C# and with reading SQL is enough to start; everything past a basic DbContext is built on the page, in order, against one running application. This is not an introduction to EF Core, and it does not repeat one. Chapter 2 opens at pooled contexts, compiled models and configuration that scales past a single file, because the introductory ground is covered well elsewhere and covering it again would cost the pages this book spends on PostgreSQL.

Readers arriving from SQL Server get a specific welcome. The habits that worked there mostly transfer, and the dozen places where they actively mislead are flagged where they arrive rather than collected in a footnote. Appendix D is the full translation guide, written after the other twenty chapters so that it could be honest about what the book had actually run.

You may also be the person who ends up operating what the team ships. Part IV was written with that pager in mind: interceptors and diagnostics, an audit trail that survives contact with real writes, tenant isolation enforced in two layers and tested for agreement, integration tests against a real server, deployments that survive a migration mid-flight, and the outbox and change-data-capture patterns that let other systems trust yours.

What This Book Covers

Twenty-one chapters in four parts, followed by five appendices. Every part works the same running application, a multi-tenant project and field-service platform called LuminaWork, built as a modular monolith with six bounded contexts that own six PostgreSQL schemas between them. Nothing here is a schema invented for one page and abandoned on the next: the table Chapter 3 maps is the table Chapter 7 indexes, Chapter 11 stores JSON in, Chapter 18 locks down with row-level security, and Chapter 21 publishes events from. The Introduction walks the four parts chapter by chapter.

Get the Most Out of This Book

This is a book meant to be read with a keyboard within reach. Chapter 1 gets a container and a solution running in minutes, and from then on the fastest way through any chapter is to run what you read. Four production decisions make that worth your time:

- Every C# listing is an excerpt of code that compiles, taken from the companion solution rather than typed into the manuscript. A tool diffs each printed excerpt against the repository before the book will build, so a listing that drifted from the code it claims to show fails the build instead of reaching you.
- Every SQL listing runs. The book generates each one into the companion repository, and an automated harness executes all of them against a live PostgreSQL 18.6 database, including the listings that are supposed to fail: those are asserted to fail with exactly the error code the prose predicts.
- Every output panel was captured from a real run. Query results, execution plans and error messages are copied from the terminal into the page by a capture tool, never retyped. The same holds for the benchmark tables, which are rendered from committed BenchmarkDotNet artifacts rather than from a person reading a console.
- The database is deterministic on purpose. The seeder uses no random number generator at all, timestamps derive from one fixed anchor, and the container image disables just-in-time compilation so that plans stay stable. When you run a listing you get the printed output: not output that may vary, but the same rows in the same order, and the same plan shape, on your machine.

One script holds it together. `scripts/reset-to-chapter.ps1 -Chapter N` on Windows, or `scripts/reset-to-chapter.sh N` on macOS and Linux, rebuilds the database to the exact state chapter N expects to find when it begins. Read from anywhere, experiment freely, and reset when an experiment goes sideways, because a clean and correct starting point is always one command away.

The Code, and How to Find Any Listing

The companion repository is `github.com/RachidD68/the-entity-framework-core-and-postgresql-handbook`. It is public, and the code in it is MIT licensed, which means you may lift any of it into your own product, commercial work included. The text of the book is a separate work and is not.

Every listing in this book can be found in that repository by its printed number, and this is the mechanism. C# listings sit inside a `#region Listing NN-MM` marker in the source, where NN is the chapter and MM is the listing's position in it. Appendices use their letter in the same slot, so Appendix A's second listing is `#region Listing A-02`. Search for the number under a listing and you land on the compiling file it came from, in its real context, with its neighbours.

SQL runs the other way. For SQL the book is the source and the repository is generated from it, so the listing numbered 13.5 is a file under `sql/ch13/` whose name begins with its id, written by the render that printed the page and carrying a do-not-edit banner. The output captured beneath it is the matching file under `sql/chNN/.expected/`. Appendix E is the full map of the repository, written last and read out of the shipped tree rather than out of the plan that produced it.

One more convention travels with the code. Captions and comments in this book cite design decisions by number, so a caption may end `(D3)` and a code comment may explain a column name with `(D4)`. Those are entries in `docs/DESIGN-DECISIONS.md` in the companion repository, which records the choices that were made before Chapter 1 and then held for the whole book.

Every chapter ends with three exercises, sixty-three in all, and every one of them has a starter project with the work stubbed out and a solution project that compiles. The starters reference the same modules the book builds, so an exercise is never a copy of the application that quietly rots away from it.

Conventions Used

- `inline code` refers to a type, member, table, column, keyword, or small fragment of C# or SQL.
- Block listings are set in Cascadia Code with light-IDE syntax colours, so C# on the page reads like C# in your editor and SQL reads like SQL. SQL keywords are uppercase throughout, and long fluent chains break one call to a line, which is the house style the code itself follows.
- Each listing carries a numbered caption. There are 526 of them: 202 in C#, 116 in SQL, 109 panels of captured output, and 99 of configuration, shell, YAML and JSON.
- Output blocks reproduce the real formatting of `psql` and of the .NET console, aligned columns and row counts and all, because they were captured from those programs rather than typeset by hand.
- 💡 **Pro Tip**, ⚠️ **Warning**, 🔬 **Going Deeper**, 🌎 **Real-World Scenario** and 📝 **Key Takeaway** callouts mark content that earns a break from the narrative, and 🔗 **Cross-Reference** points to related ground elsewhere in the book.
- Two callouts are specific to this book. 🧭 **Coming from SQL Server?** flags the places where a habit carried over from SQL Server will mislead you, and 🕒 **Version Note** flags behaviour that changed in EF Core 10 or in PostgreSQL 18, so a reader on an older stack knows which page to read with a footnote in mind.
- Figures are numbered by order of appearance within their chapter, and every one of them is referenced from the prose that needs it.

Introduction

Why This Book Exists

Most of what is written about Entity Framework Core treats the database as a detail the provider will handle. Configure a DbContext, write LINQ, call SaveChanges, and let the abstraction do its job. For a surprising amount of work that is exactly right, and it is why EF Core is worth using. It stops being right at the moment your application gets big enough to matter: when a query that looked innocent returns ten thousand duplicated rows, when a migration will not apply to a partitioned table, when two writers race and one silently wins, or when the feature the product needs is a capability your database has and your ORM has never mentioned to you.

PostgreSQL is the most capable open-source database ever shipped, and the reason to run EF Core on it is not that the provider is competent. It is that the provider gives you access to JSONB documents inside relational rows, arrays and ranges as first-class column types, full-text search with ranking, geospatial queries, row-level security the database enforces whatever the application forgets, and vector similarity search, all from LINQ, all in one engine you already operate. Those are precisely the topics that generic EF Core material skips, because they do not transfer to the next provider over.

This book teaches EF Core on PostgreSQL as its own subject rather than as a dialect of EF Core on something else. Where the abstraction holds, it says so and moves on. Where PostgreSQL goes further, that is usually the point of the chapter. And every claim is grounded in one running system: LuminaWork, a multi-tenant project and field-service platform that Chapter 2 starts building and every later chapter extends, queries, indexes, audits, secures and finally publishes events from. When Chapter 7 shows you a query plan, it is a plan over data you have. When Chapter 18 hides one tenant's rows inside the database itself, it is guarding the same boundary the named query filters in Chapter 9 already guard in the application, and the two layers are then tested against each other.

How This Book Is Structured

Part	Chapters	Focus
Part I — Intermediate Core Concepts	Ch 1–5	Configuration, PostgreSQL-native mapping, relationships, migrations.
Part II — Querying and Performance	Ch 6–10	LINQ translation, plans and indexes, loading, bulk work, concurrency.
Part III — PostgreSQL Superpowers	Ch 11–15	JSONB, full-text search, arrays and ranges, PostGIS, pgvector.
Part IV — Production Architecture and Operations	Ch 16–21	Interceptors, auditing, multi-tenancy, testing, deployment, events.

Part I — Intermediate Core Concepts (Chapters 1–5)

Chapter 1 pins the stack and gets it running: .NET 10, EF Core 10.0.11, the Npgsql provider, and PostgreSQL 18.6 in a container image pinned by digest so that your server is the book's server. Chapter 2 opens the model at the level this book starts from, with pooled contexts, compiled models, and configuration split across six modules instead of piled into one file. Chapter 3 maps C# onto PostgreSQL types deliberately, including the primary key decision that everything downstream inherits. Chapter 4 covers relationships and inheritance, and Chapter 5 closes the part with migrations, including the ones EF Core cannot generate for you.

Part II — Querying and Performance (Chapters 6–10)

Chapter 6 is about translation: which LINQ becomes which SQL, and what happens at the boundary where translation stops. Chapter 7 turns on the lights with EXPLAIN and works through the index types PostgreSQL offers, against a table the bulk seed has grown to a million rows. Chapter 8 covers loading strategies and the cartesian explosion that makes one of them a trap. Chapter 9 assembles larger queries, and Chapter 10 finishes with bulk operations and concurrency, where the interesting question is not how to go fast but how to go fast without losing a write.

Part III — PostgreSQL Superpowers (Chapters 11–15)

This part is the book's namesake argument, five chapters of capability that separate PostgreSQL from a database EF Core merely runs on. Chapter 11 stores JSONB inside relational rows and indexes it, so that a product feature users define themselves stops requiring a schema change. Chapter 12 builds search, from generated tsvector columns through ranking, trigrams, and fuzzy matching. Chapter 13 covers arrays, ranges, and the exclusion constraint that makes double-booking a field engineer impossible rather than merely unlikely. Chapter 14 adds geography with PostGIS, and Chapter 15 closes the part with pgvector: embeddings, index choice, and hybrid search that fuses Chapter 12's ranking with vector similarity.

Part IV — Production Architecture and Operations (Chapters 16–21)

The last part is the distance between an application that works and one you could put in front of paying customers. Chapter 16 covers interceptors, diagnostics and the telemetry that makes a slow request explainable. Chapter 17 turns that machinery into an audit trail. Chapter 18 enforces tenant isolation twice, in EF Core query filters and in row-level security, and then tests the two layers for agreement. Chapter 19 covers testing against a real server rather than a substitute, Chapter 20 covers deployment and the migration that has to survive a rolling release, and Chapter 21 ends the book with the outbox, notifications, and change-data capture that let other systems trust yours.

The Appendices

- **Appendix A — Npgsql Provider Reference for EF Core 10:** the type mappings, the translated methods, and the connection and data-source options, in one place to look them up in.
- **Appendix B — PostgreSQL 18 SQL Features Quick Reference:** the server-side syntax the book leans on, for the moments when the answer is SQL rather than LINQ.
- **Appendix C — EF Core 10 API Changes and Migration Guide:** what changed on the way to EF Core 10, and what it costs to move.
- **Appendix D — SQL Server to PostgreSQL Translation Guide:** the habits that transfer, the ones that quietly do not, and the few that build and then do nothing.

- **Appendix E — LuminaWork Architecture and Repository Guide:** the map of the companion code, written last and read out of the shipped repository rather than out of the plan for it.

The Companion Project

The companion repository: `github.com/RachidD68/the-entity-framework-core-and-postgresql-handbook`

Clone it once and the whole book is runnable. LuminaWork is a modular monolith: one deployable application, one database, and six bounded contexts that behave as though they were strangers. Each context is a project under src/Modules, each owns exactly one PostgreSQL schema, each has its own pooled DbContext, and each keeps its own migration history so that a migration in one module can never wedge another. Sixteen tables across those six schemas, built by thirteen chapter-labelled migrations, are the whole of the model by the last page.

The application grows across the book the way real systems grow. The model is frozen early and every later change ships as an additive migration rather than a rewrite, so nothing you learn in Chapter 3 goes stale in Chapter 17. Chapter 7 loads the bulk seed, which takes the database to a million work items, half a million activity events and three hundred thousand comments, because a chapter about query plans that measures a thirty-row table is measuring nothing. The later volume arrives with the chapters that create the tables to hold it: a hundred thousand search documents once Chapter 12 has built the search schema, five thousand service locations for Chapter 14, and the embedding corpus for Chapter 15. At every step the state is deterministic, and deterministic here means something stricter than a fixed random seed: the seeder uses no random number generator at all, and every identifier and value derives from a counter, so your database is not merely similar to the book's.

One script holds it together: `scripts/reset-to-chapter.ps1 -Chapter N` on Windows, or `scripts/reset-to-chapter.sh N` on macOS and Linux. It rebuilds the database to the state chapter N expects to find when it begins, which makes the book safe to read out of order and safe to experiment against, because no mistake survives a reset. A chapter's own migrations are applied by its own narrative, so a listing that quietly depended on a later chapter's schema fails loudly during production instead of becoming your errata.

> **Going Deeper**
>
> *The container image is the supported environment, and it is one image for the whole book rather than one per part. It carries PostgreSQL 18.6, PostGIS 3.6.4 and pgvector 0.8.6 together, pinned by digest, so a reader who starts it in Chapter 1 never hits a wall in Chapter 14 or Chapter 15. Its configuration also disables just-in-time compilation and sets the write-ahead log to the logical level from day one, which is why a plan captured in Chapter 7 still matches on your machine and why Chapter 21's change-data-capture demo needs no server reconfiguration.*

A Note on This Edition

This first edition targets EF Core 10.0.11 on the Npgsql provider 10.0.3, against PostgreSQL 18.6. If you are on EF Core 9 or on PostgreSQL 16 or 17, most of the book still applies, and the Version Note callouts flag the specific behaviour that changed so you know which pages to read with a footnote in mind. Appendix C covers the EF Core side of that move in full.

One production claim deserves to be stated plainly. Every SQL listing in this book was executed against a live PostgreSQL 18.6 database by an automated harness, every C# listing is an excerpt diffed against code that compiles, and every output panel was captured from a real run. The listings meant to fail were verified to fail with exactly the error the prose predicts. Combined with the deterministic seed, that means the output on the page is not approximate, illustrative, or lightly edited. It is what the database said.

The discipline was not cheap, and it paid for itself. More than once a captured output refused to match a claim in the prose, and the prose was wrong. Chapter 14 had advised dropping to raw SQL for nearest-neighbour search that the provider translates perfectly well from LINQ. Chapter 10 had implied that a deleted row returns nothing. Chapter 11's decision matrix refused an operation that its own captured output two pages later performs. Chapter 1 pointed at the wrong chapter for conflict handling. Every one of those was caught while writing an appendix that had to state the rule flatly, checked against the running server, and fixed in the chapter. They are corrections in the book you are holding instead of entries in an errata list.

> **Key Takeaway**
>
> *You are reading a book whose every listing has already run, in order, against a database identical to yours, inside an application that compiles. Nothing here is pseudocode, nothing is simplified past the point of running, and nothing depends on state the book did not build in front of you.*

Let's Begin

Chapter 1 starts where every real project starts: pinning versions, getting a server running, and proving the connection before writing a line of domain code. If you have run EF Core against PostgreSQL before, it will read quickly and leave you with an environment the rest of the book can rely on. If you have not, it is the calmest on-ramp here. Either way, by its last page you will have a container answering your queries and a solution that builds, and the remaining twenty chapters will never let either sit idle.

Turn the page.

Contents

A map of the book — by Part, then Chapter, with each chapter's sections and page numbers listed beneath it.

Part I — Intermediate Core Concepts

Back Matter

About the Author

Rachid Dahir

Software Architect • Technical Author • Educator

Rachid Dahir designs systems for the long haul. A software architect, technical author, and educator based in Morocco, he has spent over a decade building enterprise-grade .NET solutions in financial services, healthcare, and large-scale enterprise environments — where reliability, performance, and clean architecture are not optional.

His books are written for developers who are ready to move beyond tutorials: rigorous, production-grounded, and clear enough that advanced topics never feel out of reach. Whether you are a mid-career developer leveling up, a solution architect consolidating best practices, or an enterprise team standardizing modern .NET, his goal is the same: clarity without compromise.

Rachid's approach to technical education rests on three principles:

- **Production-grade code only.** Every sample is written as if it belongs in a real system — no toy examples, no shortcuts that would never survive a code review.
- **Explain, demonstrate, practice.** Durable understanding comes from doing, not just reading. Concepts are shown in context, then reinforced through hands-on exercises.
- **No gatekeeping — only clear teaching.** Advanced topics deserve the same patience and rigor as introductory ones. The complexity of a subject is never an excuse for a poor explanation.

