Beyond "Chunk and Pray"

Building Trustworthy RAG with Geometric Knowledge Graphs

Agus Sudjianto

Wing Yan Lau

Knowlytix.ai

Beyond "Chunk and Pray"

Published by Knowlytix LLC
Publisher imprint: KnowlytiX
First eBook edition, 2026
eBook ISBN: 979-8-9971189-0-7

The KnowlytiX AI Engineering Series

A collection of books dedicated to helping practitioners build reliable, interpretable and production-ready AI systems. Each volume can be read independently while, taken together, the series provides a comprehensive framework for modern AI engineering—from memory architectures and retrieval to validation, governance and enterprise deployment.

Contents

Preface

The dominant way to build a retrieval system is what this book calls *chunk and pray*: split the documents into windows, embed them, return the top few by cosine similarity and trust the language model to use them well. It demonstrates well in a controlled setting and fails in production in ways the demonstration cannot show. Top-k is not relevance, so the chunk that holds the answer loses a tie and a confident wrong answer is shipped. The retrieved text is prose, so nothing downstream can identify which sentence is evidence and which is filler. A number read back through a model drifts — $35 becomes 35.0, then "about thirty-five dollars." And when a second model is asked to grade the first, the result is a model judging a model: no provenance, no replay and no answer for the auditor who asks how the conclusion was reached. The distance between that demonstration and a retrieval system suitable for review by a regulator — one whose every answer is grounded, cited, exact and replayable — is the subject of this book. Chapter 1 makes this failure concrete and states the thesis in full.

The argument is one sentence: retrieval is a *grounding* discipline, not a similarity search. An answer should be produced *through* a verified knowledge graph, with a pointer back to the source span, not lifted from a bag of chunks the generator was trusted to read correctly. Underneath the system runs a deterministic geometric substrate, the Geometric Memory System (GMS), that replaces each chunk-and-pray layer with one that produces evidence. Top-k-and-hope becomes **triple-mediated retrieval** (Chapter 8): the question is turned into query triples and answered against the graph. Flat chunks become a structured store with **Exact Numerical Memory**, so a figure is recalled byte-exact rather than parsed out of text. The second-model judge becomes a non-LLM **verifier** (Chapter 12) that scores a claim as a geodesic distance — the same number every time, which an audit can reproduce. And when the graph cannot ground an answer, the system **abstains** rather than guesses, and a coverage monitor makes the blind spots measurable (Chapter 13). The dense vector index is not the foundation here; it is a distrusted, opt-in fallback, off by default and flagged when used (Chapter 16).

The book builds each piece on one running example: a question-answering system over a company's *annual financial report* — segment revenues, an income statement, a balance sheet, an organizational hierarchy and the prose of the management discussion and risk factors. It answers "what was total revenue?" byte-exact with a citation, resolves "which region runs the division that contains Cloud Platform?" by walking the graph (Chapter 10), and *abstains* on "what is management's outlook?" because the outlook lives in prose that no triple grounds. Every example runs on a real, local open-weight model — Qwen2.5-3B-Instruct — with no API key and nothing hosted.

Who this is for

The intended reader can already call a model and has built, or is about to build, a RAG system; the problem now is making one *trustworthy*. The reader is the engineer building retrieval that a

compliance team, an auditor or a regulator will examine — in finance, or any domain where "it usually retrieves the right thing" is not an acceptable answer. Familiarity with Python, embeddings and the shape of a transformer is assumed. No prior knowledge of geometry is required: the substrate is introduced through a small set of primitives, and the geometry itself is deferred to the GMS monograph (Appendix C makes the book self-contained).

Book, notebooks and library

Three artifacts stay in lockstep. The `knowlytix.knowledge.rag` and `knowlytix.knowledge.geode` library holds every type and function the book discusses, and it is the source of truth — if a listing ever disagrees with the library, the library is correct. A notebook per chapter runs every example against the trained store and closes with a self-check assertion. The book is the narrative, and each code listing in it is copied verbatim from a notebook. Most chapters end with an anti-pattern call-out — the specific ways practitioners go wrong on that chapter's material — and an exercise whose worked solution is in the notebook. A "GMS in practice" box marks the point in a chapter where the chunk-and-pray default is replaced by its deterministic counterpart.

Relationship to *Beyond Prompt and Pray*

This is the retrieval companion to *Beyond Prompt and Pray*, which builds governed agents. That book uses a retriever as one tool inside an agent; this book is the full treatment of that retriever. The two share the GMS substrate, so the cross-references are frequent and the bridge is explicit: Appendix A implements the agent book's policy-retrieval tool on the system this book builds. Either may be read first; they meet in the middle.

How the chapters depend on each other

- **The Problem (Chapter 1)** makes the failure concrete and states the thesis.
- **The Substrate (Chapters 2–3)** introduces the store's primitives and provenance — the span that points a fact back to its source.
- **Ingestion (Chapters 4–5)** builds the store from a document and self-corrects the extracted graph.
- **Retrieval (Chapters 8–10)** covers query-triple extraction, binding to the graph's vocabulary (Chapter 9) and answering — including multi-hop — through the GMS.
- **Generation and Trust (Chapters 11–13)** synthesizes a grounded answer, verifies its own claims and abstains when it cannot.
- **Calibration and Evaluation (Chapters 14–15)** fit the thresholds from data and measure trust, not accuracy alone.
- **Production (Chapters 16–17)** cover pluggable models, the distrusted dense fallback and an external store.
- **Capstone (Chapter 18)** assembles everything over the annual report end to end.

A comparison with other retrieval frameworks is given in Appendix B, and the artifacts required to reproduce every result are documented in Appendix D.

What this book does not cover

It does not pretrain a language model (the companion *llm-tutorial* does that), build agent runtimes (that is *Beyond Prompt and Pray*) or derive the GMS geometry (the GMS monograph). It starts with a capable local model in hand and builds the retrieval system around it. The argument concerns the system around the model, not the model itself.

Part I

The Problem

Chapter 1

Why "chunk and pray" fails

The default RAG recipe comprises three steps: chop a document into chunks, embed each chunk into a vector and at query time return the top-k chunks nearest the question. The chunks are handed to a language model with the instruction "answer from this context," in the hope that the answer is present. This chapter shows — on a single numeric question over the Northwind Industries FY2025 annual report — why hope is not a control. We then preview the alternative: retrieval mediated through a verified knowledge graph, where the answer is an exact-recall figure carrying a `file:line:char` provenance span. The geometric memory substrate that makes this possible is introduced in Chapter 2, and the provenance machinery in Chapter 3.

Four failure modes recur, and all four appear in the single example below:

1. **Hallucinated retrieval** — top-k ranks the wrong chunk first, so the model answers from text that does not contain the answer.
2. **No provenance** — a chunk is a line range over many table rows; there is no pointer to the one cell that holds the figure.
3. **Silently wrong numbers** — the figure is *parsed* out of retrieved text, with nothing to catch a wrong parse.
4. **Unverifiable** — the only check on the answer is another LLM grading the first, a model judging a model (see Chapter 12).

The thesis of the book is the inverse of each: retrieval is *triple-mediated* (answers come through a verified graph), the dense vector index is *distrusted* (off by default, flagged when used) and the system *abstains rather than guesses*.

1.1 A tiny vector RAG

Listing 1.1 is the entire "chunk and pray" recipe: split the document on blank lines, embed each chunk with a deterministic bag-of-words vector, rank by cosine to the question and take top-k. We use a bag-of-words encoder rather than a hosted embedder deliberately — the failure is *structural*, not a quirk of any one model, and it reproduces with no GPU and no API key.

Listing 1.1: A minimal vector RAG over the annual report.

```
import math, re
from collections import Counter
```

```
def chunk(md: str) -> list[tuple[int, str]]:
    """Blank-line paragraph chunks, paired with their 1-based start line."""
    chunks, buf, start = [], [], None
    for i, ln in enumerate(md.split("\n"), start=1):
        if ln.strip() == "":
            if buf:
                chunks.append((start, "\n".join(buf)))
            buf, start = [], None
        else:
            start = i if start is None else start
            buf.append(ln)
    if buf:
        chunks.append((start, "\n".join(buf)))
    return chunks

def embed(text: str) -> Counter:
    """Deterministic bag-of-words vector (the stand-in dense encoder)."""
    return Counter(re.findall(r"[a-z]+", text.lower()))

def cosine(a: Counter, b: Counter) -> float:
    shared = set(a) & set(b)
    dot = sum(a[t] * b[t] for t in shared)
    na = math.sqrt(sum(v * v for v in a.values()))
    nb = math.sqrt(sum(v * v for v in b.values()))
    return dot / (na * nb) if na and nb else 0.0

class VanillaRAG:
    def __init__(self, md: str):
        self.chunks = chunk(md)
        self.index = [embed(c) for _, c in self.chunks]

    def retrieve(self, q: str, top_k: int = 1):
        qv = embed(q)
        scored = [(cosine(qv, cv), self.chunks[i])
                  for i, cv in enumerate(self.index)]
        scored.sort(key=lambda x: x[0], reverse=True)
        return scored[:top_k]

rag = VanillaRAG(report)
print(f"{len(rag.chunks)} chunks indexed")
```

1.2 The failure: top-k is not relevance

We ask *"What was total revenue?"* The authoritative answer, 355.0, is the Revenue row of the income statement (line 36 of the report). The word "total" in the question, however, matches the **Balance Sheet** block — "**Total** Assets," "**Total** Liabilities" — at least as strongly as it matches the income-statement Revenue row. A bag-of-words ranker has no notion that "total revenue" is one concept; it counts overlapping tokens. The top hit is therefore the *wrong table*, containing 540.0/210.0/330.0 and not the answer.

Listing 1.2: Top-k returns the wrong table, with no cell-level provenance.

```
question = "What was total revenue?"
hits = rag.retrieve(question, top_k=3)
for rank, (score, (line_no, text)) in enumerate(hits, start=1):
    snippet = text.replace("\n", " ")[:80]
    print(f"#{rank}  score={score:.3f}  line={line_no}  {snippet!r}")

top_score, (top_line, top_text) = hits[0]
has_answer = bool(re.search(r"\b355(?:\.0)?\b", top_text))
numbers = re.findall(r"\d+\.\d+", top_text)
print()
print(f"top chunk contains the answer (355): {has_answer}")
print(f"numbers in the top chunk (ambiguous): {numbers}")
print("cell-level provenance:                None  "
      "(chunk spans many rows; no file:line:char for one cell)")
```

The top chunk is the Balance Sheet; `has_answer` is `False` and the visible numbers are `['540.0', '210.0', '330.0']`. The vanilla RAG would now hand that wrong table to an LLM and ask it to "answer from context." The model must pick a number from 540.0/210.0/330.0 — a silently wrong figure — or, when handed the right table, parse one cell out of eight with no provenance for the cell it selected. Three of the four failure modes are already present; the fourth follows the moment an LLM grader is added to the loop (Chapter 12).

1.3 The triple-mediated alternative

Listing 1.3 runs the same question through `RagPipeline`. The figure does not come from parsing prose: it comes from Exact Numerical Memory (`lookup_enm`), and the retrieved triple carries a `file:line:char` span back to the exact table cell. The construction of these triples from a document is the subject of Chapter 4, and the binding of a question to a triple that of Chapter 9. This listing loads the trained store and the local Qwen synthesizer; it is marked for CI execution, not for interactive authoring (the GPU is shared).

GMS in practice — exact recall with provenance vs. parsing a chunk

Listing 1.3: The same question through `RagPipeline` (CI-only).

```
# === CI-ONLY: loads the trained store + Qwen. Do not run during authoring. ===
import torch
from knowlytix.knowledge.store import GMSExpertStore
from knowlytix.knowledge.config import DocGMSConfig
from knowlytix.knowledge.llm_backend import LocalTransformersBackend
from knowlytix.knowledge.rag import RagConfig, RagPipeline
from knowlytix.knowledge.geode import QWEN_3B

STORE_PATH = str(DATA / "gms_annual_report_store")
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

store = GMSExpertStore(DocGMSConfig(store_path=STORE_PATH), device=device)
assert store.load(), f"no trained store at {STORE_PATH}"

# ENM gives the authoritative number directly, byte-exact, no parsing.
```

```
enm_total = store.lookup_enm("income_statement", "Revenue/FY2025")
print(f"lookup_enm -> {enm_total}")   # 355.0

qwen = LocalTransformersBackend(QWEN_3B, device=str(device))
pipe = RagPipeline.from_store(store, RagConfig(llm=qwen))
ans = pipe.query("What was total revenue?")
print("answer  :", ans.answer)
print("decision:", ans.decision, "| route:", ans.route,
      "| verified:", ans.verified)
for f in ans.sources:
    print(f"  fact: ({f.head}, {f.relation}, {f.tail})  "
          f"src={f.source}  @ {f.location}")
```

The triple route never parses a number out of prose. The answer is grounded in a triple and verifiable through its span; when nothing binds (the prose sections), the pipeline abstains instead of guessing (the abstention-and-coverage treatment of Chapter 13).

1.4 Limits

Triple-mediation helps only where triples exist. The four prose sections of this report — the title block, MD&A, Risk Factors and Outlook — carry zero triples; the store's coverage ratio is 0.56. A question answerable only from prose will *abstain*, not answer, unless the distrusted dense fallback is enabled (Chapter 16). Chunk-and-pray would have *guessed* a number from the wrong table; triple-mediation declines, audibly, when it cannot ground the answer. It yields verifiability and exact numerics at the cost of refusing prose-only questions, and it does nothing for facts that were never extracted in the first place. The extraction of those facts is treated in Chapter 4, and the coverage monitor that surfaces such gaps in Chapter 13.

Anti-patterns

- **Trusting top-k as relevance.** Cosine rank measures lexical (or embedding) proximity, not whether a chunk answers the question. "Total revenue" retrieved "Total Assets" first.
- **Treating retrieved prose as evidence.** A fluent paragraph that mentions "revenue" is not an authoritative figure. In this corpus the prose explicitly defers: "the authoritative figures are those reported in the tables above."
- **Parsing numbers from a chunk.** A retrieved table block holds many cells; a regex or an LLM picking one has no provenance for its choice and no guard against picking the wrong one.

Exercise — break the vector RAG on a second number

Ask the vanilla RAG a second numeric question — net income (the labeled answer is 70.0). Show that retrieving the right table is still not the same as answering: the income-statement chunk holds eight numbers across four rows and two years, so no single provenance-backed figure can be returned. The worked solution is in the chapter notebook; its core is Listing 1.4.

Listing 1.4: Second numeric question, same structural failure.

```
# Exercise solution: a second numeric question, same structural failure.
q2 = "What was net income?"
hit2 = rag.retrieve(q2, top_k=1)[0]
score2, (line2, text2) = hit2
print(f"top hit: line={line2}  score={score2:.3f}")
print(text2.replace(chr(10), " ")[:90])

# The vanilla RAG offers no ENM and no cell-level span. Even if the income-
# statement table is retrieved, the number must be *parsed* from text, and
# the chunk spans many rows -- there is no provenance for the 70.0 cell.
parsed_numbers = re.findall(r"\d+\.\d+", text2)
print("numbers visible in the chunk (ambiguous, unparsed):", parsed_numbers)
assert len(parsed_numbers) != 1, (
    "Either prose (0 numbers) or a multi-row table (>1 number): in neither\n"
    "case can the vanilla RAG return ONE provenance-backed figure."
)
print("=> no single, provenance-backed number is recoverable. Failure confirmed."
    )
```

1.5 What the notebook asserts

The chapter's claim is mechanical, so the notebook establishes it mechanically: the vanilla top-1 chunk for the numeric question is not a clean single-figure table cell, so the vanilla RAG cannot return one provenance-backed figure. Listing 1.5 is the final self-check (CPU-only; no store, no Qwen).

Listing 1.5: Self-check: chunk-and-pray returns no single provenance-backed figure.

```
# Self-check (CPU-only): the vanilla top-1 for the numeric question is NOT a
# clean, single-figure table cell with provenance -> chunk-and-pray fails.
top1_score, (top1_line, top1_text) = rag.retrieve("What was total revenue?", top_k=1)
    [0]
clean_table_cell = ("|" in top1_text
                    and len(re.findall(r"\d+\.\d+", top1_text)) == 1)
assert not clean_table_cell, (
    "Vanilla RAG unexpectedly isolated a single-figure cell; "
    "the chunk-and-pray failure did not reproduce."
)
print("OK: vanilla vector RAG returns no single provenance-backed figure.")
print("    The triple-mediated route (Listing 2) returns 355.0 with a span.")
```

Cross-references. The dense index demoted here is the lowest-trust memory tier, quarantined behind an opt-in flag (Chapter 16). The triple-mediated machinery previewed in Listing 1.3 is developed across the Retrieval part (Chapters 8, 9 and 10) and the Generation-and-Trust part (Chapters 11, 12 and 13). The calibration of the abstention threshold is treated in Chapter 14, and the end-to-end evaluation in Chapter 15.

Part II

The Substrate

Chapter 2

Geometric memory in one chapter

Chapter 1 showed what goes wrong when retrieval is similarity search over chunks. This chapter introduces the alternative the rest of the book builds on: the *GMS store*, a **trained triple register**. It is not a vector database under a different name. It is a small geometric model that has learned a knowledge graph, exposes a set of typed operations over it and keeps every authoritative number in a separate exact-recall memory with cryptographic integrity.

We stay at agent-author altitude. We describe what each primitive does, what its output means and where the geometry contributes — not how rotors, spherical caps or parallel transport are implemented. Those internals are presented in Appendix C and, in full, in the GMS monograph. This is the fourth memory tier of the agent book (*Beyond Prompt and Pray*, Ch. 9) made concrete for retrieval.

The store used throughout is the Northwind Industries FY2025 store built by `scripts/build_store.py` (Foundation F2). Every value cited here is byte-for-byte the value in `data/corpus_facts.md`; the notebook asserts it.

2.1 The store is six primitives over a learned graph

A `GMSExpertStore` exposes six operations. Two are exact lookups over the asserted graph (`lookup_enm`, `query_triples`); the other four are geometric reads of the trained manifold (`score_triple`, `link_predict`, `check_holonomy`, `tension_energy`). We reopen a store with the same geometry and loss configuration it was trained under, then call `load()`.

```
import torch
from knowlytix.core.config import GeometryConfig
from knowlytix.knowledge.config import DocGMSConfig
from knowlytix.knowledge.store import GMSExpertStore

REPO_ROOT = os.path.join(os.path.dirname(os.getcwd()), "code") if os.path.basename(os.
    getcwd()) == "notebooks" else os.getcwd()
STORE = os.path.join(REPO_ROOT, "data", "gms_annual_report_store")

cfg = DocGMSConfig(
    store_path=STORE,
    ingest_mode="regex",
    loss_mode="cap",
    geometry=GeometryConfig(d_v=64, d_u=64, m=32, d=32),
)
store = GMSExpertStore(cfg, device=torch.device("cpu"))
```

```
assert store.load(), f"no store at {STORE}  run scripts/build_store.py first"
print("relations:", sorted(store.adapter.relation_to_idx))
```

The configuration must match the build (cap loss, 64/64/32/32 geometry) so the saved weights load into a model of the right shape. The relations printed are the graph's vocabulary: `has_revenue`, `has_headcount`, `has_division`, `has_region`, `has_head`, `has_fy2025` and so on.

2.2 `lookup_enm`: exact recall, not re-parsing

Authoritative numbers do not live in the geometry. They live in **Exact Numerical Memory** (ENM), keyed by `(category, id)` with a SHA-256 integrity check, and `lookup_enm` returns them byte-exact. This is the practice the book recommends: **a figure that matters is looked up, never parsed out of retrieved prose.** The listing reads FY2025 revenue from ENM, then shows a naive parse of the same number silently returning the wrong figure when a sentence is reordered.

```
import re

# Authoritative path: byte-exact read from Exact Numerical Memory.
revenue_fy2025 = store.lookup_enm("income_statement", "Revenue/FY2025")
total_segment_rev = store.lookup_enm("segment_performance", "Total/All/Revenue")
print("ENM income_statement/Revenue/FY2025 =", revenue_fy2025)
print("ENM segment_performance/Total/All/Revenue =", total_segment_rev)

# Chunk-and-pray path: parse a figure out of a retrieved sentence.
# Two sentences mention '355'  and one mentions the WRONG prior-year basis.
retrieved_prose = (
    "Total revenue grew to $355M in FY2025, up from $320M, while the "
    "Cloud Platform segment alone contributed $120M."
)
first_number = float(re.search(r"\$(\d+)M", retrieved_prose).group(1))
print("naive parse (first $NM match) =", first_number)

# The parse is right here only by luck of word order. Reorder the clause and
# the same regex now returns the SEGMENT figure, not total revenue:
reordered = (
    "The Cloud Platform segment contributed $120M as total revenue "
    "grew to $355M in FY2025."
)
wrong = float(re.search(r"\$(\d+)M", reordered).group(1))
print("naive parse after reorder =", wrong, "(should be 355, got", wrong, ")")
assert wrong != revenue_fy2025, "the prose parse silently returned the wrong number"
print("ENM is order-invariant and exact; the parse is neither.")
```

Both ENM reads return `355.0`. The reordered parse returns `120.0` — the Cloud Platform segment figure — with no error and no warning. The regex was never wrong; the assumption that "the first dollar amount in the chunk is the answer" was incorrect. ENM makes no such assumption: the value is bound to a key, not to a position in text.

GMS in practice — ENM lookup vs. re-parsing

The chunk-and-pray default answers "what was total revenue?" by retrieving a passage and extracting digits from it. The geometric counterpart looks the value up under

`("income_statement", "Revenue/FY2025")` and returns `355.0` byte-exact, every time, regardless of how the surrounding prose is phrased. The number is data, not a substring.

2.3 score_triple: contradiction is distance

`score_triple(head, rel, tail)` returns a geodesic distance on the trained manifold: **lower means more plausible**. A fact the store learned sits close to the relation-conditioned cap center; a fabricated tail sits measurably farther. Numeric tails are canonicalized strings (`"340.0"`), matching the triples in `corpus_facts.md`.

```
true_d = store.score_triple("cloud platform", "has_headcount", "340.0")
false_d = store.score_triple("cloud platform", "has_headcount", "520.0")  # that's
    retail's
print(f"d(cloud platform, has_headcount, 340.0) = {true_d:.4f}  [asserted]")
print(f"d(cloud platform, has_headcount, 520.0) = {false_d:.4f}  [false]")
print(f"geodesic gap (false - true) = {false_d - true_d:.4f}")

# The asserted fact scores strictly closer than the swapped tail.
assert true_d < false_d, "true fact must score closer than the false one"
rho = store.cap_radius("has_headcount")
print("cap radius rho for has_headcount =", rho)
```

The asserted headcount (340) scores strictly closer than Retail's 520 swapped in as a decoy. The gap reflects the geometry's confidence that the second triple does not belong. The cap radius ρ for `has_headcount` is the learned admissible band around the center — the calibration of that band into an accept/abstain decision is the subject of Chapter 14, not of this chapter.