When he is not writing code or prose, Rachid explores mathematics, AI research, and natural health.

- **GitHub** RachidD68 (Rachid DAHIR)
- **Email** **rachiddahir@avisoft.ma**

Part I

Intermediate Core Concepts

LuminaWork goes from an empty folder to a correctly modeled, migration-managed multi-tenant schema: pooled contexts per module, PostgreSQL-native mappings with a defensible primary-key strategy, relationships and inheritance that match how the product actually works, and a migration pipeline that already handles what EF Core's differ cannot — partitioned tables included.

Chapter 1 — Setting the Stage with EF Core 10 and PostgreSQL 18

One pinned stack, one application, one container, and a first connection you can prove.

> **Real-World Scenario**
>
> *A field-service company I worked with ran its business on four tools that did not know about each other. Projects lived in a work-management SaaS. Visits lived in a scheduling product the dispatchers loved and nobody else could see. Documents lived in a shared drive. Search lived in a spreadsheet somebody exported every Monday. The day a customer asked "what happened on this site last quarter", four people spent an afternoon assembling an answer that was already out of date by the time it was sent. That is the gap LuminaWork exists to close, and it is why this book's running application carries projects, tasks, field visits, documents, and search in one database instead of four products. Every chapter from here builds a piece of it.*

This chapter is the setup chapter, and I have tried to make it the kind of setup chapter you actually read. Two things happen here. First, a tour of what EF Core 10 and PostgreSQL 18 shipped, framed as a map of the book: every feature named in the next two sections is a promise some later chapter redeems, and I will tell you which one. Second, the environment. By the last page you will have the book's PostgreSQL container running, the solution compiling, an empty luminawork database, and a health endpoint that reports the server version back to you. Nothing after this chapter works without that, and everything after this chapter assumes it exactly.

One decision shapes everything else, so let me put it first. This book pins one stack and never hedges. No "depending on your version", no compatibility matrices, no code that works around a release you might be on. Version hedging is how technical books turn into reference manuals that nobody finishes, and it is a bad trade: you get coverage of five configurations and confidence in none. Here is the one this book runs, top to bottom.

Component	Version	Where it is pinned
.NET SDK	10.0.1xx	global.json, rollForward latestFeature
C#	14	the language shipped with .NET 10
EF Core	10.0.11	LTS, supported to November 2028
Npgsql EF provider	10.0.3	Npgsql.EntityFrameworkCore.PostgreSQL
Npgsql ADO.NET driver	10.0.3	pulled in by the provider
PostgreSQL	18.6	the book's container image, digest-pinned
PostGIS / pgvector	3.6.4 / 0.8.6	baked into the same image

> **Version Note**
>
> *EF Core 10 is a Long Term Support release, supported until November 2028, and PostgreSQL 18 (September 2025) is supported by the community until 2030. That pairing is the reason this baseline is worth learning rather than tolerating. Both platforms will still be under support when the code you write while reading this is in its second year of production, which is more than most "current" stacks can say.*

What EF Core 10 shipped

EF Core 10 landed in November 2025. It is not a rewrite, and if you are coming from EF Core 8 or 9 almost everything you know still holds. What it did do is close five gaps that working teams have been routing around for years, and each of those five gets a real chapter in this book rather than a paragraph. Read the table as a table of contents with version numbers attached.

What shipped	What it changes for you	Where this book uses it

LeftJoin / RightJoin	.NET 10 added the LINQ operators; EF Core 10 translates them, retiring the SelectMany plus GroupJoin plus DefaultIfEmpty ritual	Chapter 6
Named query filters	several filters per entity type, each with a name you can disable one at a time	Chapters 9 and 18
ExecuteUpdateAsync setters	an ordinary lambda, so conditional updates stop meaning hand-built expression trees	Chapter 10
Complex types to JSON	value semantics, optional complex types, and ExecuteUpdate reaching inside a JSON column	Chapters 3 and 11
Redacted SQL logging	inlined literal values print as ? in logs unless you opt in to sensitive logging	Chapter 16

Two of those deserve a sentence of opinion before you meet them properly. Named query filters are the one I would call structural. Before EF Core 10 an entity type could carry exactly one global query filter, which meant a multi-tenant application with soft deletes had to fuse two unrelated predicates into a single expression and then disable both together or neither. LuminaWork registers two filters, named Tenant and SoftDelete, from the first migration onward. You will see their predicates riding along in captured SQL as early as Chapter 3, several chapters before Chapter 9 explains where they came from. That is deliberate: they are load-bearing, so they are present from the start rather than bolted on when the chapter arrives.

The redaction change is the one I would put on a team's upgrade checklist. EF has never logged parameter values by default, but it does sometimes inline a value straight into the SQL text, and those inlined values used to land in your logs in full. EF Core 10 replaces them with a question mark. If your operations team ever asked why a customer email address was sitting in an application log, this release is the answer, and Chapter 16 shows what the logs look like on both sides of the line.

What PostgreSQL 18 means for your EF Core code

PostgreSQL 18 arrived in September 2025 with a genuinely large release. This section is not a tour of it. The sibling volume, The PostgreSQL Handbook, owns the feature-tour

angle, and duplicating it here would waste your time. The question this book answers is narrower and more useful: which parts of PostgreSQL 18 change what you write, capture, or expect when EF Core is the thing talking to the database? Seven of them do.

PostgreSQL 18	What your EF Core code sees	Where this book uses it
Asynchronous I/O	io_method, with io_uring on Linux; read-heavy scans stop waiting one block at a time	Chapters 7 and 20
uuidv7()	time-ordered UUIDs in the database, matching Guid.CreateVersion7 in .NET	Chapter 3
Virtual generated columns	generated columns are computed on read by default now, not stored	Chapter 3
B-tree skip scan	a multicolumn index becomes usable when the leading column is not in the predicate	Chapter 7
RETURNING old and new	one statement can report the before and after values of the rows it changed	Chapter 10
OAuth 2.0 authentication	token-based logins at the server, no password in a connection string	Chapter 18
Statistics kept on upgrade	pg_upgrade now carries most optimizer statistics across, so plans do not start from nothing; extended statistics are the exception and still want an analyze pass	Chapters 7 and 20

Start with uuidv7, because it settles an argument this book would otherwise have to relitigate. Random version-4 UUIDs make terrible primary keys under load: every insert lands in a random leaf of the index, the buffer cache thrashes, and the index bloats. Version 7 puts a millisecond timestamp in the high bits, so new keys sort to the right-hand edge of the B-tree the way an identity column does, while staying globally unique and safe to generate on the client. .NET 9 gave us Guid.CreateVersion7, and PostgreSQL 18 gives us uuidv7() on the server side. LuminaWork uses both: EF generates the key so

the application knows the id before SaveChanges runs, and the column keeps DEFAULT uuidv7() as a backstop for rows that arrive from outside EF. Chapter 3 builds that pair and explains why having two generators is not indecision.

The asynchronous I/O subsystem is the change with the widest blast radius and the least code impact, which is an unusual combination. Nothing in your C# changes. What changes is that a sequential scan, a bitmap heap scan, or a vacuum can now have several reads in flight instead of blocking on one at a time. On Linux with io_uring the effect on scan-heavy work is large; the portable worker-based implementation, which is what the book's container runs, is more modest and still real. Chapter 7 reads io_method off a live server while looking at plans, and Chapter 20 treats it as the deployment knob it is.

Skip scans and preserved statistics are both about not being surprised. A skip scan lets PostgreSQL use a multicolumn index even when your predicate skips the leading column, which quietly rescues queries that would have seq-scanned on 17. Preserved statistics mean an upgraded cluster keeps most of its planner numbers instead of waking up statistically blind. Extended statistics are the exception and still do not travel, so the post-upgrade analyze pass stays on the runbook; it is simply no longer the difference between a working database and an outage. If you have ever watched a Monday-morning incident after a Sunday-night major upgrade, you already know what that is worth. RETURNING old and new is the smallest of the seven and the one you will reach for most: it turns "update this row and tell me what it used to be" into a single round trip, which Chapter 10 uses for optimistic concurrency and Chapter 17 for audit trails.

> **Key Takeaway**
>
> *Every feature named in these two sections is a promise. If a later chapter does not cash it, this chapter oversold it. Nothing here is included because it made the release notes; it is here because LuminaWork uses it, and the chapter number in the third column is where you can go and check.*

Npgsql: the driver, the provider, and one version rule

"EF Core talks to PostgreSQL" hides four layers, and knowing which layer owns which problem is what turns a confusing stack trace into a five-minute fix. Figure 1.1 lays them out.

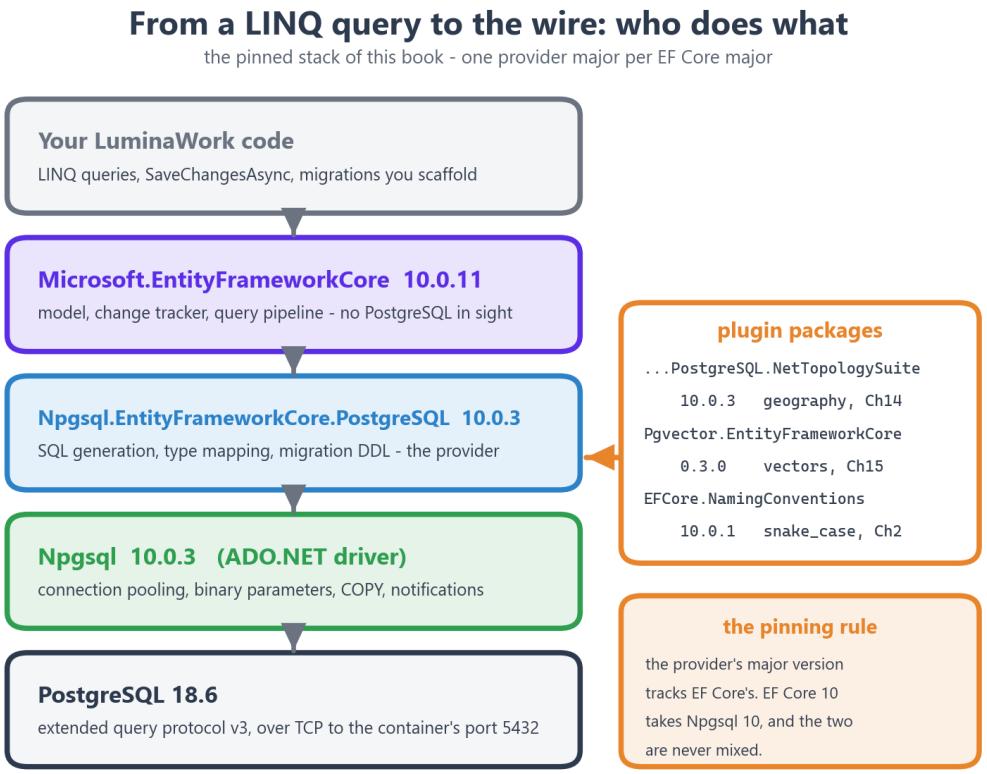

Figure 1.1 — The four layers between a LINQ query and PostgreSQL 18, the three plugin packages that extend the provider, and the version rule that keeps them compatible.

EF Core itself knows nothing about PostgreSQL. It owns the model, the change tracker, and a query pipeline that produces an abstract SQL expression tree. The provider, Npgsql.EntityFrameworkCore.PostgreSQL, turns that tree into PostgreSQL's dialect, decides how a .NET type maps to a database type, and generates migration DDL. Underneath it sits Npgsql itself, the ADO.NET driver, which owns connections, pooling, binary parameter encoding, COPY, and notifications. Below that is the wire: PostgreSQL's extended query protocol over TCP. Symptoms sort themselves by layer

once you know the layout. A LINQ expression that will not translate is an EF Core problem. Odd SQL or a type that maps to the wrong column is a provider problem. A pool exhausted under load or a timeout that fires too early is a driver problem, and its knobs live in the connection string, not in your DbContext.

The version rule is short. The provider's major version tracks EF Core's major version, so EF Core 10 takes Npgsql 10 and nothing else. There is no build of the 9.x provider that speaks EF Core 10's internal APIs, and mixing them produces a load-time failure rather than a graceful degradation. Patch versions move independently, which is why this book pins EF Core at 10.0.11 and the provider at 10.0.3 without either number being a typo. The plugin packages follow the provider: NetTopologySuite for geography in Chapter 14, Pgvector.EntityFrameworkCore for embeddings in Chapter 15, and EFCore.NamingConventions, which is the small package that gives this whole book its snake_case schema from a single call in Chapter 2.

One name worth meeting now, because it shows up in later chapters without ceremony: NpgsqlDataSource. It is the modern entry point to the driver, built from an NpgsqlDataSourceBuilder, and it owns the connection pool, parses the connection string once, and is where driver-level concerns like logging and type plugins get configured. When you call UseNpgsql with a connection string, the provider builds one for you. When you need the driver directly, you build your own: the bulk loader that puts a million rows into this database for Chapter 7 opens a COPY stream from a data source, because COPY is an Npgsql feature with no EF Core equivalent.

Coming from SQL Server?

If you are arriving from SQL Server, the shape of what you know transfers cleanly and the details do not. DbContext, change tracking, LINQ, migrations, and the whole configuration model are provider-agnostic and behave exactly as they did. What changes is everything below the provider line in Figure 1.1. Type mapping is different and more literal (Chapter 3). Case sensitivity is real, so an unquoted identifier folds to lower case and a text comparison is case-sensitive by default (Chapter 3 again). rowversion has no equivalent; concurrency uses the system column xmin (Chapter 10). MERGE exists but INSERT ON CONFLICT is what you will actually write, and Chapter 12 writes one. And there is no NOLOCK, because readers never block writers under MVCC. Appendix D is the full translation table; the rest of this book assumes you will meet these one at a time.

Meet LuminaWork

LuminaWork is a multi-tenant project and field-service platform. Teams share a workspace, organize work into projects and boards, break it into tasks, attach documents and discussion to those tasks, dispatch field visits to physical locations, and search all of it by keyword, fuzzy match, meaning, or proximity. That is the whole product. It is small enough to hold in your head and awkward enough to be honest: multi-tenancy, three kinds of task, geography, full-text and vector search, an audit trail, and an outbox all show up because real products in this space have them, not because a chapter needed a subject.

The architecture is a modular monolith: one deployable, one database, six modules that own their own schemas and do not read each other's tables. I chose that shape for a specific reason. Microservices would have forced a distributed-systems book, and a single undifferentiated schema would have let every chapter cheat by joining anything to anything. A modular monolith keeps the pressure on: modules have boundaries you can feel, and Chapter 4 will make you decide, edge by edge, which references get a foreign key and which do not. Figure 1.2 is the map, and it is worth a minute of your attention now because the rest of the book navigates by it.