2.4 query_triples and link_predict: asserted vs. predicted

`query_triples` is exact pattern-matching over the document graph. Leaving a slot `None` wildcards it. This is the path the retriever (Chapter 10) prefers — asserted edges with provenance, never a guess.

```
cloud_edges = store.query_triples(head="cloud platform")
for h, r, t in cloud_edges:
    print(f"{h:>16} {r:>14} {t}")

# All four segments that report a revenue edge:
rev_edges = store.query_triples(relation="has_revenue")
segments = {h: t for (h, r, t) in rev_edges if h != "total"}
print("\nper-segment revenue:", segments)
assert ("cloud platform", "has_division", "technology") in cloud_edges
```

Cloud Platform's edges include `has_revenue 120.0`, `has_headcount 340.0` and `has_division technology`. The per-segment revenue map is `{cloud platform: 120.0, devices: 80.0, logistics: 95.0, retail: 60.0}` — and $120 + 80 + 95 + 60 = 355$, the Total row. That sum relationship is what GEODE's anchor checker enforces in Chapter 5.

When no asserted edge exists but the store's ranked guess for `(head, relation, ?)` is wanted, `link_predict` scores every type-valid tail and returns the closest. It is type-constrained: only entities ever seen as a tail of this relation are scored, so numeric and categorical tails never mix.

```
ranked = store.link_predict("cloud platform", "has_division", top_k=3)
for tail, dist in ranked:
    print(f"{tail:>14}  d={dist:.4f}")

# Cloud Platform's division is Technology  it should rank first.
assert ranked, "link_predict returned no candidates"
top_tail, _ = ranked[0]
print("top-ranked division:", top_tail)
```

Technology ranks first. This is a *prediction*, not an assertion: Chapter 10 explains why the retriever always prefers the asserted `query_triples` edge over a `link_predict` guess whenever one exists. Link prediction serves to fill gaps, with that caveat surfaced in the audit trail — never to override a fact the document actually states.

2.5 `check_holonomy` and `tension_energy`: composition and contradiction

The last two primitives read the trained *geometry* rather than a stored table. `check_holonomy(path, direct)` measures the holonomy defect — how far composing a relation path drifts from a direct edge. **Zero is consistent.** It is the backbone of multi-hop retrieval (Chapter 10) and of GEODE's composition critic (Chapter 5).

```
defect = store.check_holonomy(["has_division", "has_region"], "has_region")
print("holonomy defect for has_division  has_region vs has_region =", defect)
tau = store.config.verify.tau_path
print("path-consistency threshold tau_path =", tau)
# A defect at or below tau_path is treated as a consistent composition.
assert defect is not None, "both relations must exist for a holonomy read"
```

`tension_energy(a, b)` reads the relationship between two entities on a 0..2 scale: 0 **= agree,** $\sqrt{2} \approx 1.41$ **= unrelated,** 2 **= contradict**. Functional facts — one division head, one fiscal-year-end — make contradiction a geometric signal rather than a string comparison.

```
te_heads = store.tension_energy("dana cole", "sam reyes")
te_self = store.tension_energy("dana cole", "dana cole")
print(f"tension(dana cole, sam reyes) = {te_heads:.4f}  [two distinct heads]")
print(f"tension(dana cole, dana cole) = {te_self:.4f}  [identical -> agree]")
assert te_self <= te_heads, "an entity must not contradict itself"
print("contradiction is a distance, not a string mismatch.")
```

`dana cole` (Technology) and `sam reyes` (Operations) are distinct heads of distinct divisions; the energy reads them as not agreeing. An entity compared with itself reads as agreement. This is the mechanism that flags a "two CEOs" contradiction in Chapter 5 and a hallucinated claim in Chapter 12 — the store measures disagreement as distance instead of relying on an LLM to notice it.

2.6 Limitations

The store does only what it was trained on. `score_triple` and `tension_energy` return *relative* geometry, not calibrated probabilities: the gap between a true and false triple is meaningful, but

turning that gap into an accept/abstain decision requires the calibration of Chapter 14 — the raw distance must not be read as a confidence. The primitives operate only over entities and relations the ingest actually populated: a query about a prose-only section (MD&A, Risk Factors, Outlook) finds no triples at all, which is the coverage blind spot quantified in Chapter 13. `lookup_enm` is exact only for values that reached ENM during ingest; a number buried in prose was never registered and cannot be looked up. The geometry detects contradictions that have geometric redundancy to check against (a sum, a duplicate, a composable path); a lone value with no such redundancy is the limit named in Chapter 5.

Anti-patterns

- **Treating the store as a black-box vector database.** It is a typed register with six distinct operations, each with different trust semantics. Calling only a "similarity" method and ignoring `lookup_enm` / `query_triples` discards the exact, asserted paths — the trustworthy ones — and retains only the approximate one.
- **Trusting a parsed number.** Any figure that matters belongs in ENM and is read with `lookup_enm`. Parsing it out of a retrieved chunk reintroduces the silent-wrong-number failure of Chapter 1; "the first $NM in the passage" is not a contract.
- **Reading a `link_predict` guess as a fact.** Link prediction ranks plausible tails when no edge is asserted. If `query_triples` returns an edge, that edge takes precedence — a prediction must never overwrite an assertion.

Exercise — Find a contradiction with `tension_energy`

The annual report names one head per division. Suppose an ingest error produced a *second* head for Technology — the "two CEOs" problem. Using only `tension_energy`, show that two entities the store treats as the same functional role read closer to agreement than two it treats as distinct roles, and explain why a string comparison would miss the case where the two names are spelled differently but denote the same person. (Worked solution: see the `tension_energy` cell and the self-check in `notebooks/02_geometric_memory.ipynb`; the assertion `te_self <= te_heads` is the kernel of the answer. The exercise extends by scoring a fabricated third head against `dana cole` and observing the energy.)

Self-check. The notebook's final cell proves this chapter's claim: ENM returns `355.0` for both FY2025 revenue and total segment revenue and `340.0` for Cloud Platform headcount, byte-exact; and the asserted headcount triple scores strictly closer on the manifold than the fabricated one.

Where this goes next. Appendix C recapitulates these six primitives in self-contained form and explains the calibration method; the GMS monograph covers the geometry that produces the distances. Provenance — how each asserted triple points back to its source span — is the subject of Chapter 3, and the construction of the graph these primitives read from is developed in Chapter 4.

Chapter 3

Provenance: span ↔ triple

A retrieved fact is only as trustworthy as the source to which it can point. In a chunk-and-pray pipeline the chain of custody is lost at ingestion: text is split and embedded and the offsets are discarded, so an answer can cite "a chunk" but never the exact cell that carried the number. The geometric memory introduced in Chapter 2 keeps the reverse map. Every triple in the store can be re-aligned to the precise span from which it came — `file:line:char`, together with the raw substring. When a triple has *no* structural anchor, the ledger reports that absence explicitly.

This chapter builds that span↔triple map for the Northwind Industries FY2025 annual report with `ProvenanceLedger`. We resolve a segment-revenue triple to its table cell, observe a prose claim resolve as `unaligned`, use the coarse `in_section` anchor as a provenance-span fallback for prose and verify that a resolved span actually *supports* its triple with `is_consistent`. The span produced here is the citation later carried onto a retrieved answer in Chapter 10.

Provenance alignment is purely text→span work over the markdown source; it does not load the trained store or the query-time LLM. We import the symbols exported by `knowlytix.knowledge.geode.provenanc`

```
from knowlytix.knowledge.geode.provenance import (
    ProvenanceLedger,
    Provenance,
    is_consistent,
    canon,
)

REPORT = "data/annual_report.md"
ledger = ProvenanceLedger(REPORT)
print("ledger built over", REPORT)
```

3.1 Resolving a triple to its table cell

The triple `(cloud platform, has_revenue, 120.0)` was extracted from a table cell. `resolve` re-aligns it post-hoc and returns a `Provenance`: the 1-based line, the absolute character span, the raw cell text, and the alignment `method`.

```
p = ledger.resolve("cloud platform", "has_revenue", "120.0")
print("location:", p.location())
print("method:  ", p.method)
print("raw:     ", repr(p.raw))
print("line:    ", repr(ledger.line(p.line_no)))
```

On the shipped corpus (line 15 is the Cloud Platform row) this prints:

```
location: data/annual_report.md:15:553-558
method:   table_cell
raw:      '120.0'
line:     '| Cloud Platform | Technology | 120.0 | 340 |'
```

The `method` is `table_cell`: the figure is anchored to a precise char span rather than parsed loosely from prose. The `raw` substring is the cell text itself — byte-exact `120.0`, the same value the store's Exact Numerical Memory returns (Chapter 2). Provenance and ENM agree because they read the same cell; nothing is re-typed from a model's memory.

3.2 A prose triple is `unaligned`

The MD&A section states that the Cloud Platform segment "continued to lead the company's topline." That sentence carries no table cell, schema bullet or section header for a `(cloud platform, leads, topline)` triple. Post-hoc structural alignment cannot find an anchor, so it resolves as `unaligned`.

```
prose = ledger.resolve("cloud platform", "leads", "topline")
print("location:", prose.location())
print("method:  ", prose.method)
print("raw:     ", repr(prose.raw))
print("aligned? ", prose.method != "unaligned")
```

```
location: data/annual_report.md:-1:-1--1
method:   unaligned
raw:      ''
aligned?  False
```

The `line_no` and char offsets are `-1`: the sentinel for *no structural anchor*. This signals that prose triples require spans captured at *extraction* time. `ProvenanceLedger` re-aligns regex-extracted *structured* triples — tables, schema bullets and section headers — only; it does not invent a span for an LLM-extracted prose claim. The coverage of these blind spots is measured directly in Chapter 13.

3.3 The provenance-span fallback: coarse triples cover prose

A prose section *can* still receive a span — not for a fine-grained fact, but at section granularity via an `in_section` triple keyed on the section slug. The ledger resolves it to the section-header line, giving the answer layer a bounded region to cite even where no table fact exists.

```
# section slugs are derived from headers: '## 6. Management Discussion and
# Analysis' -> 'management_discussion_and_analysis'
sec = ledger.resolve("md&a note", "in_section",
                     "management_discussion_and_analysis")
print("location:", sec.location())
print("method:  ", sec.method)
print("raw:     ", repr(sec.raw))
```

The MD&A header is on line 60:

```
location: data/annual_report.md:60:1527-1567
method:   section_header
raw:      '## 6. Management Discussion and Analysis'
```

This is the *provenance-span fallback*: a coarse `in_section` triple anchors a prose region to its header span. It does *not* make the prose ENM-checkable — it provides only a citable location. Fine numeric facts still come from tables (Section 3.1). When this span is carried onto a retrieved answer it becomes the citation a reader follows back to the document (Chapter 10).