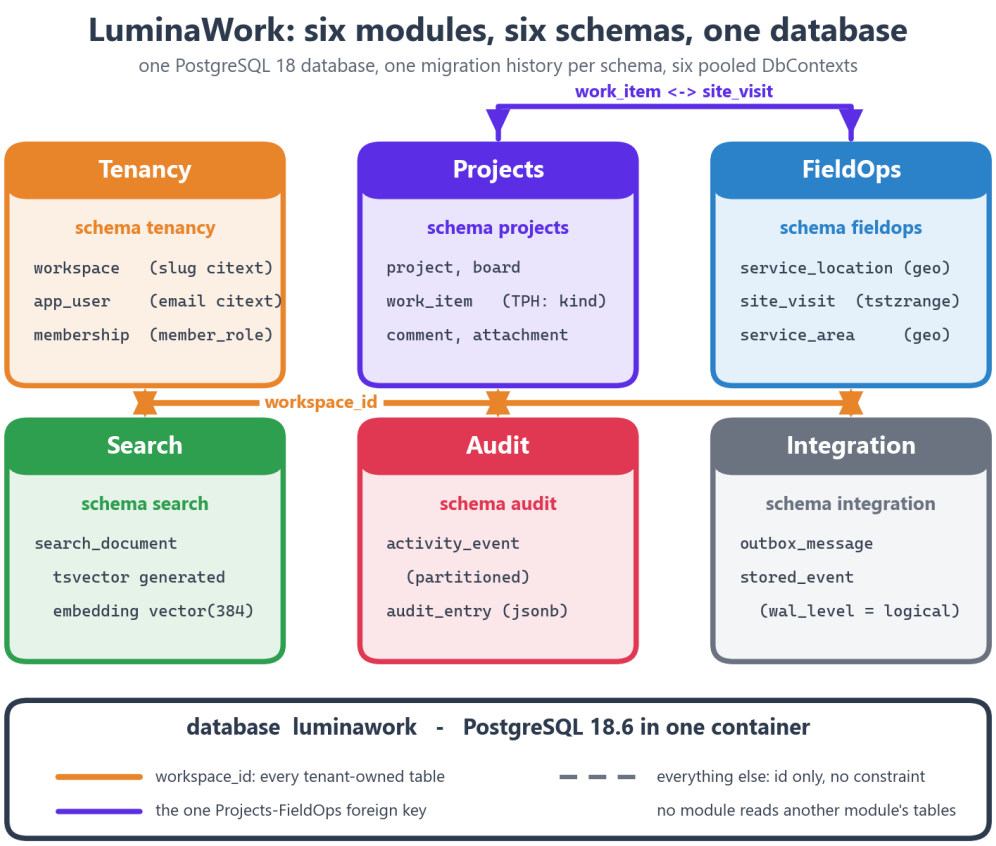

Figure 1.2 — LuminaWork's six modules, the schema each one owns, and the only two kinds of foreign key allowed to cross a module boundary.

Tenancy owns workspaces, users, and memberships, and it is the module every other module depends on: every tenant-owned table in the database carries a workspace_id, and that column is the one foreign key allowed to point across a module boundary in general. Projects owns projects, boards, tasks, comments, and attachments, and it is where most of Part I and Part II happen. FieldOps owns service locations, service areas, and site visits, which is where geography and range types live. Search owns the documents, generated tsvectors and embedding vectors that Part III indexes. Audit owns the activity feed and the change history. Integration owns the outbox and the event store that Chapter 21 turns into change data capture. One extra edge is sanctioned, between a task and its site visit, and Chapter 4 explains why that one earns an exception when the others do not.

Six modules also means six DbContexts, six schemas, and six migration histories in one PostgreSQL database. That arrangement is not the obvious one, and Chapter 2 spends most of its length earning it. For now the important consequence is practical: a migration in one module cannot break another module's history table, and any chapter can reset the database to exactly the state it needs without dragging in schema from chapters you have not read yet.

Here is the repository you are about to clone. Company code is not usually this tidy, and I will not pretend otherwise. Every path below is real.

```
luminawork/
  src/
    LuminaWork.Web/          the host: six pooled contexts, one health endpoint
    LuminaWork.Database/     the migrate + seed runner the reset script drives
    LuminaWork.Demos/        one console project, one subcommand per printed demo
    Modules/
      SharedKernel/          tenant interfaces, connection resolution, uuid v7
      Tenancy/  Projects/  FieldOps/  Search/  Audit/  Integration/
  tests/                     migration tests and Testcontainers tests (Chapter 19)
  benchmarks/                BenchmarkDotNet projects and their committed results
  exercises/chNN/            Starter and Solution per chapter
  samples/                   small demos that would distort the main application
  deploy/docker/             Dockerfile, docker-compose.yml, postgres.conf, init/
  sql/chNN/                  every SQL listing, generated from the book's source
  seed/                      the precomputed embedding corpus (Chapter 15)
  scripts/  tools/           reset-to-chapter, capture-output, verify-listings
```

One convention makes the repository usable while you read. Every C# listing in this book sits inside a `#region Listing NN-MM` marker in the source. Listing 1.8 below lives inside `#region Listing 01-08`, so searching the repository for the number printed under a listing lands you on the real, compiling file it came from, in its real context, with its neighbors. This is enforced rather than promised: a tool diffs every printed C# line against its region before the book builds, and a mismatch fails the build. SQL runs the other way. The book is the source, and the `sql/chNN/` files are generated from it, with a do-not-edit banner and a note saying which chapter state they expect.

> **Key Takeaway**
>
> *LuminaWork is one application, six modules, one database, six schemas. The module boundary is the teaching device: if a chapter wants to join across it, the chapter has to justify the edge or find another way. Figure 1.2 is worth coming back to whenever a later chapter says "the*

Projects module" and you want to know what else is in the room.

The book environment: one container, one volume, one port

Every listing in this book runs against one PostgreSQL, built from one Dockerfile, with one configuration file. That is not developer convenience; it is what makes the printed output trustworthy. When Chapter 7 prints an execution plan, you should be able to run the same query and get the same plan, and that is only true if your server has the same extensions, the same planner settings, and the same JIT policy as mine. So the image is pinned by digest, the settings are baked in, and nothing is installed by hand. Start with the compose file, which is the whole environment in under thirty lines.

```
services:
  postgres:
    build: .
    image: luminawork-postgres:18
    container_name: luminawork-pg
    environment:
      POSTGRES_DB: luminawork
      POSTGRES_USER: luminawork
      POSTGRES_PASSWORD: luminawork
    ports:
      - "${LW_PG_PORT:-5432}:5432"
    volumes:
      # PostgreSQL 18 images moved the mount point: mount /var/lib/postgresql
      # (data lands in a versioned subdirectory), NOT the pre-18
      # /var/lib/postgresql/data — the container refuses to start otherwise.
      - luminawork-data:/var/lib/postgresql
      - ./postgres.conf:/etc/postgresql/postgresql.conf:ro
      # Runs only when the data volume is brand new; creates the app/owner roles.
      - ./init:/docker-entrypoint-initdb.d:ro
    command: ["postgres", "-c", "config_file=/etc/postgresql/postgresql.conf"]
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U luminawork -d luminawork"]
      interval: 5s
      timeout: 3s
      retries: 12

volumes:
  luminawork-data:
```

Listing 1.1 — deploy/docker/docker-compose.yml, complete: the port is a variable, the config is a read-only mount, and the health check is what "ready" means.

Four details in there matter later. The published port is `${LW_PG_PORT:-5432}`, so the container listens on 5432 inside and on whatever you choose outside; the next section explains when you would choose. The configuration file is a read-only bind mount, which means editing `postgres.conf` and restarting is the whole workflow for changing a server setting. The `init` directory runs once, on a brand-new volume only, and creates the two database roles Chapter 18 needs. And the health check is a real one: `docker compose up -d` returns long before PostgreSQL accepts connections, so "the container started" and "the database is ready" are different events, and the health check is how you tell them apart.

The image itself is nine lines, and most of them are a comment explaining the pin. The base is the official PostGIS image, which is the fussy install done for you, and pgvector comes from the PostgreSQL Global Development Group repository the base image already trusts.

```
# LuminaWork book image — PostgreSQL 18 + PostGIS 3.6 + pgvector + contrib.
# One image for the whole book: Chapter 1's setup works unchanged through Ch21.
# Digest pinned at the B0 acceptance test: PostgreSQL 18.6, PostGIS 3.6.4,
# pgvector 0.8.6, pg_trgm 1.6, citext 1.8, btree_gist 1.8.
FROM postgis/postgis:18-3.6@sha256:db8c151a4e1f4686b1a985a3490cf96f9f8c8c2725...

RUN apt-get update \
 && apt-get install -y --no-install-recommends postgresql-18-pgvector \
 && rm -rf /var/lib/apt/lists/*

# --- planner pins (EXPLAIN in the book must match a reader's run) --------
# JIT flips plans above a hardware-dependent cost threshold on PostgreSQL 18
# and adds a JIT block to EXPLAIN output. Off, deliberately, for the whole book.
jit = off
random_page_cost = 1.1
max_parallel_workers_per_gather = 2

# --- logical replication (Chapter 21 CDC; baked in from day one) ---------
wal_level = logical
max_wal_senders = 10
max_replication_slots = 10
```

Listing 1.2 — The Dockerfile, and the half of postgres.conf that keeps printed plans honest. The digest's trailing hex digits are elided for the page; the repository has all sixty-four.

Take the planner pins seriously, because they are the difference between a book that reproduces and one that argues with you. With JIT enabled, PostgreSQL 18 changes plan shape above a cost threshold that depends on your hardware, and adds a JIT section to EXPLAIN output that would appear on my machine and not yours. Off, for everyone, forever. The value of random_page_cost assumes solid-state storage, which every machine that will run this book has. And wal_level is set to logical on the first day of the book for a reason we cash in at the very end: logical replication cannot be switched on without a restart, so a database that might ever need change data capture should be born with it. Chapter 21 collects on that.

> ⚠ **Warning**
>
> *PostgreSQL 18's official Docker images moved the data directory. Mount the volume at /var/lib/postgresql, not at the pre-18 /var/lib/postgresql/data. The server now keeps its cluster in a major-version subdirectory inside that mount, which is what makes an in-place pg_upgrade possible later. If you adapt an older compose file and keep the old path, the container does not start with a warning; it exits with code 1 and a message that begins "Error: in 18+, these Docker images are configured to store database data in a format which is compatible with pg_ctlcluster". I hit this on the first build of the book's own image. Exercise 1.3 makes you reproduce it on purpose, in a sandbox, so you recognize it in three years when it happens to a colleague.*

With the compose file in front of you, bringing the environment up is two commands and one wait. Run them from the deploy/docker directory. The build prints Docker's own progress trace, which is Docker's business; the answer you want comes from the second command.

```
cd deploy/docker
docker compose up -d --build
docker compose ps --format "table {{.Name}}\t{{.Status}}\t{{.Ports}}"
```

```
NAME            STATUS                   PORTS
luminawork-pg   Up 7 seconds (healthy)   0.0.0.0:5433->5432/tcp, [::]:5433->5432/tcp
```

Listing 1.3 — First boot, captured: healthy, and published where the local .env said to publish it. The default format adds four more columns than a page can hold, so this asks for the two that matter.

The expensive layer is the pinned base image, cached after the first pull. On this machine the container reported healthy six seconds after it started. The published port

in that capture is 5433 rather than 5432, because this machine already runs a native PostgreSQL on 5432 and the local .env says so; yours will read 5432 unless you change it, and the next section is where that happens.

Figure 1.3 shows what docker compose up actually assembled: the pinned image layers, the named volume at the path PostgreSQL 18 expects, the two read-only mounts, and the port mapping every later chapter assumes without saying so.

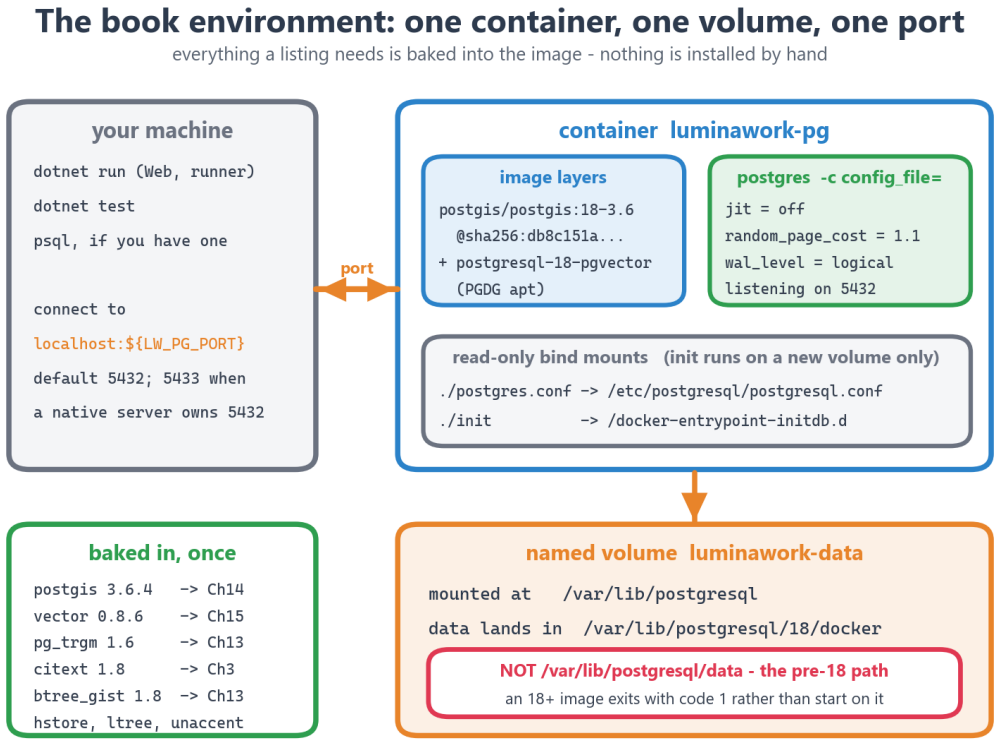

Figure 1.3 — The book image, the named volume, the read-only mounts, and the port mapping: the environment every later chapter assumes without saying so.

Now prove it, because a container that starts is not the same as a server that has what the book needs. Two queries settle it. The first asks the server what it is, and the second asks which extensions it can install.

```
SELECT current_setting('server_version') AS server_version;
```

```
SELECT name, default_version
FROM pg_available_extensions
WHERE name IN ('postgis', 'vector', 'pg_trgm', 'citext',
               'btree_gist', 'hstore', 'ltree', 'unaccent')
ORDER BY name;
```

```
        server_version
------------------------------
 18.6 (Debian 18.6-1.pgdg13+2)
(1 row)

    name    | default_version
------------+-----------------
 btree_gist | 1.8
 citext     | 1.8
 hstore     | 1.8
 ltree      | 1.3
 pg_trgm    | 1.6
 postgis    | 3.6.4
 unaccent   | 1.1
 vector     | 0.8.6
(8 rows)
```

Listing 1.4 — The image, proved rather than assumed: the exact server build, and the eight extensions Parts II and III depend on.

The setting is the readable half of what SELECT version() returns; the full banner adds the compiler and the architecture, which are worth knowing once and never again. Note what the second query asks and what it does not. It reads pg_available_extensions, which lists what this server could install, not pg_extension, which lists what a database has installed. At this point in the book the luminawork database is empty and no extension is installed in it. That is correct: extensions are schema objects, they belong to a database, and the migrations that need them create them, starting with citext in Chapter 3. What matters today is that the binaries are on the server, so nobody reaches Chapter 14 and discovers that geography needs an install they cannot do.

Pro Tip

The digest pin is the part of the image people skip, and it is the part that makes the rest work. A tag like postgis:18-3.6 is a moving target: the same tag on your machine in six months is a different image, with a different PostgreSQL patch level and possibly a different plan for the same query. Pinning by sha256 means every reader of this book, on every machine, on any day, gets the same server. Combined with ${LW_PG_PORT:-5432}, which lets your port differ from mine without a single file being edited, that is the whole reproducibility story.

One more query, because two PostgreSQL 18 features from the table above can be made concrete in a handful of lines, and seeing them beats reading about them.

```
SELECT current_setting('io_method')  AS io_method,
       current_setting('wal_level')  AS wal_level,
       current_setting('jit')        AS jit;

-- uuidv7() is not deterministic, so assert its properties, not its value.
SELECT uuid_extract_version(uuidv7())    AS uuid_version,
       uuid_extract_timestamp(uuidv7())
           > now() - interval '1 second' AS is_time_ordered;
```

```
 io_method | wal_level | jit
-----------+-----------+-----
 worker    | logical   | off
(1 row)

 uuid_version | is_time_ordered
--------------+-----------------
            7 | t
(1 row)
```

Listing 1.5 — Two promises made concrete: the settings the image bakes in, and a native uuidv7() whose timestamp is genuinely now.

The io_method value is worker, which is the portable implementation and the default. On a Linux host you can set it to io_uring and get more from the same hardware; the book's container stays on worker so that a reader on Windows, macOS, or Linux sees the same setting in the same place. The uuid query is written the way every non-deterministic thing in this book is written: it asserts the properties that matter, the version digit and the fact that the embedded timestamp really is the current time, instead of printing a value that would differ on every run and turn the page into fiction.

Key Takeaway

The environment is pinned three ways: the image by digest, the server settings by a baked-in configuration file, and the port by a variable. That is why the output printed in this book is reproducible on your machine, and it is also the shape I would recommend for any team that wants its local database to behave like its production one.

Project setup: the toolchain, the pins, the connection

The .NET side is smaller than the database side, which is how it should be. Clone the repository, restore the local tools, build. The tool restore matters more than it looks: dotnet-ef is pinned as a local tool rather than installed globally, so the migration commands in Chapter 5 run the same version on your machine as on mine and as in continuous integration, with nothing to install and nothing to conflict.

```
dotnet --version
dotnet tool restore
dotnet ef --version
```

```
10.0.400
Tool 'dotnet-ef' (version '10.0.11') was restored. Available commands: dotnet-ef

Restore was successful.
Entity Framework Core .NET Command-line Tools
10.0.11
```

Listing 1.6 — The toolchain, captured: an SDK that satisfies global.json, and dotnet-ef restored at the pinned version.

Then build it: `dotnet build LuminaWork.sln` for the application and `dotnet build exercises/Exercises.sln` for the practice projects. Both are configured with warnings as errors, so the only acceptable warning count is zero, and both should reach "Build succeeded" in well under a minute on a warm NuGet cache. If the build fails here, stop and fix it: every chapter after this one prints code from that solution.

The SDK number you see will not necessarily match mine, and that is by design. The global.json in the repository asks for the 10.0.1xx feature band with rollForward set to latestFeature, which means any .NET 10 SDK from 10.0.100 upward satisfies it. What it will not do is silently roll forward to .NET 11 when that ships. Package versions get the same treatment, one level stricter: central package management puts every version in one file, so no project can quietly float.

```
// global.json
{ "sdk": { "version": "10.0.100", "rollForward": "latestFeature" } }

<!-- Directory.Packages.props (excerpt: the whole file is one ItemGroup) -->
<PackageVersion Include="Microsoft.EntityFrameworkCore" Version="10.0.11" />
<PackageVersion Include="Microsoft.EntityFrameworkCore.Design" Version="10.0.11" />
```

```
<PackageVersion Include="Npgsql" Version="10.0.3" />
<PackageVersion Include="Npgsql.EntityFrameworkCore.PostgreSQL" Version="10.0.3" />
<PackageVersion Include="Npgsql.EntityFrameworkCore.PostgreSQL.NetTopologySuite"
                Version="10.0.3" />
<PackageVersion Include="EFCore.NamingConventions" Version="10.0.1" />
<PackageVersion Include="Pgvector.EntityFrameworkCore" Version="0.3.0" />

// .config/dotnet-tools.json
"dotnet-ef": { "version": "10.0.11", "commands": [ "dotnet-ef" ] }
```

Listing 1.7 — Three files, one rule: the SDK band, every package version, and the EF tool are pinned in source control, not in anybody's habits.

Now the connection. Every process in this repository resolves it the same way, through one small class in the shared kernel. The reason is boring and important. The web host, the migrate-and-seed runner, the design-time factories that dotnet-ef uses, the test suites, and the capture harness must all agree on which server they are talking to, or the book's output stops being reproducible.

```
/// <summary>
/// One connection-string resolution order everywhere - runner, design-time factories,
/// and the reset script all agree: <c>LUMINAWORK_CS</c> wins outright; otherwise
/// localhost with <c>LW_PG_PORT</c> (default 5432) and the book image's
/// luminawork database, user, and password.
/// </summary>
public static class LuminaWorkConnection
{
    public static string Resolve()
    {
        var explicitConnectionString = Environment.GetEnvironmentVariable("LUMINAWORK_CS");
        if (!string.IsNullOrWhiteSpace(explicitConnectionString))
        {
            return explicitConnectionString;
        }

        var port = Environment.GetEnvironmentVariable("LW_PG_PORT");
        if (string.IsNullOrWhiteSpace(port))
        {
            port = "5432";
        }

        return $"Host=localhost;Port={port};Database=luminawork;" +
               "Username=luminawork;Password=luminawork";
    }
}
```

Listing 1.8 — Two environment variables and a default: the whole connection story, in the one place every process reads it from.

Notice what is not there. No appsettings.json, no configuration binding, no user secrets. That is a simplification for a book, not a recommendation for production, where the connection string belongs in your configuration system and the password in a secret store. What it buys is a reader who can change the target server with one environment variable and no file edits.

Here is that story. The container publishes its 5432 on whatever host port you name, and plenty of developers already run a native PostgreSQL on 5432. I do. So the book's default is 5432 and the escape hatch is a single line in a file docker compose reads.

```
# deploy/docker/.env.example - copy to deploy/docker/.env and edit if needed
LW_PG_PORT=5432


# On a machine where a native PostgreSQL already owns 5432:
cd deploy/docker
cp .env.example .env
#   ... then set LW_PG_PORT=5433 in it and: docker compose up -d
```

Listing 1.9 — The ports story in one file. It lives beside the compose file because that is the .env docker compose reads.

The location is the part people get wrong, so I will be explicit: the file has to be deploy/docker/.env, next to the compose file. A .env at the repository root is read by nothing. That same file is where the reset script and the capture harness look for the port as well, so setting it once puts those three on the same server. Processes you start yourself are the exception, because a .env file is not shell configuration: `dotnet run` and `dotnet test` read LW_PG_PORT from your environment. On a machine where 5432 is taken, set it there too, with `$env:LW_PG_PORT = '5433'` in the shell you work in.

The host that pulls it together is a minimal ASP.NET Core application. Chapter 2 spends a whole chapter on its six context registrations, so here just read the shape: resolve the connection once, decide how much SQL you want in the log, register the services every module shares, register the contexts, build. One registration is printed in full; the other five are elided, because they are the same four lines with a different context type.

```
var builder = WebApplication.CreateBuilder(args);
```

```
// One resolved connection string for the whole host - the runner, the design-time
// factories, and scripts/reset-to-chapter.ps1 all read the same two variables.
var connectionString = LuminaWorkConnection.Resolve();

// EF Core 10 redacts inlined literal values from the SQL it logs, so this category
// can stay at Information in environments where those values would be a liability.
builder.Logging.AddFilter(
    "Microsoft.EntityFrameworkCore.Database.Command", LogLevel.Information);

builder.Services.AddSingleton<ITenantContextAccessor, TenantContextAccessor>();
builder.Services.AddSingleton<TenantConnectionInterceptor>();

builder.Services.AddDbContextPool<TenancyDbContext>((provider, options) =>
{
    TenancyDbContext.Configure(options, connectionString);
    options.AddInterceptors(provider.GetRequiredService<TenantConnectionInterceptor>());
});

// …

var app = builder.Build();
```

Listing 1.10 — The composition root: one connection string, one log decision, two shared singletons, and the first of Chapter 2's six pooled context registrations.

AddDbContextPool is where a context enters the container, and the provider attaches one level down: TenancyDbContext.Configure calls UseNpgsql with that connection string, which is the line that makes this an EF Core application talking to PostgreSQL rather than to anything else. Chapter 2 prints that method in full and explains why the recipe lives on the context class instead of at the registration site, where the design-time tooling cannot reach it.

That logging line is worth thirty seconds. Turning the Database.Command category up to Information prints every SQL statement EF executes, which is the single most useful thing you can do while learning an ORM, and historically the most dangerous thing you could leave on in production. EF Core 10 changes that calculation, because the inlined literals that used to leak into those logs are now redacted to question marks unless you explicitly call EnableSensitiveDataLogging. Chapter 16 shows both outputs side by side and covers the interceptors that give you finer control.

> **Cross-Reference**
>
> *Two production concerns are deliberately absent from this host and get proper treatment later. Connection resiliency, EnableRetryOnFailure, wraps every command in an execution strategy that retries transient failures, and it changes how you must write manual transactions; Chapter 7 introduces it beside async, cancellation, and the rest of the connection story, and Chapter 19 proves it against a container that really does drop connections. Deployment concerns, including pooling limits, health probes, and io_method on the server, are Chapter 20.*

First connection, and how this book runs

The last piece of setup is an endpoint that answers the only question that matters right now. It does two things: asks EF whether it can reach the database at all, then asks the database what it is. The second half is the part I would keep in a real service, because "cannot connect" is an easy failure to diagnose and "connected to the wrong server" is not.

```
app.MapGet("/health", async (TenancyDbContext tenancy) =>
{
    if (!await tenancy.Database.CanConnectAsync())
    {
        return Results.Problem(
            "database unreachable",
            statusCode: StatusCodes.Status503ServiceUnavailable);
    }

    // SqlQuery<T> wants the column aliased "Value". Reporting the server version
    // catches "connected to the wrong PostgreSQL" as well as "not connected".
    var serverVersion = await tenancy.Database
        .SqlQuery<string>($"SELECT version() AS \"Value\"")
        .SingleAsync();

    return Results.Ok(new { status = "ok", server = serverVersion });
});
```

Start the host with `dotnet run --project src/LuminaWork.Web` and ask it: `curl http://localhost:5000/health`. With every log category except the command one turned down to warnings, the console shows precisely the two statements the endpoint ran, and then the answer:

```
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (Nms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT 1
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (Nms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT s."Value"
      FROM (
          SELECT version() AS "Value"
      ) AS s
      LIMIT 2

{"status":"ok","server":"PostgreSQL 18.6 (Debian 18.6-1.pgdg13+2) on x86_64-pc-linux-gnu,
        compiled by gcc (Debian 14.2.0-19) 14.2.0, 64-bit"}
```

Listing 1.11 — The first connection, captured: CanConnectAsync spends a SELECT 1, the version query arrives wrapped for SingleAsync, and the endpoint answers with the server it actually reached.

Read the log, because both halves are visible in it. CanConnectAsync spends exactly one round trip, SELECT 1, which proves reachability and credentials without asserting that any table exists; that is precisely right for a database with no schema in it yet. SqlQuery is EF's way of running SQL that returns a scalar or a simple shape rather than an entity, parameterized through an interpolated string so the same protection you get from LINQ applies, and it wants the column aliased Value so EF knows which one to read. The wrapping in the capture is SingleAsync doing its job: it asks for two rows so it can throw if a second one turns up, which is the difference between Single and First made visible in SQL.

Which leaves the workflow every remaining chapter uses. The database has a defined state at the start of each chapter, and one script produces it.

```
scripts\reset-to-chapter.ps1 -Chapter 2

Resetting luminawork to chapter 2 via container (luminawork-pg) ...
Migrating to chapter 2 entry state...
  [2/3] Projects    -> (not migrated at this state)
  [2/3] FieldOps    -> (not migrated at this state)
  [2/3] Search      -> (not migrated at this state)
  [2/3] Audit       -> (not migrated at this state)
  [2/3] Integration -> (not migrated at this state)
  [3/3] Tenancy     -> (not migrated at this state)
seed: chapter 2 entry state has no schema yet - nothing to seed.
Done. State: chapter 2; seed: core; VACUUM (ANALYZE) complete.
```

Listing 1.12 — Entry state for Chapter 2, captured: six contexts considered in order, none migrated yet, nothing to seed. An empty database is exactly what Chapter 2 wants.

Read the flag as an entry state, not a target. Passing -Chapter 2 gives you the database as it is at the moment you begin Chapter 2, which means a chapter's own migration is never pre-applied; the chapter applies it in front of you, in reading order. That is why the output above has nothing to do: Chapter 2 changes no schema at all, so its entry state is the empty database you already have. Run the same script with -Chapter 8 and it migrates six contexts to the targets Chapter 8 expects, seeds deterministic data, loads a million tasks in bulk, and finishes with VACUUM ANALYZE, because plans printed against stale statistics are noise.

Underneath, the script drops and recreates the luminawork database, so treat it as destructive and never point it at anything you care about. It talks to the container by name when the container is running, which is why a reader with Docker needs no psql installed at all, and falls back to a local psql otherwise. Seeded data is deterministic by construction, with no random number generator anywhere in it: a fixed time anchor of 2026-01-15 09:00 UTC, and every id derived from a counter rather than a clock. Two people running -Chapter 8 on opposite sides of the world get byte-identical rows.

Going Deeper

The printed output in this book is captured, never typed. A tool reads each chapter's listing manifest, replays every SQL listing against the correct chapter state in order, and writes the result to a file the render script reads at build time. If a listing's output changes because the schema or the seed changed, the book changes with it or the build fails. That inversion is the single most useful thing I have built for a technical book, and it is why the phrase "your output may vary" does not appear anywhere in these pages.

Key Takeaway

You now have the environment the rest of the book assumes: a digest-pinned PostgreSQL 18 container with PostGIS and pgvector, a solution that builds, a pinned toolchain, and a health endpoint that proves the connection. The workflow to remember is one line long. Before any chapter, run reset-to-chapter with that chapter's number, and the database will be exactly what the chapter expects.

Summary

This chapter pinned a stack and refused to hedge about it: .NET 10, C# 14, EF Core 10.0.11 on the Npgsql 10.0.3 provider, and PostgreSQL 18.6 in a digest-pinned container. You toured what both platforms shipped, with every feature tied to the chapter that uses it rather than left as a headline. From EF Core 10 that means named query filters, the LeftJoin and RightJoin translations, ExecuteUpdateAsync with ordinary lambdas, complex types mapped to JSON, and SQL logs that redact inlined literals. From PostgreSQL 18 it means native uuidv7, asynchronous I/O, virtual generated columns, B-tree skip scans, RETURNING old and new, OAuth authentication, and planner statistics that mostly survive an upgrade.

You met LuminaWork, its six modules, and the modular-monolith rule that makes the module boundary a real constraint instead of a folder layout. Then you built the environment: one compose file, one digest-pinned image with PostGIS and pgvector already in it, one volume mounted where PostgreSQL 18 wants it, and a port that moves with a variable rather than an edit. The server proved itself with its own version banner and its extension list, the toolchain proved itself with a restored dotnet-ef, and a health endpoint proved the connection end to end. The last thing you learned is the one you will use most: reset-to-chapter produces the state a chapter begins with, so no listing in this book ever depends on your having finished the previous one.

Key Terms

- **EF Core 10** — the November 2025 Long Term Support release of Entity Framework Core, supported until November 2028; requires .NET 10.
- **PostgreSQL 18** — the September 2025 major release; the book runs 18.6 from a digest-pinned container image.
- **LTS** — Long Term Support: a release with a published, multi-year support window, which is why this book's baseline is worth learning rather than tolerating.
- **Npgsql** — the ADO.NET driver for PostgreSQL; the EF provider Npgsql.EntityFrameworkCore.PostgreSQL sits on top of it and tracks EF Core's major version.

- **NpgsqlDataSource** — the driver's modern entry point, built by NpgsqlDataSourceBuilder; owns the connection pool and driver-level configuration.
- **connection resiliency** — EnableRetryOnFailure, an execution strategy that retries transient failures and changes how manual transactions must be written (Chapter 7).
- **io_method** — PostgreSQL 18's asynchronous I/O implementation: worker everywhere, io_uring on Linux; the book's container runs worker.
- **uuidv7()** — a PostgreSQL 18 function returning a time-ordered UUID; the server-side counterpart of .NET's Guid.CreateVersion7.
- **B-tree skip scan** — PostgreSQL 18's ability to use a multicolumn index when the query's predicate skips the leading column.
- **virtual generated column** — a generated column computed on read rather than stored; the default in PostgreSQL 18.
- **RETURNING old/new** — PostgreSQL 18 syntax letting one statement report both the pre-change and post-change values of the rows it touched.
- **named query filters** — EF Core 10's several-filters-per-entity feature, each filter carrying a name that can be disabled independently.
- **SQL log redaction** — EF Core 10's default of printing inlined literal values as ? in logged SQL unless sensitive data logging is enabled.
- **Docker volume** — the named store holding the database files; on PostgreSQL 18 images it is mounted at /var/lib/postgresql, not the pre-18 /var/lib/postgresql/data.
- **LuminaWork** — this book's running application: a multi-tenant project and field-service platform built as a modular monolith of six modules over one PostgreSQL database.

Practice Exercises

Starter and solution projects live in the companion repository under `exercises/ch01`. Each starter compiles with failing tests; the tests are the definition of done. Chapter 1's entry state is an empty database, so `scripts\reset-to-chapter.ps1 -Chapter 1` is all the setup these need, and the container has to be running.