GMS in practice — provenance is recovered, not discarded

The chunk-and-pray default embeds text and drops the offsets, so a citation can only name a chunk. The geometric counterpart keeps the triple as the unit of memory and re-derives its exact source span on demand: a `table_cell` provenance is byte-exact down to `file:line:char-char`, and a missing anchor is reported as `unaligned` rather than guessed.

3.4 Verifying that a span supports its triple

Resolving to a span is not sufficient; the span must actually *support* the triple. `is_consistent` re-reads the raw substring and checks it against the tail — numeric-aware for table cells, slug-aware for headers and schema bullets.

```
good = ledger.resolve("total", "has_revenue", "355.0")
print("total revenue:", good.location(), "raw=", repr(good.raw),
      "consistent=", is_consistent(good))

# An unaligned prose triple can never be consistent.
print("prose triple:  consistent=", is_consistent(prose))
```

```
total revenue: data/annual_report.md:19:698-703 raw= '355.0' consistent= True
prose triple:  consistent= False
```

`is_consistent` parses `355.0` out of the raw cell and matches it to the tail within `1e-6`. The unaligned prose triple has no span to read, so it is correctly `False` — the system does not claim support that it cannot point to.

3.5 Limitations

`ProvenanceLedger` re-aligns *structured* triples by re-reading the markdown; it is not a general fact verifier. It cannot anchor an LLM-extracted prose claim — such claims require a span captured at extraction time, and absent one the ledger returns `unaligned` rather than a best guess. Its keys are `(entity, relation)` pairs, so on documents with recurring row labels (financial scenario tables that reuse "Cumulative" or "Total" across sub-tables) those keys collide and post-hoc alignment can land on the wrong cell; reliable provenance there requires table-scoped entity IDs assigned at extraction time rather than after the fact. Finally, `is_consistent` checks that a span contains the tail value — it does not check that the *extraction* chose the right triple in the first place; that is the role of the GEODE critic and anchors (Chapter 5). The extraction step that produces these triples is detailed in Chapter 4.

Anti-patterns

- **Discarding extraction provenance.** Splitting text into chunks and embedding them discards the offsets, leaving the system able to cite "a chunk" but never the exact cell. Keep the triple as the unit of memory and recover its span on demand.
- **Loosely-keyed provenance that collides.** Keying provenance on a bare row label means reused labels ("Total", "Cumulative") resolve to whichever cell the index saw last. Scope keys to the table when labels recur.
- **Treating "resolved" as "verified".** A span that exists is not a span that supports the triple. Run `is_consistent` before presenting a provenance as evidence.

Exercise — audit a batch of triples for consistency

Resolve all five segment-revenue triples with `ledger.build(...)` and confirm every aligned one is consistent. The worked solution is in `notebooks/03_provenance.ipynb`.

```
triples = [
    ("cloud platform", "has_revenue", "120.0"),
    ("devices", "has_revenue", "80.0"),
    ("logistics", "has_revenue", "95.0"),
    ("retail", "has_revenue", "60.0"),
    ("total", "has_revenue", "355.0"),
]
provs = ledger.build(triples)
for key, pr in provs.items():
    flag = "OK " if is_consistent(pr) else "BAD"
    print(flag, key, "->", pr.location(), repr(pr.raw))
```

Every row prints `OK`: each segment-revenue triple aligns to a table cell whose raw value equals the tail, and the Total row aligns as well — so the sum anchor (Total = $\sum$ segments, Chapter 5) has a provenance to cite should it ever break.

3.6 Self-check

The chapter's claim, asserted in the notebook's final cell: a resolved segment-revenue triple carries the exact table cell as its provenance, the cell supports the triple, and a prose triple with no structural anchor is reported as `unaligned` and inconsistent.

```
p = ledger.resolve("cloud platform", "has_revenue", "120.0")

# 1. the raw span IS the table value, byte-exact
assert p.raw == "120.0", p.raw
# 2. it aligned to a table cell with a real char span
assert p.method == "table_cell"
assert p.line_no == 15 and p.char_start >= 0 and p.char_end > p.char_start
# 3. the span actually supports the triple
assert is_consistent(p)
# 4. a prose triple with no structural anchor is unaligned + inconsistent
u = ledger.resolve("cloud platform", "leads", "topline")
assert u.method == "unaligned" and u.line_no == -1
```

```
assert not is_consistent(u)
print("OK: span<->triple provenance verified; prose triple correctly unaligned")
```

Part III

Ingestion

Chapter 4

From Document to Graph

A retrieval system is only as trustworthy as the graph it answers through. This chapter builds that graph: it turns the Northwind Industries annual report (`data/annual_report.md`) into a trained, queryable `GMSExpertStore`. The geometric memory substrate that this graph populates was introduced in Chapter 2, and the triple-mediated retrieval model it serves is developed in Chapter 8. The pipeline is deliberately plain — ingest the markdown deterministically, run GEODE self-correction (Chapter 5), train the geometric model, populate Exact Numerical Memory (ENM) and save — but two decisions in it determine whether the store's numbers stay exact. First, ingestion runs in *regex mode*: a deterministic parser, no LLM in the loop, byte-stable across hosts. Second, every authoritative number is parsed *once* into ENM rather than left as a string that a model re-reads at query time. The store we build here is the substrate every later chapter retrieves through.

The entire build is a single call. `build_rag_store` composes ingestion, correction, training and persistence; `store_from_triples` is its inner half (train and save from an already-corrected triple set) and is what Chapter 5 calls after it produces a cleaned graph.

4.1 Configuration

We bootstrap the branch library exactly as every notebook does (the canonical three lines from the global brief), then assemble a `DocGMSConfig`. The geometry is small ($d = 32$) because the corpus is small; `ingest_mode` is `"regex"` so that no model-extracted numbers can enter the store, and `loss_mode` is `"cap"` for the relation-conditioned spherical-cap admissibility used by the verifier in Chapter 12.

```
import os, sys
KNOWLYTIX_SRC = os.environ.get("KNOWLYTIX_SRC", "/home/asudjianto/jupyterlab/GMS-
    knowlytix")
sys.path.insert(0, KNOWLYTIX_SRC)
```

```
import torch

from knowlytix.core.config import GeometryConfig, TrainConfig
from knowlytix.knowledge.config import DocGMSConfig
from knowlytix.knowledge.geode.rag import build_rag_store
from knowlytix.knowledge.geode.loop import make_default_trainer
```

```
REPO_ROOT = os.path.join(os.path.dirname(os.getcwd()), "code") if os.path.basename(os.
    getcwd()) == "notebooks" else os.getcwd()
MD = os.path.join(REPO_ROOT, "data", "annual_report.md")
STORE = os.path.join(REPO_ROOT, "data", "gms_annual_report_store")

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
config = DocGMSConfig(
    store_path=STORE,
    ingest_mode="regex",   # deterministic: no model-extracted numbers
    loss_mode="cap",
    geometry=GeometryConfig(d_v=64, d_u=64, m=32, d=32),
    train=TrainConfig(epochs=150, batch_size=64, neg_samples=16,
                      lr=5e-3, lr_riemannian=2e-3),
)
```

4.2 Building the store

The first listing builds and saves the store, then prints the counts and the GEODE audit. The audit records what the correction loop did before any number reached the index. On the clean shipped corpus there is nothing to fix, so corrections and anchor violations are both zero; Chapter 5 corrupts a figure to make the loop visibly correct an error.

```
res = build_rag_store(
    MD, config, device=device,
    geode_trainer=make_default_trainer(device, epochs=80),
)
store = res.store

print(f"converged={res.converged}  iterations={res.iterations}")
print(f"entities={res.n_entities}  triples={res.n_triples}  enm={res.n_enm}")
print(f"relations={len(store.adapter.relation_to_idx)}")
print(f"GEODE corrections:      {len(res.corrections)}")
print(f"GEODE anchor violations:{len(res.anchor_violations)}")
```

```
converged=True  iterations=1
entities=... triples=... enm=21
relations=10
GEODE corrections:      0
GEODE anchor violations:0
```

The store carries **ten** relations — `has_amount`, `has_division`, `has_fy2024`, `has_fy2025`, `has_head`, `has_headcount`, `has_region`, `has_revenue`, `has_value`, `in_section` — and **21** ENM entries covering the segment-performance, income-statement and balance-sheet tables. The relational edges (`has_division`, `has_region`, `has_head`) are what the multi-hop chains in Chapter 10 traverse; the ENM entries are the exact figures those chains terminate in. The binding of an entity to its geometric representation, which makes these lookups exact, is developed in Chapter 9.

4.3 Reload and query

A store is a set of files under `store_path`. To demonstrate that it persisted, the second listing constructs a fresh `GMSExpertStore` from the same config and calls `load()` — no rebuild, no GEODE, no training — then reads a figure from ENM and pattern-matches a triple. `lookup_enm` returns the value byte-exact; it does not parse a number out of text at query time.

```
from knowlytix.knowledge.store import GMSExpertStore

reloaded = GMSExpertStore(config, device)
assert reloaded.load(), f'no store at {STORE}'

# Exact numeric recall (segment_performance category, byte-exact).
cloud_rev = reloaded.lookup_enm("segment_performance",
                                "Cloud Platform/Technology/Revenue")
total_rev = reloaded.lookup_enm("income_statement", "Revenue/FY2025")
print(f"Cloud Platform revenue = {cloud_rev}")
print(f"Total FY2025 revenue   = {total_rev}")

# Pattern-match the relational triple that links the segment to its division.
div = reloaded.query_triples(head="cloud platform", relation="has_division")
print("cloud platform division:", div)
```

```
Cloud Platform revenue = 120.0
Total FY2025 revenue   = 355.0
cloud platform division: [('cloud platform', 'has_division', 'technology')]
```

The reloaded store answers identically to the freshly built one. The figures match the report's Segment Performance and Income Statement tables exactly, and the `cloud platform` → `technology` edge is present for the hierarchy chains in Chapter 10.

GMS in practice — ingest once, recall exact

The chunk-and-pray default (Chapter 1) leaves numbers as text in retrieved chunks and lets the generator re-read "120.0" at answer time — one OCR slip or tokenizer quirk away from a wrong figure. The geometric counterpart parses each authoritative number *once*, at ingestion, into ENM with SHA-256 integrity, and recalls it byte-exact. The number does not pass through the model's text channel again. Provenance back to the source table cell is the subject of Chapter 3.

4.4 Limitations

This chapter builds a graph; it does not guarantee that the graph is *complete*. Regex ingestion extracts triples from tables and declared schema, not from free-form prose: the report's Management Discussion, Risk Factors and Outlook sections produce zero triples and constitute measurable coverage blind spots (`coverage_ratio` = 0.56 on this corpus). This is by design — prose carries no authoritative figure — but it means that a question whose answer lives only in prose cannot be answered through the graph; the system abstains (Chapter 13) or falls back to a distrusted dense index (Chapter 16). Regex mode also does not recover relations that a language model might infer

from narrative; `ingest_mode="hybrid"` adds that capability at the cost of determinism and is out of scope here. Finally, a build that completes without error is not necessarily correct: GEODE catches contradictions and broken sum anchors, but a lone, non-redundant value with no anchor to check it against is accepted as given. Chapter 5 states that limitation precisely.