Exercise 1.1 — Basic: Light it up

The starter ships a failing connectivity check. Bring the container up, put the right port in deploy/docker/.env if 5432 is taken on your machine, then implement two methods so the first two tests go green: one that opens a connection and runs the smallest possible round trip, and one that reports the server's major version, which the test asserts is 18 or newer. Let a wrong host or port throw. A broken environment should say so loudly rather than return false.

Hint: the server exposes its version as an integer setting, server_version_num, shaped like 180006. One division gets you the major version and avoids parsing the banner string.

Exercise 1.2 — Intermediate: Trust, then verify

Extend the check to prove the image, not your luck. Query the catalog for the five extensions this book leans on hardest, postgis, vector, pg_trgm, citext, and btree_gist, and make the third test green by returning the ones the server offers. Think about which catalog answers the question at this point in the book, when the database is empty and nothing is installed yet.

Hint: pg_extension lists what is installed in this database; pg_available_extensions lists what the server could install. Only one of them can pass today, and understanding why is the exercise.

Exercise 1.3 — Challenge: Break the mount on purpose

The starter includes a compose file for a throwaway container on its own project name, its own volume, and port 5436, mounted the way every pre-18 compose file mounted it. Bring it up, read the exit, and fix the mount. Then make the two durability tests green against your book container: one asserts that the server's data directory really sits inside the mounted volume, and the other writes a marker row once and reads it back. Take the container down and up between two runs and watch the marker's timestamp refuse to move. Then take it down with docker compose down -v, the flag that removes the volume along with the container, and watch what that costs you.

Hint: SHOW data_directory tells you where the server actually keeps its files, and the answer names the major version. The marker table belongs in the ex_ch01 schema, and

the insert has to be written so that running it twice does not overwrite the first timestamp.

What's Next

The environment is running and the model is still empty, which is the right moment to decide what the model is. Chapter 2 opens the hood on DbContext: how long one should live and who hands it out, why LuminaWork pools all six of them and what pooling forbids in exchange, which of the three configuration surfaces owns which decision, and how value converters and comparers behave when the change tracker gets involved. No migration runs in Chapter 2. By its last page the model is finished and PostgreSQL still holds zero tables, which means Chapter 3 has nothing left to decide when it turns the model into schema.

Chapter 15 — Vector Search and Embeddings with pgvector

Chapter 12 taught the database to find words; this one teaches it to find meaning. Embeddings as a column type, distance as a query operator, an index that trades certainty for speed, and a search box that finally hears what people meant.

> **Real-World Scenario**
>
> *The screenshot that opened Chapter 12 came back, promoted. A facilities manager typed "hvac" into the search box LuminaWork now ships, and full-text search did exactly what it promises: it returned everything that says HVAC (one quarterly filter swap and its comment thread) and stopped. The monthly rooftop unit inspection, the row she was actually looking for and the row her own technicians would name first, was not in the list, because nobody who writes "rooftop unit" bothers to also write HVAC. Chapter 12 closed by capturing that exact miss and calling it the ceiling of lexical search.*
>
> *The fix is not a bigger synonym file. Somewhere between "rooftop unit" and "hvac" there is a fact about the world, namely that they are the same kind of work, and that fact lives in no dictionary the team will ever finish writing. It does live, statistically, in a language model. This chapter puts that model's judgment into a column, indexes it, queries it from LINQ in one SQL statement, and then fuses it with the lexical machinery so the search box answers with both kinds of evidence. The rooftop row gets found on these pages, captured, for the very intent lexical search failed three chapters ago.*

Two promissory notes come due. Chapter 1's stack figure named two provider plugins and promised each a chapter; Chapter 14 cashed the NetTopologySuite half and deliberately elided one line from its data-source listing — the pgvector registration, one chapter early. And Chapter 12's closing section promised that where lexical search stops being enough, semantic search picks up the thread. This chapter pays both, and closes Part III: documents, text, time, place, and now meaning.

Bring the database to this chapter's entry state with scripts\reset-to-chapter.ps1 -Chapter 15, and note that the entry state is, for the only time in this book, ahead of the narrative: the Ch15_Vectors migration is already applied and search.search_document.embedding already holds five thousand real vectors. Every other chapter applies its own migration on the page; this one cannot, because filling the column honestly requires an embedding model, and the harness never calls one — the vectors are computed once, offline, and committed to the repository with full provenance. The chapter still walks the mapping, the migration and the index as the teaching they are; the only thing you will not see is a migrate command, because the reset already ran it.

One quiet consequence deserves saying out loud. Since Chapter 12, every query in the Search module has projected DTOs because the entity mapped a column the database did not have; Chapters 12 through 14 worked in the model's past. As of this chapter's entry state the database has caught up with the frozen model, and materializing a SearchDocument is finally safe. The DTO discipline stays (it was the right shape anyway), but from here on it is a choice, not a constraint.

From matching words to measuring meaning

Here is the entire machine-learning theory this chapter needs. An embedding model is a function from text to a fixed-length array of floating-point numbers, trained so that texts with similar meaning map to nearby points. That is the whole contract. The model this book's corpus uses reads a task's title and body and returns 384 numbers; "Monthly rooftop unit inspection" and "hvac inspection" come out close together because the model has read enough of the internet to know rooftop units are HVAC equipment, and nothing about the letters in either string is involved. Distance in this space is a semantic judgment, precomputed and frozen into geometry.

The number 384 deserves one honest sentence, because readers meet vector columns of 384, 768, 1536 and 3072 in the wild and wonder which PostgreSQL wants. PostgreSQL wants none of them: the dimension is the embedding model's output width, full stop. MiniLM-class sentence models emit 384 floats; larger models emit more, cost more to store and compare, and are not automatically better at the "find the task where we fixed this before" problem a work-item catalogue actually has. The column type pins

whatever width you choose, and mixing widths, or models, in one column is meaningless even when the widths happen to agree.

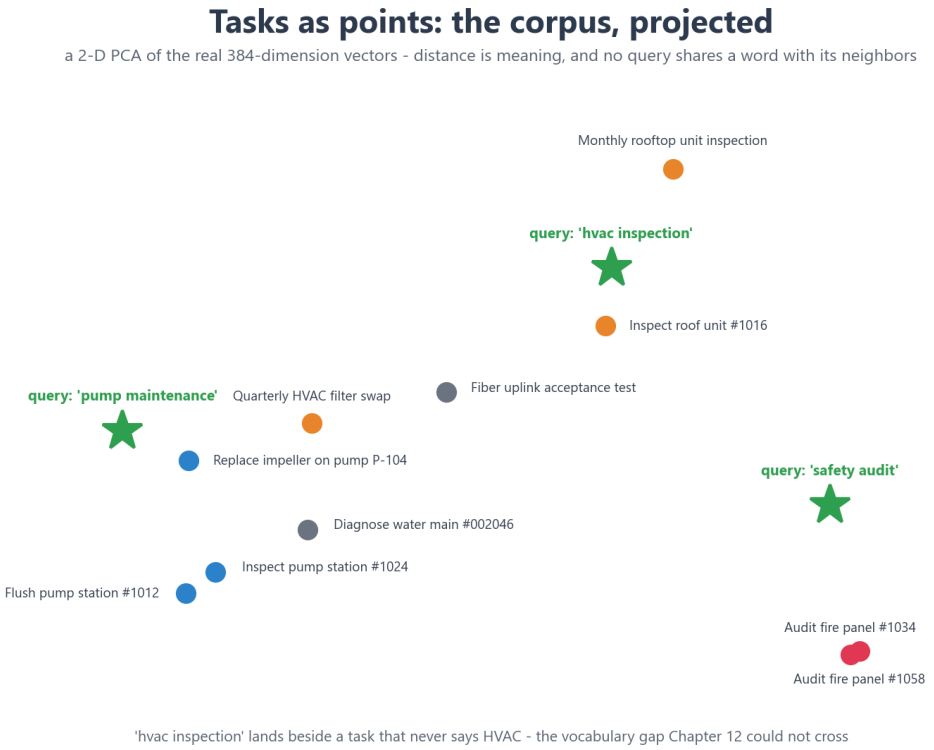

Figure 15.1 — The corpus, projected. A two-dimensional PCA of the real 384-dimension vectors behind this chapter: pump work clusters with pump work, the committed query vectors land beside the tasks they mean, and 'hvac inspection' sits next to a row that never says HVAC.

Figure 15.1 is computed from the actual vectors this chapter queries, not sketched: the projection flattens 384 dimensions to two, so it distorts distances the way any map distorts a globe, but the neighborhoods are real. Look at where the stars fall. Each star is a query string's own embedding, and each lands inside the cluster of tasks it is about, sharing not one word with most of them. That picture is the entire value proposition: search stops being string comparison and becomes nearest-neighbor lookup in a space where nearness was decided by a model that read the whole internet, and the database turns out to be very good at nearest-neighbor lookups.

> **Key Takeaway**
>
> *An embedding is a model's semantic judgment frozen into a fixed-width vector: similar meaning, nearby points. The dimension is the model's output width, not a PostgreSQL rule, and a vector is only comparable to vectors from the same model. Semantic search is nearest-neighbor search in that space — a geometry problem, which is why a database can be taught to do it.*

The plugin, the column, and the migration that was already applied

pgvector is to vectors what PostGIS is to shapes: an extension that adds a column type (vector, with the dimension as its typmod), a family of distance operators, and two index access methods. The book image ships it compiled and ready; CREATE EXTENSION vector is all a database needs, and the Search module has declared it on the model since Chapter 12 — HasPostgresExtension("vector") sat beside pg_trgm, waiting. Declared is not emitted: the declaration entered the model after Ch12_SearchDocuments was scaffolded, so that migration's snapshot never knew it, and its CREATE EXTENSION has been waiting for a migration to carry it; the annotation diff in Listing 15.5 is where it finally lands. On the .NET side the pairing is the Pgvector.EntityFrameworkCore plugin and its Vector CLR type, and after Chapter 14 the registration ceremony will feel like muscle memory. The EF half:

```
// UseVector() is the EF half of the pgvector plugin: it teaches the provider to
// translate CosineDistance and friends to the distance operators and to
// materialize vector columns as Pgvector.Vector. Registered here since Chapter 12
// - harmless while no mapped property needed it, exactly like UseNetTopologySuite.
public static void Configure(DbContextOptionsBuilder builder, string connectionString)
    => builder
        .UseNpgsql(connectionString, npgsql => npgsql
            .MigrationsHistoryTable("__efmh", "search")
            .UseVector())
        .UseSnakeCaseNamingConvention();
```

Listing 15.1 — The EF half of the plugin, on the Search context's provider options — in place since Chapter 12, doing nothing until a mapped Vector property gave it work.

And the wire half, on the seeder's raw data source — this is the exact listing Chapter 14 printed with one line elided, and the elision can finally come out. Any connection EF

does not own needs the registration repeated, the same per-connection-source rule MapEnum and UseNetTopologySuite already taught twice:

```
    public static NpgsqlDataSource Create(string connectionString)
    {
        var builder = new NpgsqlDataSourceBuilder(connectionString);
        // …
        builder.UseNetTopologySuite();
        builder.UseVector();
        return builder.Build();
    }
```

Listing 15.2 — The wire half, unelided at last: builder.UseVector() is the line Chapter 14's caption promised to this chapter, and without it the embedding loader could not bind a Vector parameter.

The entity has been carrying the column since the model froze, which is why Chapter 12 printed it as an excerpt and why every Search query since has projected DTOs. Here is the part the elision hid:

```
public class SearchDocument : ITenantOwned
{
    public Guid Id { get; set; }
    public Guid WorkspaceId { get; set; }
    // …
    public NpgsqlTsVector? Tsv { get; set; }

    /// <summary>
    /// vector(384) — MiniLM-class sentence embedding (D10). Null until the embedding
    /// pipeline has visited the row; the HNSW index skips nulls.
    /// </summary>
    public Vector? Embedding { get; set; }

    public DateTime CreatedAt { get; set; }
}
```

Listing 15.3 — The search document with its last member revealed: a nullable Vector beside the tsvector, because one projection table serves both kinds of search.

Nullable is a decision, not an accident, and it is the same decision the geography column made in Chapter 14: a document exists before its vector does. Embeddings arrive from a model (a network call or a local inference), and any pipeline that computes them can lag, fail, or be backfilled later; a null simply never matches a nearest neighbor. The configuration maps the property to the typmod-carrying column type:

```
    public void Configure(EntityTypeBuilder<SearchDocument> builder)
    {
        builder.ToTable("search_document");
        // …
        builder.Property(d => d.Embedding)
            .HasColumnType("vector(384)");
        // …
    }
```

Listing 15.4 — One HasColumnType call, one dimension pinned. The second elision is the HNSW index configuration, which this chapter unwraps in its indexing section.

The migration those mappings produced is Ch15_Vectors, and reading it is this chapter's version of applying it. Three moves: the vector extension joins pg_trgm in the database annotations, the column lands nullable, and the index arrives with an access method and an operator class the indexing section will take apart:

```
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AlterDatabase()
                .Annotation("Npgsql:PostgresExtension:pg_trgm", ",,")
                .Annotation("Npgsql:PostgresExtension:vector", ",,")
                .OldAnnotation("Npgsql:PostgresExtension:pg_trgm", ",,");

            migrationBuilder.AddColumn<Vector>(
                name: "embedding",
                schema: "search",
                table: "search_document",
                type: "vector(384)",
                nullable: true);

            migrationBuilder.CreateIndex(
                name: "ix_search_document_embedding",
                schema: "search",
                table: "search_document",
                column: "embedding")
                .Annotation("Npgsql:IndexMethod", "hnsw")
                .Annotation("Npgsql:IndexOperators", new[] { "vector_cosine_ops" });
        }
```

Listing 15.5 — Ch15_Vectors, read rather than run: CREATE EXTENSION via the annotation diff, a nullable vector(384) column, and an hnsw index with vector_cosine_ops — pre-applied by the harness because filling it honestly needs a model.

Trust, then verify. The catalog confirms what the reset already installed — \d search.search_document works too, but its generated-column definition folds badly at this page width, so a purpose-built query answers the two questions that matter:

```
SELECT a.attname AS column_name, format_type(a.atttypid, a.atttypmod) AS data_type
FROM pg_attribute AS a
WHERE a.attrelid = 'search.search_document'::regclass
  AND a.attnum > 0
ORDER BY a.attnum;

SELECT indexdef
FROM pg_indexes
WHERE schemaname = 'search'
  AND indexname = 'ix_search_document_embedding';
```

```
 column_name  |        data_type
--------------+-------------------------
 id           | uuid
 workspace_id | uuid
 source_kind  | character varying(16)
 source_id    | uuid
 title        | character varying(500)
 content      | text
 created_at   | timestamp with time zone
 tsv          | tsvector
 embedding    | vector(384)
(9 rows)

                                                 indexdef
-----------------------------------------------------------------------------------------------------------
        ----------
 CREATE INDEX ix_search_document_embedding ON search.search_document USING hnsw (embedding
        vector_cosine_ops)
(1 row)
```

Listing 15.6 — The column live with its typmod, and the index built USING hnsw with the cosine operator class. The database will now refuse a 383-float vector at the column boundary.

Version Note

Pgvector.EntityFrameworkCore 0.3.0 with Pgvector 0.3.2 on the .NET side; pgvector 0.8.6 in the book image, pinned by the image digest. The 0.8 line matters beyond the version string: it added iterative index scans, which this chapter's recall section leans on, so an older server extension would change what these pages can demonstrate.

Key Takeaway

vector(384) is a column type like any other: mapped with HasColumnType, migrated with AddColumn, its extension declared on the model, its plugin registered once per connection source — the EF half on the provider options, the wire half on any NpgsqlDataSourceBuilder EF does not own. The dimension in the typmod is enforced by the database, and null means not yet embedded, which is a state every real pipeline has.

Where the vectors come from

A vector column is only as meaningful as the pipeline that fills it, and this is the section where this book must be honest about having two pipelines. A production application embeds at write time: when a task is created or its text changes, something calls a model and stores the result. The book cannot do that — a chapter whose captures depend on a network call to a model API would never replay byte-identically, and shipping an 86-megabyte model inside a companion repository helps nobody. So the book computes its vectors once, offline, commits them, and loads them deterministically. Figure 15.2 draws both roads; they end at the same column.

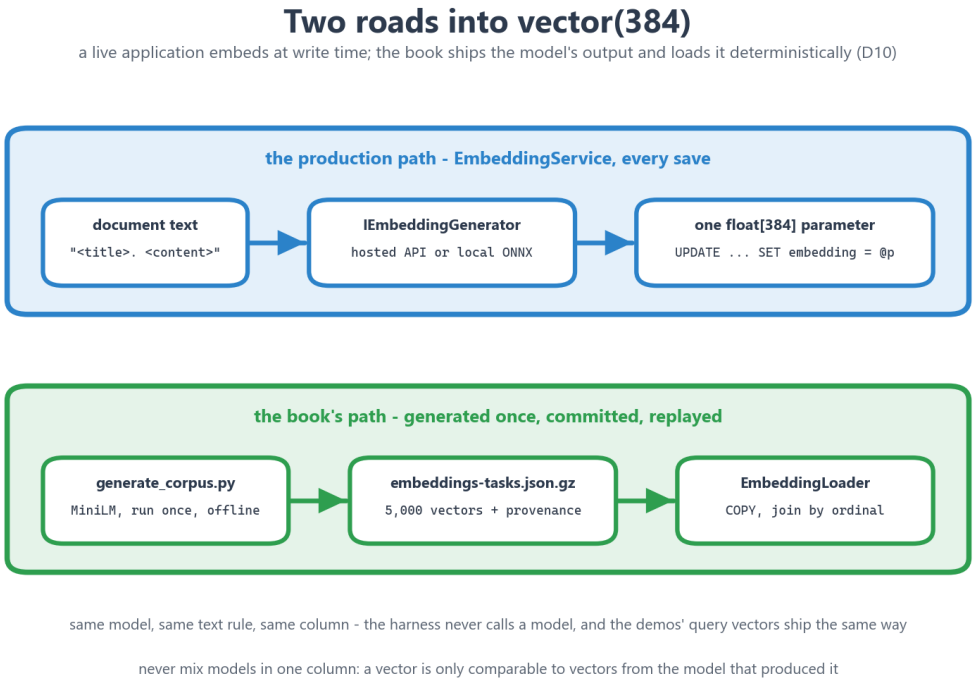

Figure 15.2 — Two roads into vector(384): the production write path through an IEmbeddingGenerator, and the book's committed corpus loaded by ordinal. Same model, same text rule, same column.

The production road first, because it is the one you will build. .NET's abstraction for embedding models is Microsoft.Extensions.AI's IEmbeddingGenerator — one interface whether the model is a hosted API or an ONNX file on your own CPU, which means the choice of model is configuration, not architecture:

```
    /// <summary>
    /// Embed one search document at write time. The text rule matches the corpus
    /// generator exactly: title and content as one short passage, title not repeated
    /// when the content already opens with it. Same model, same text rule, or the
    /// column silently mixes two geometries that cannot be compared.
    /// </summary>
    public async Task<Vector> EmbedAsync(
        string title, string content, CancellationToken ct = default)
    {
        var vector = await _generator.GenerateVectorAsync(
            DocumentText(title, content), options: null, cancellationToken: ct);
        return new Vector(vector);
    }

    /// <summary>
    /// The batch backfill: one call per page of documents, because every hosted
    /// generator meters by request and every local one amortizes tokenization.
    /// The generator returns embeddings in input order - the caller zips them back
    /// onto the rows it took the texts from.
    /// </summary>
    public async Task<IReadOnlyList<Vector>> EmbedBatchAsync(
        IReadOnlyList<(string Title, string Content)> documents,
        CancellationToken ct = default)
    {
        var texts = documents.Select(d => DocumentText(d.Title, d.Content));
        var embeddings = await _generator.GenerateAsync(
            texts, options: null, cancellationToken: ct);
        return embeddings.Select(e => new Vector(e.Vector)).ToList();
    }
```

Listing 15.7 — The production path: IEmbeddingGenerator behind EmbeddingService, write-time and batch. The harness never runs this class — that is a determinism doctrine, not a limitation of the code.

Write time or backfill is the first real design question a team meets, and the answer is usually both. Write-time embedding keeps the column fresh but puts a model call, with its latency, cost and failure modes, inside the save path, which is why mature systems queue it instead: the row commits with a null embedding, and a worker fills it moments later. The batch path is how you get started at all: five thousand existing tasks do not

embed themselves. LuminaWork's search projection already has the machinery this wants in the write-through sync from Chapter 12 — and Chapter 21's outbox projection, already promised for the search documents, is where the embedding refresh will ride reliably.

Now the book's road, which is also a story about reproducibility your own test suites can borrow. The committed corpus is the output of sentence-transformers/all-MiniLM-L6-v2 — real vectors from a real model, with the model name, revision hash, pooling, rounding and file hashes recorded in tools/embeddings/README.md, and two independent generation runs verified byte-identical. The loader's only job is to marry five thousand committed vectors to five thousand seeded rows, and the join key is a small piece of Chapter 5 paying off three hundred pages later:

```
    public static void Run(NpgsqlDataSource dataSource, string embeddingsFile)
    {
        // …
        var entries = ReadEntries(embeddingsFile);

        using var connection = dataSource.OpenConnection();

        using (var create = new NpgsqlCommand(
            """
            CREATE TEMP TABLE embedding_load
                (ordinal int PRIMARY KEY, embedding vector(384))
            """,
            connection))
        {
            create.ExecuteNonQuery();
        }

        using (var importer = connection.BeginBinaryImport(
            "COPY embedding_load (ordinal, embedding) FROM STDIN (FORMAT BINARY)"))
        {
            foreach (var entry in entries)
            {
                importer.StartRow();
                importer.Write(entry.Ordinal, NpgsqlDbType.Integer);
                importer.Write(new Vector(entry.Vector));
            }

            importer.Complete();
        }
```

```
        // Ordinal -> Nth work item by id -> that task's search document.
        using var update = new NpgsqlCommand(
            """
            UPDATE search.search_document d
            SET embedding = l.embedding
            FROM (
                SELECT w.id, row_number() OVER (ORDER BY w.id) AS ordinal
                FROM projects.work_item w
            ) ranked
            JOIN embedding_load l ON l.ordinal = ranked.ordinal
            WHERE d.source_kind = 'task'
              AND d.source_id = ranked.id
            """,
            connection);
        // …
    }
```

Listing 15.8 — The committed corpus loaded: binary COPY into a temp table (the wire half of the plugin at work), then one UPDATE joining ordinal to row. The Nth vector belongs to the Nth work item, forever.

That row_number join only works because "the Nth seeded work item" is a stable idea, and it is stable because Chapter 5 gave every entity family a disjoint counter range for its deterministic UUIDv7 ids. Core work items occupy a range far below the bulk range, ids sort by counter, and so ordinal 6 is the rooftop inspection on every machine that ever runs the seed — which is precisely what lets a file of five thousand vectors, computed once on my desk, land on the right five thousand rows on yours. When this chapter's demos filter to "the hand-written sixty", they do it with an id range comparison for the same reason. Determinism compounds.

Pro Tip

Pin the embedding model's name and revision in provenance you commit, the way this repository's tools/embeddings/README.md pins all-MiniLM-L6-v2 down to the file hashes. A vector column silently accepts vectors from any model with the right width, and two models' geometries are incommensurable even at the same dimension — mix them and every distance involving the minority model is noise. One column, one model, one recorded version; re-embed everything or embed nothing.

Key Takeaway

Embedding is a pipeline concern before it is a query concern: embed at write time through IEmbeddingGenerator, backfill in batches, and treat null as the honest "not yet" state. For

anything that must reproduce (a book, a test suite, a regression corpus), precompute the vectors, commit them with provenance, and load them deterministically; the join between committed vectors and seeded rows is only as stable as your id discipline.

Distance functions, from LINQ

Three distance operators matter, and the plugin exposes each as an extension method on Vector. CosineDistance measures the angle between vectors and ignores their length — for sentence embeddings this is the default choice, because meaning lives in direction. L2Distance is straight-line Euclidean distance. MaxInnerProduct is the dot product, negated by pgvector so that ascending order still means best-first. For vectors normalized to length one (this corpus is, and most sentence-embedding pipelines are) the three produce different numbers but identical rankings, related by exact algebra. The flagship query uses cosine, and its shape is the whole lesson:

```
    /// <summary>
    /// The flagship: nearest documents by meaning. CosineDistance translates to
    /// the cosine-distance operator, in the ORDER BY shape the HNSW index serves;
    /// the whole chain — filter, order, limit, projection — lands in ONE SQL
    /// statement. No id tie-break here, deliberately: a secondary sort key would
    /// cost the plan its index-ordered walk, and real vectors tie only when the
    /// text itself does.
    /// </summary>
    public static IQueryable<SemanticHit> Nearest(
        SearchDbContext db, Vector query, int take) =>
        db.Documents
            .Where(d => d.Embedding != null)
            .OrderBy(d => d.Embedding!.CosineDistance(query))
            .Take(take)
            .Select(d => new SemanticHit(
                d.SourceKind,
                d.SourceId,
                d.Title,
                d.Embedding!.CosineDistance(query)));
```

Listing 15.9 — The flagship semantic query: filter, order by distance, take, project. The Vector parameter is the user's embedded words; everything else is ordinary composable LINQ.

Chapter 14 ended its index section with a warning shot: OrderBy(Distance) on geography translated to ST_Distance and left the KNN operator on the table, so the obvious question here is whether pgvector's plugin does better. Measured against the shipped 0.3.0 assembly, it does — the demo below captures EF Core's own log, and the

statement is everything the spike promised. One round trip, the distance operator in both the projection and the ORDER BY, the vector travelling as a parameter the log redacts, and the named Tenant filter riding along unasked:

```
-- the flagship: 1 statement, vector bound as a parameter
Executed DbCommand (...ms) [Parameters=[@query='?' (DbType = Object),
        @ef_filter__TenantId='?' (DbType = Guid), @p='?' (DbType = Int32)],
        CommandType='Text', CommandTimeout='30']

SELECT s.source_kind, s.source_id, s.title, s.embedding <=> @query
FROM search.search_document AS s
WHERE s.workspace_id = @ef_filter__TenantId AND s.embedding IS NOT NULL
ORDER BY s.embedding <=> @query
LIMIT @p
```

Listing 15.10 — The flagship translated, from the EF log: ONE statement, the <=> operator twice, @query redacted in the parameter header (a 384-float vector never belongs in your logs), and the tenant predicate applied by the named filter.

Read the parameter header the way an operator would. DbType Object is the same tell Chapter 14 explained for geography: a vector has no slot in ADO.NET's classic type list, so the plugin's own mapping carries it, and the question mark is EF Core's default refusal to put parameter values in the log. Then the execution, and here the chapter owes you its first honesty about approximation. These executed captures pin the planner to an exact scan, because the HNSW index built by your reset and the one built by mine are different graphs that return slightly different candidates — the indexing section demonstrates this properly. Exact means every distance is computed and the five smallest win, which at a few thousand vectors costs milliseconds:

```
-- executed exact (index scans off for print determinism - the recall lab says why)
-- top 5, whole catalogue: a seeded catalogue repeats itself
  0.4697  task  Inspect pump station #1024
  0.4748  task  Inspect pump station #001584
  0.4804  task  Inspect pump station #001728
  0.4807  task  Inspect pump station #001704
  0.4824  task  Inspect pump station #002664

-- top 5 among the hand-written sixty (id range, Chapter 5)
  0.4697  task  Inspect pump station #1024
  0.5142  task  Replace impeller on pump P-104
  0.5253  task  Flush pump station #1012
  0.6391  task  Quarterly HVAC filter swap - Sidi Maarouf DC
  0.6794  task  Service compressor #1017
```

```
documents with embeddings       2502
documents in the workspace      50096
```

Listing 15.11 — Executed, exact. The whole catalogue answers 'pump maintenance' with one pump station five times (a seeded catalogue repeats itself by construction), while the hand-written sixty answer with five genuinely different pieces of pump work.

Both result sets teach. The top block is the catalogue being honest about itself: the bulk seeder builds titles from eight verbs and twelve assets, so 4,940 of the five thousand documents share twenty-four title shapes, and a nearest-neighbor query over near-duplicates returns near-duplicates — real production catalogues do this too, which is why the exercises build a dedup-aware top-k. The bottom block is the payoff on real variety: scoped to the sixty hand-written tasks, "pump maintenance" finds an inspection, an impeller replacement, a flush and a compressor service, ranked by meaning, none of them containing the word maintenance. The distances are doing exactly what Figure 15.1 promised.

The variant methods complete the toolbox, and their real content is the caveat in each comment:

```
    public static IQueryable<SemanticHit> NearestByL2(
        SearchDbContext db, Vector query, int take) =>
        db.Documents
            .Where(d => d.Embedding != null)
            .OrderBy(d => d.Embedding!.L2Distance(query))
            .Take(take)
            .Select(d => new SemanticHit(
                d.SourceKind, d.SourceId, d.Title, d.Embedding!.L2Distance(query)));

    /// <summary>
    /// Inner product, negated by pgvector so that ORDER BY ascending still means
    /// best-first. Only meaningful when vectors are normalized - then it ranks
    /// exactly like cosine and saves the normalization arithmetic.
    /// </summary>
    public static IQueryable<SemanticHit> NearestByInnerProduct(
        SearchDbContext db, Vector query, int take) =>
        db.Documents
            .Where(d => d.Embedding != null)
            .OrderBy(d => d.Embedding!.MaxInnerProduct(query))
            .Take(take)
            .Select(d => new SemanticHit(
                d.SourceKind, d.SourceId, d.Title, d.Embedding!.MaxInnerProduct(query)));
```

Listing 15.12 — The other two rulers: L2Distance and MaxInnerProduct translate to their operators the same way. On normalized vectors all three rank identically — the first exercise proves the algebra.

> **⚠ Warning**
>
> *Distance values are relative, never absolute. 0.47 was this corpus's best answer for one query and would be a poor one for another; a different model moves every number; and there is no universal threshold under which a match is "good". What survives all of that is the ordering, so build features on "the closest k" and treat any hard distance cutoff as a tuned, model-specific hint that expires with the model — and never paginate deep into nearest-neighbor results, because page forty of an approximate index scan is noise sorted confidently.*

> **Key Takeaway**
>
> *CosineDistance, L2Distance and MaxInnerProduct translate to pgvector's three operators, compose with Where, Take and Select into one SQL statement, and carry the query vector as one parameter. Choose cosine for sentence embeddings, know that normalization makes the three agree on order, and build on ranks, not on distance thresholds.*

The HNSW index, and recall you must measure

An exact nearest-neighbor query reads every vector, and five thousand vectors forgive that. Five million do not: distance math over 384 dimensions is real arithmetic, and no B-tree can help, because "near in 384 dimensions" is not an order a sorted structure can precompute. pgvector's answer is the HNSW index the migration already built, and its configuration is the elision Listing 15.4 left behind:

```
        // HNSW over cosine distance: approximate nearest neighbor in graph time
        // (Chapter 15). The operator class must match the query operator (<=>).
        builder.HasIndex(d => d.Embedding)
            .HasMethod("hnsw")
            .HasOperators("vector_cosine_ops");
```

Listing 15.13 — The index in the model: HasMethod("hnsw"), the same lever GIN and GiST pulled, plus an operator class that must match the query's operator — or the index serves nothing.

HNSW stands for Hierarchical Navigable Small World, which unpacks into Figure 15.3: a stack of graphs, sparse at the top and complete at the bottom. Every vector is a node in layer 0, connected to its nearest neighbors; a random few are promoted into higher

layers whose long edges cross the whole space in a hop. A search enters at the top, greedily walks toward the query, and drops a layer each time it can get no closer — coarse steering first, fine neighborhood last, the same trick a skip list plays on a linked list. The result is a lookup that touches a few hundred nodes instead of five million.