Anti-patterns

- **Trusting the extractor blindly.** A built store is not a verified store. Inspect `res.corrections` and `res.anchor_violations` before indexing, and treat a non-empty anchor-violation list as a build failure to investigate rather than a warning to ignore.
- **Mixing model-extracted numbers into ENM.** ENM is the authoritative numeric channel. Populating it from `hybrid` or `llm_only` extraction allows a model's parse to become ground truth. Authoritative figures come from tables via deterministic ingestion; `ingest_mode="regex"` should be retained for anything that a number is read back from.
- **Re-parsing numbers at query time.** Once a figure is in ENM, it should be read with `lookup_enm`. Re-extracting "$120.0M" from a retrieved chunk reintroduces precisely the failure mode that ENM exists to remove.

Exercise — add a row, rebuild, retrieve

Add a fifth segment to the report and confirm the new fact is retrievable. Append a `Media | Technology | 40.0 | 90` row to the Segment Performance table (write to a *copy* so the shipped corpus stays untouched), rebuild into a scratch store, and look up `Media/Technology/Revenue`. The worked solution is in the chapter notebook; it asserts the new figure equals `40.0`.

```
import shutil, tempfile

with open(MD, encoding='utf-8') as f:
    text = f.read()
# Insert the new row directly above the Total row of the segment table.
new_row = "| Media | Technology | 40.0 | 90 |\n"
total_line = "| Total | All | 355.0 | 1500 |"
augmented = text.replace(total_line, new_row + total_line)
assert new_row in augmented

scratch_dir = tempfile.mkdtemp(prefix='ch4_aug_')
aug_md = os.path.join(scratch_dir, 'annual_report_aug.md')
with open(aug_md, 'w', encoding='utf-8') as f:
    f.write(augmented)

import dataclasses
aug_config = dataclasses.replace(
    config, store_path=os.path.join(scratch_dir, 'store'))
aug = build_rag_store(aug_md, aug_config, device=device,
                      geode_trainer=make_default_trainer(device, epochs=80))
media_rev = aug.store.lookup_enm('segment_performance',
```

```
                                    'Media/Technology/Revenue')
print('new segment Media revenue =', media_rev)
assert media_rev == 40.0
shutil.rmtree(scratch_dir)
```

4.5 Self-check

The notebook ends with the assertion that establishes this chapter's claim: a store built from the document saves and reloads with identical authoritative figures and carries the four reportable segments. The asserted values are exactly those in `data/corpus_facts.md`.

```
# Round-trip: the reloaded store agrees with the freshly built one
# on the authoritative figures, and carries the four segments.
for cat, key, expected in [
    ('segment_performance', 'Cloud Platform/Technology/Revenue', 120.0),
    ('segment_performance', 'Devices/Technology/Revenue', 80.0),
    ('segment_performance', 'Logistics/Operations/Revenue', 95.0),
    ('segment_performance', 'Retail/Operations/Revenue', 60.0),
    ('segment_performance', 'Total/All/Revenue', 355.0),
]:
    assert store.lookup_enm(cat, key) == expected
    assert reloaded.lookup_enm(cat, key) == expected

segments = {h for (h, r, t) in reloaded.triples if r == 'has_revenue'}
assert {'cloud platform', 'devices', 'logistics', 'retail'} <= segments
assert reloaded.stats()['enm_entries'] == store.stats()['enm_entries'] == 21
print('OK: store round-trips; 4 segments present; 21 ENM entries')
```

With the graph built and persisted, Chapter 5 shows how GEODE cleans it before indexing — corrupting a figure so that the sum anchor fires, and a relation so that the composition critic catches it. The retrieval and answering that this store supports are taken up in Chapter 10, and the provenance ledger that ties each figure back to its source cell is developed in Chapter 3. Foundation brief F2 is the build script that this chapter wraps.

Chapter 5

GEODE: self-correcting extraction

Chapter 4 built a store from the annual report by trusting the extractor: whatever triples the ingest produced went straight into the graph. This chapter does not trust it. Before a triple is indexed, GEODE runs a closed loop that lets the *geometry judge the extraction.* A *composition critic* catches relational triples that contradict the rest of the graph, and external *anchors*—a declared sum, a duplicate value—catch numeric errors to which the geometry is structurally blind. We corrupt the corpus deliberately and observe the loop flag each break *with provenance*, then state the one class of error that neither mechanism can catch. The provenance ledger that records each source span was introduced in Chapter 3; the retrieval that consumes the corrected graph is developed in Chapter 8.

Every symbol here comes from `knowlytix.knowledge.geode`. The actor LLM, when it is used at all, is Qwen 3B only—a research-integrity constraint recorded in `EXTRACTION_AGENT_DESIGN.md`. Claude is never in the loop; the geometry, not a large model, is the source of correction.

5.1 The redundancy the loop exploits

GEODE cannot correct what it cannot cross-check. The annual report provides two forms of redundancy. The segment table declares a `Total` row whose revenue equals the sum of the four segment revenues—$120 + 80 + 95 + 60 = 355$. That arithmetic is what the **sum anchor** verifies. Separately, the `segment` → `division` → `region` hierarchy is a relational chain: a segment's division, composed with that division's region, must land on the same region that every other segment in that division reaches. That composition is what the **composition critic** verifies.

```
# Canonical triples from the annual report (grounded in data/corpus_facts.md).
# In a real run these come from `ingest_markdown` / `build_rag_store`; here we
# state them explicitly so the corruption is unambiguous.
clean_triples = [
    ("cloud platform", "has_revenue", "120.0"),
    ("devices", "has_revenue", "80.0"),
    ("logistics", "has_revenue", "95.0"),
    ("retail", "has_revenue", "60.0"),
    ("total", "has_revenue", "355.0"),
    ("cloud platform", "has_division", "technology"),
    ("devices", "has_division", "technology"),
    ("logistics", "has_division", "operations"),
    ("retail", "has_division", "operations"),
    ("technology", "has_region", "north america"),
```

```
    ("operations", "has_region", "europe"),
]

# The Total row declares: Total = sum of the four segments.
parts = [120.0, 80.0, 95.0, 60.0]
assert sum(parts) == 355.0
```

5.2 Corrupt a segment figure: the sum anchor flags it

Consider changing Devices revenue from `80.0` to `88.0`. Now $120 + 88 + 95 + 60 = 363$, but the `Total` row still reads `355.0`. No single triple is *internally* wrong—each is a plausible revenue figure—so the geometry has nothing to flag. The sum anchor does flag it: it reads the exact values back through the integrity-checked ENM register and compares the declared total to the sum of its parts. `AnchorChecker.auto_sum_constraints` derives the constraint from the `Total` row just as a financial statement already declares it.

```
from knowlytix.knowledge.geode import (
    enm_from_triples, AnchorChecker, ProvenanceLedger,
)

# Inject the sum break: Devices 80.0 -> 88.0.
corrupted = [
    (h, r, "88.0") if (h, r) == ("devices", "has_revenue") else (h, r, t)
    for (h, r, t) in clean_triples
]

# Build the integrity-checked exact register, then the anchor checker.
enm = enm_from_triples(corrupted)
ledger = ProvenanceLedger.from_text(open("../code/data/annual_report.md").read(),
                                    "../code/data/annual_report.md")
checker = AnchorChecker.from_enm(enm, ledger)

# The sum constraint is derived automatically from the "total" row.
constraints = checker.auto_sum_constraints()
for c in constraints:
    print("constraint:", c.total, "= sum", [p[0] for p in c.parts])

violations = checker.check_all(corrupted)
for v in violations:
    print(v.kind, "|", v.message)
    for loc in v.locations:
        print("   provenance:", loc.location(), "raw:", repr(loc.raw))
```

```
constraint: ('total', 'has_revenue') = sum ['cloud platform', 'devices',
                                            'logistics', 'retail']
sum | declared total total.has_revenue=355 != sum(parts)=363 (off by 8); a
      part value is likely wrong (candidates: ['cloud platform', 'devices',
      'logistics', 'retail'])
```

```
    provenance: ../code/data/annual_report.md:19:698-703 raw: '355.0'
    provenance: ../code/data/annual_report.md:16:592-596 raw: '80.0'
```

The anchor *detects* the break, quantifies it (off by 8) and points back into the segment table. It does *not* auto-rewrite a number: a sum constraint establishes that the total disagrees with its parts but cannot, on its own, determine which part is wrong. This is the localization limit of a sum—it escalates to review rather than guessing. We contrast this with the relational case below, where a second source of truth makes the correct value recoverable.

GMS in practice — the anchor reads exact values, never re-parses text

A chunk-and-pray pipeline, of the kind examined in Chapter 1, would re-read the number from the retrieved cell and trust it. GEODE does not: `enm_from_triples` stores each tail losslessly under a SHA-256 integrity check, and `AnchorChecker.from_enm` pulls it back through `read_exact`, which raises on a corrupted entry. The arithmetic that catches the break therefore runs on integrity-checked exact numbers, not on a value re-parsed from prose.

5.3 Corrupt a relational triple: the critic catches and fixes it

We now break a *relational* fact: we claim `cloud platform has_division operations` when it is `technology`. This contradicts the redundant chain. The `CompositionCritic` measures the geodesic gap between the composed-path prediction ($r_1 \circ r_2$) and the asserted tail; a clean edge closes to approximately zero, a contradicting one does not. `GeodeLoop` localizes the flag to its source span and repairs it with the geometry's prediction. This listing trains a small GMS, so it is GPU/CI-only—marked for the lead to execute.

```
from knowlytix.knowledge.geode import GeodeLoop, make_default_trainer

# Break a relational triple: cloud platform's division technology -> operations.
relational_break = [
    (h, r, "operations") if (h, r) == ("cloud platform", "has_division") else (h, r, t)
    for (h, r, t) in clean_triples
]

# Write the corrupted triples to a tiny markdown table so the loop's regex
# ingest + provenance ledger can re-read the source span.
import tempfile, textwrap
md_doc = textwrap.dedent('''
    # Northwind segments

    | Segment | Division |
    | :--- | :--- |
    | Cloud Platform | Operations |
    | Devices | Technology |
    | Logistics | Operations |
    | Retail | Operations |

    | Division | Region |
    | :--- | :--- |
    | Technology | North America |
    | Operations | Europe |
```

```
''').strip()
path = tempfile.NamedTemporaryFile("w", suffix=".md", delete=False).name
open(path, "w").write(md_doc)

# Geometry is authoritative; no LLM is needed to FIND or FIX the relational error
# (the Qwen actor only raises confidence when it agrees). Trainer is injected so
# orchestration is testable; this still trains a GMS -> GPU/CI only.
loop = GeodeLoop(make_default_trainer(epochs=300), check_anchors=False)
result = loop.run(path)

print("converged:", result.converged, "| iterations:", result.iterations)
for c in result.corrections:
    print("fixed:", c["triple"], "-> geometry:", c["geometry"],
          "| residual:", round(c["residual"], 3), "| at", c["location"])
```

The critic flags `('cloud platform','has_division','operations')` with a high composition residual; the geometry's predicted tail is `technology`, the loop rewrites the triple and the graph re-converges. Each correction carries its source `file:line:char` location, recoverable through the same `ProvenanceLedger` introduced in Chapter 3. Unlike the numeric case, this error *is* auto-fixed, because the graph itself contains a second source of truth (the composition path), so the correct value is recoverable rather than merely detectable.

5.4 The limit: a lone value with no redundancy

Consider `total_assets has_amount 540.0` from the balance sheet. It appears once, with no `Total = Σ parts` row over the balance-sheet line items and no second statement of the figure anywhere. If we corrupt it to `999.0`, *neither* mechanism fires: the composition critic requires a redundant relational chain (there is none) and the sum anchor requires declared parts (there are none).

```
# A lone numeric fact: one value, no parts, no duplicate, no composition chain.
lone = [("total assets", "has_amount", "999.0")]   # truth is 540.0

enm_lone = enm_from_triples(lone)
checker_lone = AnchorChecker.from_enm(enm_lone)

# No sum constraint can be derived (no "total" row over parts), and there is no
# duplicate of the same (entity, relation) -> nothing to cross-check.
print("derived sum constraints:", checker_lone.auto_sum_constraints())
print("anchor violations:", checker_lone.check_all(lone))
```

```
derived sum constraints: []
anchor violations: []
```

The corrupted `999.0` passes silently. Geometry corrects what violates redundancy; it cannot invent a cross-check that the document never provided. The remedy is upstream—declare a constraint, add a duplicate statement or carry the figure as an ENM-checked exact value sourced to its one cell (Chapter 3)—rather than a more elaborate critic. GEODE makes confident-wrong *relational* and *aggregate* facts very difficult to introduce, but a single corrupted isolated figure with no redundancy remains the extractor's responsibility. The complementary mechanism, abstaining when no grounded answer is available, is treated in Chapter 13.

Anti-patterns

- **Indexing unverified extractor output.** A regex or LLM extractor produces plausible-looking triples; sending them straight to the index bakes every extraction error into the store. The loop should be run first—the geometry catches the contradictions that the extractor cannot see in its own output.
- **Auto-rewriting a number with no anchor behind it.** The sum anchor flags a broken total but cannot localize the single wrong part; "fixing" it by selecting a part to overwrite manufactures a new error. Detection without a recoverable second source of truth escalates to review; it does not auto-edit.
- **Assuming the loop catches everything.** It does not. A lone value with no redundancy (§5.4) is invisible to both the critic and the anchors. Treating it as caught reproduces the same overconfidence that chunk-and-pray exhibits.

Exercise — the duplicate anchor

The sum anchor cannot localize its break to one part. The *duplicate* anchor can: when the same `(entity, relation)` is asserted twice with conflicting exact values, it names the exact pair. Assert `total has_revenue 350.0` alongside the canonical `355.0` and confirm that `check_duplicates` fires. The worked solution is in `notebooks/05_geode_self_correction.ipynb`.

```
# EXERCISE SOLUTION -- conflicting duplicate of the same (entity, relation).
dup = clean_triples + [("total", "has_revenue", "350.0")]  # conflicts with 355.0

checker_dup = AnchorChecker.from_enm(enm_from_triples(dup))
dups = checker_dup.check_duplicates(dup)
for v in dups:
    print(v.kind, "|", v.message, "| residual:", v.residual)

assert any(d.kind == "duplicate" for d in dups), "duplicate anchor should fire"
```

5.5 Self-check

The chapter's claim is that when a segment figure is corrupted so that Total $\neq \Sigma$ parts, the loop's anchor layer surfaces the break. The notebook's final cell asserts that the injected sum break appears in the anchor violations with the expected residual—the property the brief names. This path is CPU-only (no GMS training, no Qwen), so it runs deterministically in CI; the relational-fix listing of §5.3 is the GPU-gated companion that the lead executes.

```
# Self-check: the injected sum break (Devices 80 -> 88) is detected as a "sum"
# anchor violation. This is the CPU-only anchor path (no GMS training, no Qwen),
# so it runs deterministically in CI.
enm_chk = enm_from_triples(corrupted)
checker_chk = AnchorChecker.from_enm(enm_chk)
viols = checker_chk.check_all(corrupted)

sum_viols = [v for v in viols if v.kind == "sum"]
```

```
assert sum_viols, "expected a sum anchor violation for the corrupted segment"
v = sum_viols[0]
assert abs(v.residual - 8.0) < 1e-6, f"expected residual 8.0, got {v.residual}"
print("OK -- sum anchor flagged the injected break:", v.message)
```

This chapter cleans the graph before it is indexed. Before retrieval, the next part of the book turns the corrected store into its own source of data: Chapter 6 mines it with a designed experiment to generate the evaluation and training cohort, and Chapter 7 tunes the encoders on that data. Retrieval then follows: Chapter 8 poses questions *through* the corrected graph rather than against a vector index, and Chapter 9 develops the binding step that grounds an answer in specific triples. For the design rationale and validation behind the loop—why the composition residual is the load-bearing detector and where its limits were measured—see `EXTRACTION_AGENT_DESIGN.md`; for the geometry internals, see the GMS substrate described in Appendix C.

Part IV

Data Generation and Encoder Tuning

Chapter 6

Generating Data with a Designed Experiment

The store built in Chapter 4 can also serve as an *oracle*. Every fact it holds is a known, graph-derived answer, so we can turn the store itself into the source of the data that trains and tests the rest of the pipeline. This chapter mines the store for questions whose answers are provably correct, then enriches each one across a designed space of presentation conditions, producing one corpus that serves two consumers: the encoder fine-tuning of Chapter 7 and the evaluation of Chapter 15.

The organizing principle is *ground-truth-invariant enrichment*. A question about Cloud Platform's revenue has one answer fixed by the graph; whether it is asked tersely or verbosely, plainly or hesitantly, by a novice or an analyst, the answer does not move. So we separate *what* is asked — a base taxonomy whose answers the GMS supplies — from *how* it is presented — a layer of factors that vary the surface without touching the truth. Crossing the two with a designed experiment gives a corpus where any failure can later be attributed to the presentation factor that caused it, because the answer was never supposed to change. All of this is library functionality: the base generators and the DoE design come from `knowlytix.harness.suite`; we only select and run them.

6.1 Base questions from the store

A `CatalogBaseSource` mines the store with the benchmark generators named in the suite catalog. We select the two content-retrieval base types the report supports — `exact_recall` (read one figure) and `multi_hop` (the highest or lowest figure in a section). Each generated question carries its graph-derived ground truth, so no hand labeling is involved.

```
from knowlytix.harness.suite import (
    Catalog, resolve, CatalogBaseSource, compose, graphdoe_design)

CAT = Catalog.load()                        # the shipped base + factor catalogs
suite = resolve(CAT, ["exact_recall", "multi_hop"],
                ["clarity", "style", "length", "expertise", "paraphrase_depth"],
                mode="embedded")
items = CatalogBaseSource(store, max_per_category=8, seed=42).items(suite)
print(f"{len(items)} base questions, each with graph-derived ground truth")
for it in items[:3]:
    print(f"  [{it.base}] {it.answer!r}")
```

```
14 base questions, each with graph-derived ground truth
  [exact_recall] 120.0
  [exact_recall] 355.0
  [multi_hop] ('Total Assets', 540.0)
```

The generators phrase questions in the graph's own terms. That is fine for an oracle but not how a reader asks; we naturalize the surface in the materialization step below. The two content base types yield 14 seed questions on this corpus. The remaining base types in the catalog (counting, cross-reference, contradiction) ask about graph *structure* rather than report content, and the clean single-document report offers them little to bind to, so we leave them out.

6.2 The designed experiment

Enrichment is a design over the presentation factors. We use *embedded* mode: the base question is itself a factor in the design, so the number of scenarios is exactly the number of runs we ask for — a designed experiment of a chosen size, not a full cross product and not a random sample. The presentation factors are space-filled by a Sobol sequence with a refinement pass, and the base question is assigned as a balanced blocking factor so every seed is exercised about equally.

```
from functools import partial

scns = compose(suite, items, n_runs=150, seed=42,
               design_fn=partial(graphdoe_design, method="sobol+refine"),
               balance_base=True)
print(f"{len(items)} base x designed presentation -> {len(scns)} scenarios")
print("factors:", suite.factor_names)
```

```
14 base x designed presentation -> 150 scenarios
factors: ['clarity', 'style', 'length', 'expertise', 'paraphrase_depth']
```

Five factors of three levels each span a 3^5 presentation space; a full cross of 14 seeds over that space is $14 \times 243 = 3402$ runs. The Sobol design covers that space with 150 runs — the efficiency of a designed experiment over an exhaustive grid — while the blocking keeps the seeds balanced. The alternative, `mode="cross"`, crosses every seed with the same design grid for full balance at #seeds $\times$ n_runs cost; embedded trades a little balance for far fewer runs.

6.3 Materializing contextual questions

Each scenario is a base question plus a row of factor levels. A factor level is turned into a natural-language question by conditioning a local Qwen2.5-3B rewrite on it, holding the target and the ground truth fixed. The rewrite is batched — one `generate` over a padded batch rather than one call at a time — and guarded: it must stay on topic and must not invent a number, a year or a company the target did not name, or it falls back to a plain question. The graph slug (`has_revenue`, `Cloud Platform/Technology/Revenue`) never reaches the model; a deterministic step turns it into a plain target phrase first.

```
# Per scenario: factor levels -> a natural question for the SAME target.
prompt = build_rewrite_prompt(target="revenue of Cloud Platform",
                              levels={"clarity": "Misleading", "style": "Formal",
```

```
                                        "length": "Short", "expertise": "Novice",
                                        "paraphrase_depth": "Light"})
print(qwen_rewrite(prompt))
```

```
I might be misreading the table, but could you help me understand how much
revenue the Cloud Platform segment earned at Northwind Industries?
```

The clarity factor opens the question hesitantly; the target and its answer are untouched. Across the cohort the guard let no fabricated figure through, and a small fraction of rewrites fell back to a plain question.

6.4 One corpus, three consumers

The enriched scenarios are emitted as the artifacts the rest of the book consumes. The test cohort pairs each contextual question with its graph answer; the embedding corpus groups the questions by the attribute they ask about; the contradiction groups and the draft pairs feed the encoder tuning and the generator.

```
# data/enrichment/
#   rag_cohort.json        150 (question, ground-truth answer) test cases
#   embedding_sft.jsonl    150 {text, label=attribute} rows  (Ch7 v-space)
#   embedding_u_groups.json     per-attribute question groups (Ch7 u-space)
#   llm_draft_sft.jsonl    150 grounded (prompt, answer) pairs
```

GMS in practice — the answer is fixed; only the surface varies

A designed experiment is only sound when the response is well defined. Here the response is correctness against a *fixed* answer the graph supplies, so the presentation factors are genuine experimental factors: each one moves the surface of a question whose truth is held constant. That is what lets Chapter 15 attribute a failure to the factor that caused it — a clarity level or a register — rather than to an ambiguous question. Using the GMS as the oracle is what makes the enrichment ground-truth-invariant by construction.

6.5 Limitations

The cohort is only as broad as the generators that mine it. The two content base types cover figure lookup and section extrema; reasoning, comparison and contradiction questions need graph structure this single, clean report does not carry, so they are absent here and would appear on a richer corpus. The materialization uses a language model at build time, so it is not byte-deterministic across hosts; we seed it and guard its output, and the ground truth — which comes from the graph, not the model — is invariant regardless. A guard that falls back to a plain question trades a little surface variety for safety; the fallback rate is reported by the builder so the loss of variety is visible rather than silent.