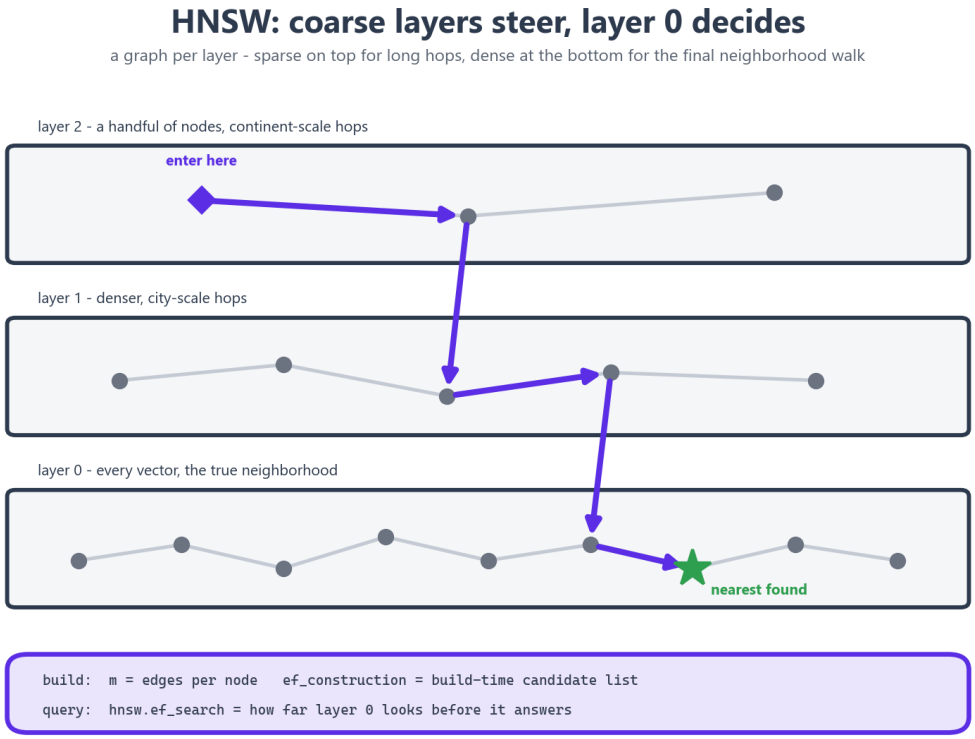

Figure 15.3 — The HNSW descent: enter at a sparse top layer, walk greedily toward the query, drop down when stuck, and let layer 0 decide. m and ef_construction shape the graph at build time; hnsw.ef_search decides how far layer 0 looks at query time.

Two families of dials fall out of that structure. At build time, m fixes how many edges each node keeps and ef_construction how many candidates the builder examines before choosing them — the migration's index accepts both as WITH options and this book's spine uses the defaults, sixteen and sixty-four. At query time, hnsw.ef_search sets how large a candidate list the layer-0 walk maintains before the best k are returned. Here is the index actually engaging, with the dial set and the plan shape on the page; the probe is a stored row's own embedding standing in for a query vector, because a 384-float literal cannot print, and the application binds it as a parameter anyway:

```
\pset tuples_only on
SET max_parallel_workers_per_gather = 0;
SET hnsw.ef_search = 200;

EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF, SUMMARY OFF, BUFFERS OFF)
SELECT title
FROM search.search_document
WHERE embedding IS NOT NULL
ORDER BY embedding <=> (
    SELECT embedding FROM search.search_document
    WHERE source_kind = 'task' AND title = 'Replace impeller on pump P-104')
LIMIT 5;
```

```
SET
SET
 Limit (actual rows=5.00 loops=1)
   InitPlan 1
     ->  Bitmap Heap Scan on search_document search_document_1 (actual rows=1.00 loops=1)
           Recheck Cond: ((title)::text = 'Replace impeller on pump P-104'::text)
           Rows Removed by Index Recheck: 2
           Filter: ((source_kind)::text = 'task'::text)
           Heap Blocks: exact=2
           ->  Bitmap Index Scan on ix_search_document_title (actual rows=3.00 loops=1)
                 Index Cond: ((title)::text = 'Replace impeller on pump P-104'::text)
                 Index Searches: 1
   ->  Index Scan using ix_search_document_embedding on search_document (actual rows=5.00
loops=1)
         Order By: (embedding <=> (InitPlan 1).col1)
         Filter: (embedding IS NOT NULL)
         Index Searches: 1
```

Listing 15.14 — The HNSW scan engaged: Order By on the index scan itself and no Sort node anywhere (the graph walk IS the ordering), with the probe fetched once by an InitPlan. ANN plans look like this or they are not using the index.

No Sort node is the shape to memorize: the index hands rows out already ordered by distance and the Limit simply stops it, which is why a KNN query can answer in milliseconds at any scale. And now the price, which this book refuses to soften. Approximate means the graph walk can miss, and this corpus (4,940 near-duplicates crowding twenty-four points of the space) turns out to be an adversarial recall benchmark by accident. Measured during authoring, on the graph one particular reset built: the default walk answered "pump maintenance" with nine of the true top ten, missing the true nearest row; for "hvac inspection" under the tenant filter it returned nothing at all, because every candidate the walk surfaced belonged to the other workspace and the filter discarded them; and raising ef_search alone did not rescue it. The companion repository's ch15-probe demo replays that grading against whatever graph your machine builds, and the point of running it is that your numbers will differ:

HNSW assigns node levels randomly, so every build is a different graph with different misses. Recall is a property of the index you actually built, measured against your data — never a number to copy out of a book, this one included.

The dials, in the order to reach for them. hnsw.iterative_scan, new in pgvector 0.8, fixes the filtered-starvation case: instead of surrendering after one candidate batch, the scan keeps walking until enough rows survive the filter, and in the authoring measurements it reliably turned zero-row answers into full ones. ef_search trades latency for a wider look at an otherwise-good graph. When the graph itself learned the wrong shape, only a rebuild helps: the ch15-rebuild lab reconstructs the index at ef_construction = 200 inside a rolled-back transaction, and on most runs the rebuilt graph recovers the missed rows — on most runs, because each rebuild rolls new dice, which is the lab's real lesson. Measure, dial, re-measure; and when a filtered semantic query simply must be exact and the candidate set is small, an exact scan is not defeat, it is arithmetic.

Going Deeper

Why can a whole workspace vanish from an ANN answer? The bulk seeder assigns workspaces alternately, and title shapes repeat every twenty-four rows, so each near-duplicate cluster of ~200 rows belongs entirely to ONE workspace. A query vector that lands near the wrong tenant's cluster hands the walk a candidate list the filter then annihilates. Production data does the same thing whenever a selective predicate correlates with position in embedding space — one tenant's jargon, one product line's vocabulary. That correlation is why pgvector grew iterative scans, and why filtered ANN recall deserves its own measurement, separate from unfiltered recall.

IVFFlat is the other index pgvector offers, and the fairest way to meet it is inside a transaction that cannot damage the spine. Where HNSW builds a graph, IVFFlat runs k-means over the existing vectors and files each one under its nearest centroid; a query visits the ivfflat.probes nearest lists and reads only those. That design has one hard prerequisite the fragment below encodes: the data must exist before the index, because clustering an empty table yields useless centroids.

```
\pset tuples_only on
SET LOCAL max_parallel_maintenance_workers = 0;
SET LOCAL max_parallel_workers_per_gather = 0;
```

```
-- An experiment, not a change: the harness rolls this back, and the
-- spine keeps its HNSW index.
DROP INDEX search.ix_search_document_embedding;

CREATE INDEX ix_search_document_ivfflat ON search.search_document
    USING ivfflat (embedding vector_cosine_ops) WITH (lists = 50);

SET LOCAL ivfflat.probes = 10;

EXPLAIN (COSTS OFF)
SELECT title
FROM search.search_document
WHERE embedding IS NOT NULL
ORDER BY embedding <=> (
    SELECT embedding FROM search.search_document
    WHERE source_kind = 'task' AND title = 'Replace impeller on pump P-104')
LIMIT 5;
```

```
SET
SET
DROP INDEX
CREATE INDEX
SET
 Limit
   InitPlan 1
     ->  Bitmap Heap Scan on search_document search_document_1
           Recheck Cond: ((title)::text = 'Replace impeller on pump P-104'::text)
           Filter: ((source_kind)::text = 'task'::text)
           ->  Bitmap Index Scan on ix_search_document_title
                 Index Cond: ((title)::text = 'Replace impeller on pump P-104'::text)
   ->  Index Scan using ix_search_document_ivfflat on search_document
         Order By: (embedding <=> (InitPlan 1).col1)
         Filter: (embedding IS NOT NULL)
```

Listing 15.15 — IVFFlat, built and engaged inside a rolled-back transaction: k-means lists at build time, probes at query time, the same no-Sort plan shape. lists = 50 and probes = 10 are choices, not defaults, and the benchmark uses the same pair.

Which one, then? The honest comparison is measured, and the benchmark methods are short enough to read whole — build each index over the corpus, then run the same top-ten query under each, with the exact scan as the control every ANN claim should be graded against:

```
    [Benchmark]
    public void BuildHnsw() => Execute(HnswDdl);

    [Benchmark]
    public void BuildIvfflat() => Execute(IvfflatDdl);
```

```
    [Benchmark(Baseline = true)]
    public int SearchExact() => RunTopTen();

    [Benchmark]
    public int SearchHnsw() => RunTopTen();

    [Benchmark]
    public int SearchIvfflat() => RunTopTen();
```

Listing 15.16 — The benchmark surface: two builds, three searches, one committed run. IVFFlat exists only inside this benchmark and the fragment above — the spine model never maps it.

Method	Mean
BuildHnsw	899.05 ms
BuildIvfflat	181.37 ms
SearchExact	15.09 ms
SearchHnsw	860.3 us
SearchIvfflat	1.10 ms

Measured on Intel Core i7-4790 CPU 3.60GHz (Haswell), .NET 10.0.11, Windows 11, 2026-08-27. Read the relationships, not the absolute figures, which belong to that machine on that afternoon. The exact scan is the control, and its row is the one to respect: every ANN claim in this chapter is a claim about beating that number while returning almost the same rows.

Three relationships survive any hardware. IVFFlat builds several times faster than HNSW (k-means over five thousand vectors is cheap, graph construction is not), and that gap widens with corpus size, which is why bulk-loading pipelines sometimes build IVFFlat first and swap later. Both indexes answer the top-ten query an order of magnitude faster than the exact scan, at five thousand vectors; at five million the exact bar would leave the chart. And HNSW edges out IVFFlat on query latency while holding recall better as data drifts, because k-means centroids describe yesterday's distribution

while a graph absorbs inserts incrementally. That last property is why the spine ships HNSW: LuminaWork's catalogue grows row by row, and an index that degrades gracefully under growth beats one that wants periodic re-clustering. IVFFlat still wins when you bulk-load a stable, warehouse-style corpus and rebuild on a schedule — and the fragment above is exactly the experiment to run when you suspect you are in that world.

> **Key Takeaway**
>
> *HNSW is a layered graph: no Sort node, milliseconds at any scale, recall decided by the graph your build happened to produce. Measure recall against an exact scan on your own data; reach for iterative_scan when filters starve results, ef_search when the graph is good but the look is narrow, and ef_construction when the graph itself is the problem. IVFFlat builds far faster and suits stable bulk-loaded corpora; HNSW absorbs growth, which is why LuminaWork ships it.*

Hybrid search: fusing words and meaning

Semantic search is not a replacement for Chapter 12, and the "safety audit" query proves it in one sentence: the only document in the demo workspace containing both words is a comment, and comments have no embeddings — the semantic lane is structurally deaf to them, while the lexical lane finds exact identifiers, part numbers and rare words with a precision no embedding matches. Meaning finds what words miss; words find what meaning cannot see. A production search box wants both, and the standard way to combine two ranked lists whose scores share no scale is reciprocal rank fusion. Build it stepwise. First, each lane as a CTE that reduces its own scoring to positions:

```
-- The word lane: Chapter 12's machinery, reduced to positions.
WITH by_words AS (
    SELECT source_id, title,
           row_number() OVER (
               ORDER BY ts_rank(tsv, websearch_to_tsquery('english', 'fire panel')) DESC,
                        title, source_id) AS lane_rank
    FROM search.search_document
    WHERE workspace_id = '019bc0e2-1281-7cc1-9e41-ab087439611e'
      AND tsv @@ websearch_to_tsquery('english', 'fire panel')
    ORDER BY lane_rank
    LIMIT 40
```

```
)
SELECT lane_rank, title FROM by_words LIMIT 5;

-- The meaning lane: this chapter's, the probe standing in for the
-- user's query vector.
WITH by_meaning AS (
    SELECT source_id, title,
           row_number() OVER (
               ORDER BY embedding <=> (
                   SELECT embedding FROM search.search_document
                   WHERE source_kind = 'task' AND title = 'Audit fire panel #1058'),
               source_id) AS lane_rank
    FROM search.search_document
    WHERE workspace_id = '019bc0e2-1281-7cc1-9e41-ab087439611e'
      AND embedding IS NOT NULL
    ORDER BY lane_rank
    LIMIT 40
)
SELECT lane_rank, title FROM by_meaning LIMIT 5;
```

```
 lane_rank |          title
-----------+-------------------------
         1 | Audit fire panel #000010
         2 | Audit fire panel #000034
         3 | Audit fire panel #000058
         4 | Audit fire panel #000082
         5 | Audit fire panel #000106
(5 rows)

 lane_rank |           title
-----------+-------------------------
         1 | Audit fire panel #1034
         2 | Diagnose fire panel #1022
         3 | Audit fire panel #001330
         4 | Audit fire panel #002818
         5 | Audit fire panel #001354
(5 rows)
```

Listing 15.17 — The two lanes side by side, each fetching forty candidates and keeping only positions. The probe row belongs to the other tenant, so it cannot appear in its own results — the workspace predicate holds even here.

Three deliberate details before the join. Each lane over-fetches (forty candidates for a final list of eight) because a row ranked thirtieth in both lanes may still beat a row ranked first in one, and a lane cut to the final size cannot know that. Each lane carries the deterministic tie-break this book has insisted on since Chapter 8, and here it is doing heavy lifting: ts_rank scores one identical value for every fire-panel title, so without the tie-break the word lane's order would be whatever the heap felt like. And row_number is the great equalizer — once each lane is positions, ts_rank's arbitrary

scale and cosine's bounded one stop mattering, which also means you can swap either lane's scoring function without touching the fusion. The join:

```
WITH by_words AS (
    SELECT source_id, title,
           row_number() OVER (
               ORDER BY ts_rank(tsv, websearch_to_tsquery('english', 'fire panel')) DESC,
                        title, source_id) AS lane_rank
    FROM search.search_document
    WHERE workspace_id = '019bc0e2-1281-7cc1-9e41-ab087439611e'
      AND tsv @@ websearch_to_tsquery('english', 'fire panel')
    ORDER BY lane_rank
    LIMIT 40
),
by_meaning AS (
    SELECT source_id, title,
           row_number() OVER (
               ORDER BY embedding <=> (
                   SELECT embedding FROM search.search_document
                   WHERE source_kind = 'task' AND title = 'Audit fire panel #1058'),
                   source_id) AS lane_rank
    FROM search.search_document
    WHERE workspace_id = '019bc0e2-1281-7cc1-9e41-ab087439611e'
      AND embedding IS NOT NULL
    ORDER BY lane_rank
    LIMIT 40
)
SELECT round(COALESCE(1.0 / (60 + w.lane_rank), 0)
     + COALESCE(1.0 / (60 + m.lane_rank), 0), 4) AS fused,
       w.lane_rank AS words,
       m.lane_rank AS meaning,
       COALESCE(w.title, m.title) AS title
FROM by_words AS w
FULL OUTER JOIN by_meaning AS m USING (source_id)
ORDER BY fused DESC, title, source_id
LIMIT 8;
```

```
 fused  | words | meaning |          title
--------+-------+---------+-------------------------
 0.0311 |     1 |       8 | Audit fire panel #000010
 0.0304 |     3 |       9 | Audit fire panel #000058
 0.0263 |    11 |      22 | Audit fire panel #000250
 0.0256 |    36 |       6 | Audit fire panel #000850
 0.0215 |    31 |      35 | Audit fire panel #000730
 0.0213 |    37 |      31 | Audit fire panel #000874
 0.0164 |       |       1 | Audit fire panel #1034
 0.0161 |     2 |         | Audit fire panel #000034
(8 rows)
```

Listing 15.18 — Reciprocal rank fusion: FULL OUTER JOIN so either lane alone can place a row, COALESCE so a missing lane contributes zero, and 1/(60+rank) turning positions into scores.