Anti-patterns

- **Letting the rewriter invent facts.** A model asked to rephrase a question will, unprompted, supply a plausible number or company. The target and its answer must be pinned and the output guarded; a rewrite that introduces an unrequested figure is discarded, not shipped into the cohort.
- **Confusing a sample with a design.** Drawing presentation conditions at random covers the factor space unevenly and starves attribution. Use the space-filling design at the size you want, not a random subset of a grid.
- **Hand-writing the test set.** A hand-authored cohort encodes the author's blind spots and cannot be attributed by factor. Generating from the GMS oracle gives provable answers and a designed factor structure.

Exercise — widen the design

Re-run the enrichment with a sixth factor (`noise`, which injects typographical artifacts) added to the suite, holding `n_runs` at 150. Confirm the cohort still has 150 cases and that the new factor is balanced across its levels. The worked solution is in the chapter notebook.

6.6 Self-check

The notebook asserts the enrichment is sound: every cohort case carries a graph-derived answer, the design has exactly the requested number of runs, and the presentation factors are present and balanced.

```
import json, collections
cohort = json.load(open("data/enrichment/rag_cohort.json"))
assert len(cohort) == 150
assert all(c["expected_answer"] is not None for c in cohort)
factors = collections.Counter(c["_factors"]["clarity"] for c in cohort)
assert min(factors.values()) >= 30      # each clarity level well represented
print("OK: 150 designed cases, all with ground truth, factors balanced")
```

With the corpus generated, Chapter 7 uses it to tune the GMS encoders — the questions teach the encoders the attribute in context — and Chapter 15 uses the same cohort to test the assembled RAG with the GMS as the oracle.

Chapter 7

Tuning the Encoders: Embedding SFT

Retrieval and the relevance gate both rest on an encoder that turns text into a vector. A frozen, general-purpose sentence encoder treats "topline" and "revenue" as merely similar strings and has no notion at all of two claims *contradicting* each other. This chapter tunes the encoders on the data generated in Chapter 6 so the geometry carries the two relationships the GMS needs: *semantic proximity* in the v-space (a paraphrased question lands near the attribute it asks about) and *logical contradiction* in the u-space (a claim that states the wrong value for a fact sits far from the truth in tension). Both are supervised fine-tunes of a small low-rank adapter over a frozen base encoder, from `knowlytix.embedding`; the base embedding never moves, only the adapter trains.

7.1 v-space: concepts that must be close

The v-space encoder decides which attribute a question is about. We tune it on `embedding_sft.jsonl` — the contextual questions from Chapter 6, each labeled with the attribute it asks for — with a prototype objective: questions about the same attribute are pulled toward a shared prototype, attributes are pushed apart. Because the supervision is the *contextual* question, not a bare keyword, the encoder learns the attribute *in context*: a full sentence about revenue lands near the revenue prototype even when the word "revenue" never appears.

```
from knowlytix.embedding import EmbeddingSFTConfig, finetune_embedding

v_ft = finetune_embedding(
    "data/enrichment/embedding_sft.jsonl",
    EmbeddingSFTConfig(rank=8, mode="full", out_dim=store.model.cfg.d_v,
                       objective="prototype"),
    text_col="text", label_col="label")
print(f"v-space val_accuracy = {v_ft.val_accuracy:.3f}")

v_ft.save("data/gms_annual_report_store/tuned_encoder")
ent_names = sorted({e for h, _r, t in store.triples for e in (h, t)})
torch.save(v_ft.export_vectors(ent_names),
           "data/gms_annual_report_store/v_emb.pt")     # GMS Mode-B vectors
```

```
v-space val_accuracy = 0.966
```

The tuned encoder separates the eight attributes of this corpus with 0.97 held-out accuracy. It is saved for query-time use, where the retrieval parser of Chapter 8 and the relevance gate of Chapter 14 both run on it. Its transform is exported as name-keyed entity vectors that warm-start the store's own v-space.

7.2 u-space: terms that genuinely contradict

The u-space encoder is the logical half: it must give *low* tension to two statements that agree and *high* tension to two that conflict. The supervision has to be genuine contradiction, not merely different topics. Two questions, one about revenue and one about headcount, are not contradictory — they are about different things. A contradiction is two claims that cannot both be true: the same fact asserted with different values. We mine those from the store — each (entity, attribute) paired with its true value (consistent, reworded) versus a different real figure from the report (contradictory) — and train the adapter pairwise.

```
pos, neg = contradiction_pairs(store)      # same fact: same value vs wrong value
print(f"{len(pos)} consistent pairs, {len(neg)} contradictory pairs")

u_ft = fit_contradiction_pairs(pos, neg,
    EmbeddingSFTConfig(rank=32, mode="full", objective="contradiction",
                       encoder="sentence-transformers/nli-mpnet-base-v2",
                       out_dim=store.model.cfg.d_v, epochs=400, margin=1.3))
u_ft.save("data/gms_annual_report_store/contradiction_encoder")
```

```
63 consistent pairs, 189 contradictory pairs
tension  consistent (same value)  = 0.065
tension  contradictory (wrong val)= 1.353
```

Two ways of stating the same figure sit at tension 0.07; a statement of the wrong figure sits at 1.35, near the maximum. The encoder has learned to separate a differently-worded-but-true claim from a false one — the signal the self-verifier of Chapter 12 reads to catch a confident wrong number.

GMS in practice — a rotation cannot move tension

The contradiction objective requires the `full` adapter mode, not `rotation`. A rotation preserves all angles, so it cannot change the tension between any two vectors — it would leave a contradiction looking exactly as consistent as the truth. Only a transform that can stretch the space (the `full` mode) can pull conflicting claims apart while keeping agreeing ones together. Training u-space on different-*topic* pairs instead of genuine conflicting-*value* pairs has the same failure: the encoder collapses all tension toward zero and the veto never fires.

7.3 Calibrating the relevance gate

With both encoders tuned, the relevance gate's operating point is fit from the same data rather than guessed — the v-accept floor and the per-attribute u-veto cut — and persisted beside the

store. The gate itself is assembled and exercised in Chapter 14.

```
from knowlytix.knowledge.rag.relevance import calibrate_relevance_thresholds

relcal = calibrate_relevance_thresholds(u_groups, v_ft.encode, u_ft.encode)
json.dump(relcal,
          open("data/gms_annual_report_store/relevance_calibration.json", "w"))
```

7.4 Limitations

The encoders are only as good as the generated data behind them. The v-space is tuned on eight attributes; a corpus with more relations needs proportionally more supervision to separate them. The u-space contradictions here are value-conflicts on numeric facts, which the report supplies in abundance; a corpus whose facts are categorical rather than numeric would need a different way to construct genuine conflicts. Both fine-tunes run at build time and are seeded but not byte-deterministic across hosts. Finally, a high held-out accuracy on the generated questions is not a guarantee on arbitrary phrasings far outside the factor design; the evaluation of Chapter 15 measures the encoders where it matters, in the assembled pipeline, rather than in isolation.

Anti-patterns

- **Training u-space on different topics.** Grouping "revenue" questions against "headcount" questions teaches the encoder that everything unalike is contradictory, which collapses to no usable tension. The contradiction must be a conflicting *value* for the same fact.
- **Using a rotation for the contradiction encoder.** A norm-preserving rotation cannot change tension at all; the contradiction objective demands the full stretch-and-rotate transform.
- **Tuning on bare keywords.** An encoder tuned on the word "revenue" never learns that "how much did the segment sell" is the same attribute. The supervision must be the contextual question the reader actually asks.

Exercise — break the u-space on purpose

Retrain the contradiction encoder in `rotation` mode (or on the different-topic attribute groups instead of conflicting-value pairs) and re-measure the consistent and contradictory tensions. Confirm they no longer separate. The worked solution is in the chapter notebook.

7.5 Self-check

The notebook asserts what the two fine-tunes are supposed to deliver: the v-encoder separates attributes on held-out questions, and the u-encoder gives genuinely contradictory claims higher tension than consistent ones.

```
assert v_ft.val_accuracy >= 0.85
cons = uspace_tension(u_ft, "Retail's headcount was 520.",
                      "The headcount of Retail is 520.")
```

```
contra = uspace_tension(u_ft, "Retail's headcount was 520.",
                        "Retail's headcount was 210.")
assert contra > cons + 0.5
print(f"OK: v-acc {v_ft.val_accuracy:.2f}; tension {cons:.2f} vs {contra:.2f}")
```

The tuned encoders now carry both relationships the pipeline needs. Chapter 8 parses questions into query triples on the v-encoder, Chapter 9 binds terms to the graph through it, and the u-encoder is the contradiction signal behind the self-verifier (Chapter 12) and the relevance veto (Chapter 14).

Part V

Retrieval

Chapter 8

Triple-Mediated Retrieval

In the chunk-and-pray pipeline of Chapter 1 a question is handed, verbatim, to a vector index: the natural-language string is embedded, the nearest chunks are returned and the model is trusted to read an answer out of them. We depart from that design at the first step. A question is instead *translated into query triples*—`(head, relation, tail)` patterns in which the asked-for value is the bare variable `?`—and those triples are bound to the graph's vocabulary and answered *through* the geometric memory introduced in Chapter 2. The surface form of the question never touches an opaque similarity search.

Retrieval, on this account, is a *grounding* discipline. The query triple is the grounding contract: it names the entity, the relation and the slot we are asking about, in the graph's own terms. Everything downstream—binding (Chapter 9), answering through the store (Chapter 10), verification (Chapter 12) and abstention (Chapter 13)—depends on the correctness of this translation.

All listings in this chapter are grounded in `data/corpus_facts.md` (Northwind Industries FY2025). The deterministic cells inject a scripted `LLMBackend` so the chapter runs identically in CI; the real Qwen2.5-3B-Instruct path is shown in one listing, tagged for the lead to execute on the shared GPU. The trained store is not loaded during authoring.

8.1 The query-triple vocabulary

Three symbols define the query-triple vocabulary. `QueryTriple` is the pattern. `ASKED` is the bare `?` sentinel marking the one slot whose value answers the question. `QueryTripleExtractor` turns natural language into a list of query triples, with an injectable LLM backend. `schema_from_store` reads the graph's relation and entity names so that extraction is *grounded*—the principal reliability lever for a small model, which otherwise invents relation names that never bind.

```
from knowlytix.knowledge.rag.query_triples import (
    ASKED, QueryTriple, QueryTripleExtractor, is_var, schema_from_store,
)
from knowlytix.knowledge.llm_backend import LLMBackend

print("asked-slot sentinel:", repr(ASKED))
print("is_var('?'):", is_var(ASKED), "| is_var('?x'):", is_var("?x"),
      "| is_var('cloud platform'):", is_var("cloud platform"))
```

A slot is a variable if and only if it starts with `?`. The bare `?` is the asked value; `?x`, `?y` are intermediate unknowns that link triples in a multi-hop question.