Now the captured table explains the famous constant better than any citation. With k = 60, appearing in one lane at all is worth at most 1/61 ≈ 0.0164, and the gap between a lane's first and second place is 1/61 minus 1/62 — about 0.0003, under two percent of the score. Presence is nearly everything; position within a lane is a gentle nudge. You can read that policy straight off the output: ranks 1 and 8 across the two lanes fused to 0.0311, ranks 31 and 35 (mediocre twice) fused to 0.0215, and the meaning lane's own champion, ranked first but found by one lane only, sits below both at 0.0164. Shrink k and lane winners start to dominate; grow it and fusion flattens toward counting lanes. Sixty is the literature's default, and now you know exactly what it buys. Figure 15.4 draws the same fusion as one picture.

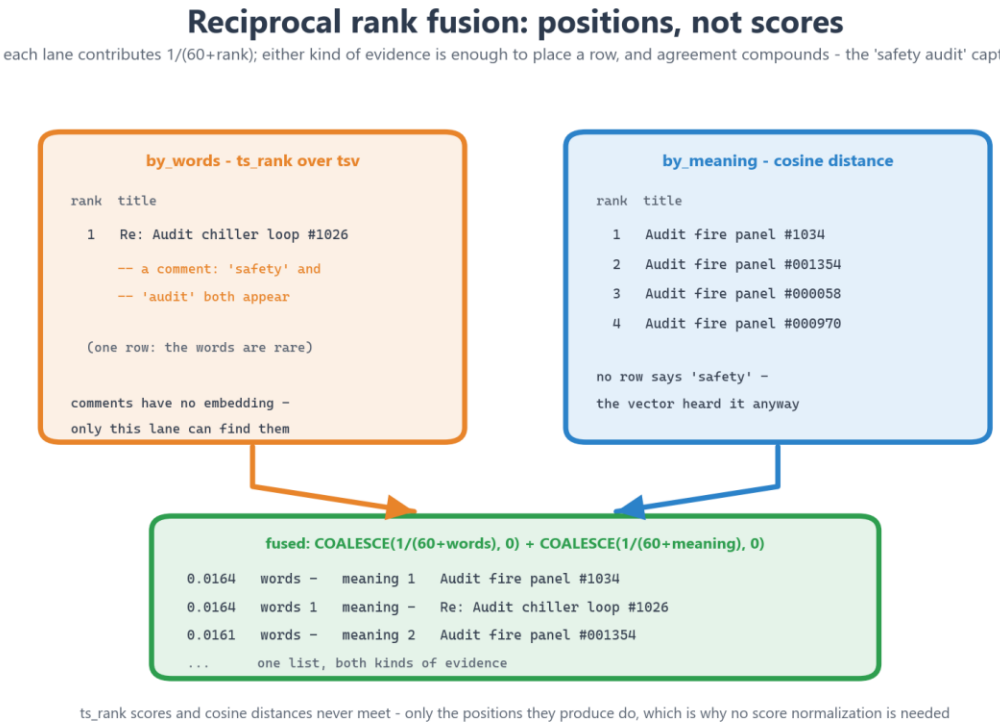

Figure 15.4 — The fusion, drawn from the 'safety audit' capture: the word lane contributes a comment no vector will ever find, the meaning lane contributes tasks that never say safety, and 1/(60+rank) merges the two kinds of evidence without comparing their scores.

From EF Core, this query is exactly what Chapter 9's raw-SQL escape hatch exists for: two window functions and a FULL OUTER JOIN are beyond LINQ, and SqlQuery composes them with parameters instead of literals. One subtlety earns the comment:

SqlQuery bypasses the model, so the named Tenant filter that has silently guarded every query since Chapter 9 does not apply here, and the workspace predicate returns to being your job, written once per lane:

```
    /// <summary>
    /// SqlQuery bypasses the model, so the named Tenant filter does NOT ride along
    /// here — the workspace predicate is written out, once per lane. The query
    /// vector travels as one parameter; ts_rank scores and cosine distances never
    /// meet, only the positions they produce.
    /// </summary>
    public static async Task<IReadOnlyList<HybridHit>> SearchAsync(
        SearchDbContext db,
        Guid workspaceId,
        string terms,
        Vector query,
        int take = 10,
        int laneDepth = 40,
        CancellationToken ct = default)
    {
        return await db.Database.SqlQuery<HybridHit>($"""
        WITH by_words AS (
            SELECT source_id, title,
                   row_number() OVER (
                       ORDER BY ts_rank(tsv, websearch_to_tsquery('english', {terms}))
DESC,
                                title, source_id) AS lane_rank
            FROM search.search_document
            WHERE workspace_id = {workspaceId}
              AND tsv @@ websearch_to_tsquery('english', {terms})
            ORDER BY lane_rank
            LIMIT {laneDepth}
        ),
        by_meaning AS (
            SELECT source_id, title,
                   row_number() OVER (
                       ORDER BY embedding <=> {query}, source_id) AS lane_rank
        // …
        """)
            .ToListAsync(ct);
    }
```

Listing 15.19 — The fusion from EF Core: SqlQuery<HybridHit> with every interpolation hole becoming a parameter — the terms twice, the workspace, the vector, the depths. The elided half is the fusion SELECT Listing 15.18 already printed.

One implementation note, measured because it surprised me: the meaning lane inside this SQL runs exact. The planner cannot feed a window function's ORDER BY from the HNSW scan (the captured plan shows a Sort over the workspace index), so at this

corpus size the lane costs milliseconds and returns the true ranking. At real scale you would restructure the lane around an index-served inner query with LIMIT before ranking; the candidate-depth parameter is already where that change lands. And with that, the promise from three chapters ago comes due:

```
-- the lexical lane alone: 'hvac inspection'
  (no rows)

-- the semantic lane alone: the committed 'hvac inspection' vector, exact top 5
  0.5212  Monthly rooftop unit inspection - Tower A
  0.5767  Inspect conveyor #1032
  0.5792  Quarterly HVAC filter swap - Sidi Maarouf DC
  0.6108  Inspect roof unit #1016
  0.6560  Repaint air handler #1013

-- fused: reciprocal rank fusion, k = 60
  score   words meaning  title
  0.0164      -      1   Monthly rooftop unit inspection - Tower A
  0.0161      -      2   Inspect conveyor #1032
  0.0159      -      3   Quarterly HVAC filter swap - Sidi Maarouf DC
  0.0156      -      4   Inspect roof unit #1016
  0.0154      -      5   Repaint air handler #1013

-- 'safety audit': the lexical lane rescues what semantics cannot see
  score   words meaning  title
  0.0164      -      1   Audit fire panel #1034
  0.0164      1      -   Re: Audit chiller loop #1026
  0.0161      -      2   Audit fire panel #001354
  0.0159      -      3   Audit fire panel #000058
  0.0156      -      4   Audit fire panel #000970
```

Listing 15.20 — The payoff, captured: 'hvac inspection' finds nothing lexically, the semantic lane leads with the rooftop inspection Chapter 12 proved unfindable, the fusion returns it first — and 'safety audit' shows the word lane rescuing a comment semantics cannot see.

Listing 12.17 captured the rooftop inspection being invisible to a search for "hvac"; Listing 15.20 captures the same row arriving first, from the same database, for the same user intent. The vocabulary gap did not close because anyone wrote a synonym — it closed because a model's judgment now lives in a column, four hundred pages of infrastructure deep. That is the milestone Part III was building toward: semantic and hybrid search over the tasks, in the same database, inside the same transactions, behind the same tenant discipline as everything else.

> **Key Takeaway**
>
> *Hybrid search fuses evidence, not scores: each lane keeps its own ranking, positions meet through 1/(60+rank), and either lane alone can put a row on the board. Vectors live best next to the rows they describe: one database, one transaction, one query — which is what makes the fusion a single SQL statement instead of a distributed system.*

Where this book stops

Everything on these pages searched one short document per task. The moment your documents are long (manuals, transcripts, contract PDFs) a single vector per document stops working, and a family of new problems arrives: chunking text into embeddable passages without severing meaning, reranking a cheap wide retrieval with a more expensive model, and wiring retrieval into generation so a language model answers from your data instead of its memory. Those are retrieval-augmented generation problems, they are architecture rather than database features, and they deserve a book that starts where this chapter ends rather than a section that waves at them.

> **Cross-Reference**
>
> *That book exists in this series: Production-Grade RAG with C# and .NET builds the full pipeline (chunking, embedding, hybrid retrieval, reranking, generation and evaluation) on the same PostgreSQL foundations you now have. What you built here IS its retrieval layer: the fusion query on these pages is a production hybrid retriever, and everything past it is what you do with the rows it returns.*

Summary

You closed the vocabulary gap that Chapter 12 could only demonstrate. An embedding turned out to need one page of theory (text to vector, meaning to nearness, dimension set by the model and nobody else), and the rest was engineering you already knew how to judge: a plugin with an EF half and a wire half, a vector(384) column mapped with one HasColumnType call, and a migration the harness pre-applied because filling the column honestly requires a model the book refuses to call at build time. The committed corpus carries full provenance, and the ordinal join that lands five thousand vectors on the right five thousand rows is Chapter 5's id discipline collecting its debt.

The flagship query composed Where, OrderBy(CosineDistance) and Take into one captured SQL statement with the vector as a redacted parameter, answering the question Chapter 14's geography translation left open. The HNSW index showed its no-Sort plan shape and then its price: on a catalogue that repeats itself by construction, the graph walk missed real answers, starved entirely under a tenant filter, and graded differently on every rebuild — which is why the chapter pinned its printed executions to exact scans, shipped the probe and rebuild labs instead of a recall table, and taught iterative_scan, ef_search and ef_construction as dials you turn after measuring, never before. The benchmark put honest ratios behind the folklore: IVFFlat builds several times faster, both indexes answer an order of magnitude quicker than exact, and the spine ships HNSW because it absorbs growth.

Hybrid search closed the loop: two lanes reduced to positions, fused by 1/(60+rank) through a FULL OUTER JOIN, composed from EF Core with SqlQuery — where the named tenant filter does not follow you, and the workspace predicate is yours again. The rooftop inspection that opened Chapter 12's closing argument arrived first in the fused results, and a comment only words could find arrived beside it. Part III is complete: documents, text, arrays and ranges, place, and meaning, all in the one database LuminaWork already operates.

Key Terms

- **embedding** — a model's mapping from text to a fixed-width float vector, built so similar meaning lands at nearby points. The model decides the dimension; the column just stores it.
- **pgvector** — the extension that makes vectors a PostgreSQL column type: vector(n), three distance operators, and the HNSW and IVFFlat access methods. Paired in .NET with Pgvector.EntityFrameworkCore and its Vector type.
- **vector(384)** — the column type with its dimension as typmod, enforced at the boundary; 384 is MiniLM's output width, not a PostgreSQL preference.
- **cosine distance** — one minus the cosine of the angle between vectors; direction-only, the default ruler for sentence embeddings. CosineDistance in LINQ, served by vector_cosine_ops.

- **CosineDistance / L2Distance / MaxInnerProduct** — the plugin's three extension methods, translating to pgvector's three operators; on normalized vectors all three rank identically.
- **approximate nearest neighbor (ANN)** — trading certainty for speed: an index that returns almost the closest rows, almost always. The 'almost' is recall, and it is measured, not promised.
- **HNSW** — the layered navigable-graph index: sparse top layers steer, layer 0 decides, no Sort node in the plan. Built with random level assignment, so every build is a slightly different graph.
- **ef_search** — the query-time candidate-list width of the layer-0 walk; hnsw.iterative_scan is its 0.8-era sibling that keeps walking until a filter is satisfied.
- **IVFFlat** — the k-means index: vectors filed under centroids, probes lists visited per query. Builds far faster than HNSW; wants existing data and periodic re-clustering.
- **recall** — the fraction of the true top k an ANN answer actually contains, graded against an exact scan. A property of your index on your data, and this chapter's central measurement.
- **hybrid search** — lexical and semantic lanes answering the same query, fused into one list; each lane finds what the other structurally cannot.
- **reciprocal rank fusion (RRF)** — scoring a row as the sum of 1/(k+rank) across lanes, k = 60 by convention: presence in a lane is nearly everything, position a gentle nudge, and no score normalization is ever needed.

Practice Exercises

Starter and solution projects live in the companion repository under `exercises/ch15`. Each starter compiles with failing tests, and the tests are the definition of done. All three need State(15), where the corpus is pre-applied, and every query vector comes from `seed/query-embeddings.json`: no exercise calls a model, and the third one is explicit about which of its numbers belong to your machine's graph rather than to the book.

Exercise 15.1 — Basic: One space, three rulers

The starter's three top-five methods secretly all measure with CosineDistance, so their orderings agree for the wrong reason and the numbers cannot pass the algebra the test

checks. Make each method use its own distance function, then watch the test hold the corpus to the normalization contract: identical order three times, l2 squared equal to twice the cosine distance, inner product equal to cosine minus one.

Hint: the variants listing is the shape; only the operator changes. The algebra works because the corpus is L2-normalized, and the test's tolerance absorbs the six-decimal rounding the corpus generator applied.

Exercise 15.2 — Intermediate: Five results, five different jobs

A plain top five for "pump maintenance" is one pump station five times with different work-order numbers — the catalogue repeats itself, and a result list that repeats itself helps nobody. Build the dedup-aware top-k: over-fetch candidates, walk them in distance order, keep the first hit of each title shape, stop at k. The failing test demands five distinct shapes, the nearest representative of each, and distances still ascending.

Hint: the candidate multiplier is a bet on how repetitive the data is, and one shape here repeats about two hundred times — when a fetch comes back all-duplicates, fetch deeper and try again rather than guessing a bigger constant.

Exercise 15.3 — Challenge: Grade your own graph

The starter's recall report grades the HNSW index against itself and never runs the iterative scan, so every mark comes back perfect — exactly the report that ships bad search. Make the truth exact and the grades honest: for each committed query vector, the exact top ten with index scans disabled, then the plain ANN answer and the strict-order iterative answer graded by overlap. The test asserts only the structural guarantees (the exact set is full, plain ANN may come back short, iterative never does) and prints the grades, because the grades belong to the graph your reset built, not to this book.

Hint: SET LOCAL inside a transaction scopes both enable_indexscan and hnsw.iterative_scan to one measurement without leaking into the pooled connection; the fixture already wraps the first for you. Run the suite, reset, run it again, and watch which columns move.

What's Next

Part III is done: LuminaWork's PostgreSQL now speaks documents, full text, arrays, ranges, geography and vectors, and every one of them from EF Core in statements you have read on these pages. Part IV turns the feature-complete application into an operable product, and it starts where operators start: Chapter 16 instruments everything the last fourteen chapters built, with interceptors on every save, both logging modes side by side, and diagnostics you can leave on in production — because before you can harden, audit or scale a system, you have to be able to see it.

End of the Sample

This sample is two chapters from opposite ends of the book, not two in a row. Chapter 1 got a container answering queries; Chapter 15, fourteen chapters later, fuses full-text ranking with pgvector similarity in one query. In between sit mapping, migrations, query plans, JSONB, full-text search, and PostGIS — the material that makes that payoff possible, and none of it is here.

What the complete book covers

- **Intermediate Core Concepts** — configuration, type mapping, relationships, migrations.
- **Querying and Performance** — LINQ translation, EXPLAIN, loading, keyset pagination, concurrency.
- **PostgreSQL Superpowers** — JSONB, full-text search, ranges, PostGIS, and the pgvector chapter previewed here.
- **Production Architecture and Operations** — auditing, tenant isolation, testing, deployment, event-driven outbox.

Every SQL listing in those chapters ran against a live PostgreSQL 18.6 before it went to print; every C# listing is an excerpt of code that compiles, diffed against the companion repository.

> **Key Takeaway**
>
> *The complete book runs 21 chapters, 4 parts, 5 appendices, 526 listings, and 63 exercises — all on Leanpub in EPUB and PDF, DRM-free.*

Get the complete book

https://leanpub.com/theentityframeworkcorehandbook

Companion code: https://github.com/RachidD68/the-entity-framework-core-and-postgresql-handbook