8.2 A deterministic LLM for CI

`QueryTripleExtractor` accepts any `LLMBackend`. For reproducible CI we inject a scripted backend that replays the JSON a correctly grounded Qwen returns for the five questions. The extractor code is identical to the production path; only the backend is swapped. The same extraction logic runs deterministically here and against the real model in §8.6.

```
class ScriptedBackend(LLMBackend):
    """Replays canned JSON keyed by question  stands in for Qwen in CI."""

    def __init__(self, script: dict[str, str], default: str = "[]"):
        self._script = script
        self._default = default

    def call(self, system: str, user: str, max_tokens: int = 2048) -> str:
        question = user.split("\n\nNote:")[0].strip()
        return self._script.get(question, self._default)

    @property
    def model_name(self) -> str:
        return "scripted/qwen2.5-3b-instruct"


# Canned extractions: the JSON a schema-grounded Qwen returns for each question.
SCRIPT = {
    "What was Cloud Platform revenue?":
        '[{"head":"cloud platform","relation":"has_revenue","tail":"?"}]',
    "What was total revenue for FY2025?":
        '[{"head":"total","relation":"has_revenue","tail":"?"}]',
    "How many people work in Logistics?":
        '[{"head":"logistics","relation":"has_headcount","tail":"?"}]',
    "Which division does Cloud Platform belong to?":
        '[{"head":"cloud platform","relation":"has_division","tail":"?"}]',
    "In which region is Cloud Platform's division?":
        '[{"head":"cloud platform","relation":"has_division","tail":"?x"},'
        '{"head":"?x","relation":"has_region","tail":"?"}]',
}
extractor = QueryTripleExtractor(ScriptedBackend(SCRIPT))
```

8.3 Five questions become query triples

Factual, numeric and multi-hop questions all reduce to triple patterns. In every case the asked value is the bare `?`; the multi-hop question links two triples through the named variable `?x`.

```
QUESTIONS = [
    "What was Cloud Platform revenue?",          # factual
    "What was total revenue for FY2025?",        # numeric / exact
    "How many people work in Logistics?",        # factual count
    "Which division does Cloud Platform belong to?",  # hierarchy
    "In which region is Cloud Platform's division?",  # multi-hop (?x -> ?)
]

for q in QUESTIONS:
```

```
    qts = extractor.extract(q)
    asked = sum(t.tail == ASKED for t in qts)
    print(f"Q: {q}")
    for t in qts:
        print(f"   {t.as_tuple()}")
    print(f"   asked-slots: {asked}, intermediate vars: "
          f"{sorted({s for t in qts for s in t.as_tuple() if is_var(s) and s != ASKED})
    }\n")
    assert asked == 1, f"expected exactly one '?' slot for: {q}"
```

The first four questions yield a single triple each: `('cloud platform', 'has_revenue', '?')`, `('total', 'has_revenue', '?')`, `('logistics', 'has_headcount', '?')` and `('cloud platform', 'has_di` The numeric question ("total revenue for FY2025") is not special at extraction time: it has the same shape as a factual lookup. The exactness of the number is enforced later, by the store's exact numerical memory (Chapter 10), not by parsing it out of prose.

8.3.1 Multi-hop: chaining through a named variable

The fifth question merits closer study. `In which region is Cloud Platform's division?` cannot be a single `(head, relation, ?)` lookup: the graph has no `cloud platform -> has_region` edge. The extractor instead emits a two-triple chain—`cloud platform has_division ?x`, then `?x has_region ?`—in which `?x` is the division resolved in the first hop and substituted into the second.

```
multi = extractor.extract("In which region is Cloud Platform's division?")
assert len(multi) == 2
assert multi[0].tail == "?x" and multi[1].head == "?x"   # the chain link
assert multi[1].tail == ASKED                            # the asked value
print("hop 1:", multi[0].as_tuple())
print("hop 2:", multi[1].as_tuple())
print("expected resolution (corpus_facts): ?x=technology, ?=north america")
```

Per `corpus_facts.md`, `?x` binds to `technology` and the final `?` resolves to `north america`. The query-triple form makes the hop structure explicit and inspectable *before* any answering happens; a vector query over "region of Cloud Platform's division" would collapse both hops into one opaque similarity score with no means of auditing the intermediate result.

8.4 Schema grounding fixes a non-binding relation

Left to its own vocabulary, a small model emits plausible but incorrect relation names: `headcount`, `staff_count`, `topline`. None of those exist in the graph; the actual relations are `has_revenue`, `has_headcount`, `has_division`, `has_region` and so on. `schema_from_store` injects the actual relation names into the prompt so that the model emits binding-compatible triples.

We contrast two scripted backends—*ungrounded* (the model guesses `staff_count`) and *grounded* (it selects the actual `has_headcount`)—to illustrate the failure mode and its correction without consuming GPU time.

```
# Same question, two model behaviours.
UNGROUNDED = {"How many people work in Logistics?":
              '[{"head":"logistics","relation":"staff_count","tail":"?"}]'}
GROUNDED = {"How many people work in Logistics?":
            '[{"head":"logistics","relation":"has_headcount","tail":"?"}]'}
```

```
# The graph's real relation vocabulary (from corpus_facts.md / schema_from_store).
GRAPH_RELATIONS = ["has_amount", "has_division", "has_fy2024", "has_fy2025",
                   "has_head", "has_headcount", "has_region", "has_revenue",
                   "has_value"]

def relation_binds(rel: str) -> bool:
    return rel in GRAPH_RELATIONS

q = "How many people work in Logistics?"
bad = QueryTripleExtractor(ScriptedBackend(UNGROUNDED)).extract(q)[0]
good = QueryTripleExtractor(ScriptedBackend(GROUNDED)).extract(q)[0]

print("ungrounded relation:", bad.relation, "-> binds?", relation_binds(bad.relation))
print("grounded   relation:", good.relation, "-> binds?", relation_binds(good.relation)
    )
assert not relation_binds(bad.relation)   # staff_count is not in the graph
assert relation_binds(good.relation)      # has_headcount is
```

`schema_from_store(store)` produces that relation list at run time. It returns `{'relations': [...], 'entit` filtering out the `in_section` bookkeeping relation and any numeric value-entities (those are answers, not query terms). Passed as `vocab=`, it is folded into the system prompt by the extractor.

```
expected_vocab = {
    "relations": ["has_amount", "has_division", "has_fy2024", "has_fy2025",
                  "has_head", "has_headcount", "has_region", "has_revenue",
                  "has_value"],
    "entities": ["cloud platform", "devices", "logistics", "retail",
                 "technology", "operations", "total", "revenue",
                 "net income", "operating income"],
}
grounded_extractor = QueryTripleExtractor(
    ScriptedBackend(SCRIPT), vocab=expected_vocab
)
# The vocab block is injected into the system prompt the extractor sends.
sys_prompt = grounded_extractor.llm  # backend; prompt assembled inside .extract
assert "has_headcount" in expected_vocab["relations"]
assert "in_section" not in expected_vocab["relations"]   # filtered out
print("grounded relations:", expected_vocab["relations"])
```

GMS in practice — schema-grounded extraction vs. a free-text query

The chunk-and-pray default sends the raw question to an embedding model and relies on the nearest chunk being relevant. The geometric counterpart sends the question *together with the graph's vocabulary* to the extractor, so that the output is a triple whose relation is a candidate for binding. The free relation name an ungrounded model emits (`staff_count`) does not exist in the graph and is rejected downstream; the grounded relation (`has_headcount`) resolves. Grounding does not enlarge the model; it makes the model's output *checkable*.

8.5 The query-only repair fallback

Grounding reduces but does not eliminate mis-extraction. The `_extract_and_bind` method of `RagPipeline` runs a *query-only* repair loop: if a non-variable slot fails to bind, it re-asks the extractor once with a hint listing the terms that did not match and the actual relations to use instead. This is a query-side repair; it never edits the graph. Correcting the graph itself is the role of GEODE (Chapter 5); here we correct only the *question's* encoding.

```
class RepairingBackend(LLMBackend):
    """First call mis-extracts; a follow-up with a hint corrects it."""

    def call(self, system: str, user: str, max_tokens: int = 2048) -> str:
        if "Note:" in user:   # the repair retry carries a hint
            return '[{"head":"logistics","relation":"has_headcount","tail":"?"}]'
        return '[{"head":"logistics","relation":"headcount","tail":"?"}]'

    @property
    def model_name(self) -> str:
        return "scripted/repairing"


rb = QueryTripleExtractor(RepairingBackend())
first = rb.extract("How many people work in Logistics?")[0]
repaired = rb.extract("How many people work in Logistics?",
                      hint="These terms did not match: ['headcount']. "
                           "Re-express using ONLY: has_headcount.")[0]
print("first pass :", first.as_tuple(), "-> binds?", relation_binds(first.relation))
print("after repair:", repaired.as_tuple(), "-> binds?", relation_binds(repaired.
    relation))
assert not relation_binds(first.relation)
assert relation_binds(repaired.relation)
```

The first pass emits `headcount` (no `has_` prefix, does not bind); the hinted retry emits `has_headcount`. The loop runs at most `max_extract_retries` times and stops as soon as all slots bind, so a well-grounded first pass incurs no additional cost.

8.6 The real Qwen path

The preceding sections used scripted backends for determinism. The production wiring is a local Qwen2.5-3B-Instruct via `LocalTransformersBackend`, grounded with the live `schema_from_store`. This listing loads the trained store and a GPU model, so the notebook tags it `ci-gpu`; the lead executes it, and it is not run during a shared-GPU authoring session.

```
# CI-ONLY (ci-gpu): loads the trained store + a GPU Qwen. Do not run while authoring.
import torch
from knowlytix.knowledge.config import DocGMSConfig
from knowlytix.knowledge.store import GMSExpertStore
from knowlytix.knowledge.llm_backend import LocalTransformersBackend
from knowlytix.knowledge.rag.query_triples import (
    QueryTripleExtractor, schema_from_store, ASKED,
)
```

```
REPO_ROOT = os.path.join(os.path.dirname(os.getcwd()), "code") if os.path.basename(os.
    getcwd()) == "notebooks" else os.getcwd()
STORE = os.path.join(REPO_ROOT, "data", "gms_annual_report_store")
dev = "cuda" if torch.cuda.is_available() else "cpu"
store = GMSExpertStore(DocGMSConfig(store_path=STORE), device=torch.device(dev))
assert store.load(), "trained store not found  build it with scripts/build_store.py"

vocab = schema_from_store(store)
qwen = LocalTransformersBackend("Qwen/Qwen2.5-3B-Instruct", device=dev)
extractor_qwen = QueryTripleExtractor(qwen, vocab=vocab)

q = "How many people work in Logistics?"
qts = extractor_qwen.extract(q)
print("Qwen ->", [t.as_tuple() for t in qts])
assert any(t.tail == ASKED for t in qts)          # an asked slot was produced
assert any(t.relation == "has_headcount" for t in qts)  # grounding bound it
```

The query-time LLM is the same local Qwen the agent book pins for its memory tier; no API key and no hosted provider are required. The grounded vocabulary turns a 3B model's noisy first guess into a relation that binds.

Anti-patterns

- **Free-text "retrieval queries."** Embedding the raw question and trusting the nearest chunk yields no asked slot, no relation and nothing to bind or audit. The query triple is the contract; a string is not.
- **Trusting a small model's first extraction.** A 3B model guesses relation names that do not exist (`staff_count`, `topline`). Without schema grounding and the query-only repair loop, those guesses silently fail to bind or bind to the wrong relation.
- **Putting a guessed value in the asked slot.** Filling the queried slot with a placeholder number or the word "value" instead of the bare `?` turns the question into an assertion the system attempts to verify rather than answer. The extractor guards against this; a hand-written query triple must do so as well.

Exercise — find a question Qwen mis-extracts, then fix it with grounding

`What is Northwind's topline?` invites the wrong relation: `topline` is a declared alias for revenue, but an ungrounded model emits `topline` as the relation, which does not bind. The exercise is to exhibit the ungrounded miss, then the grounded correction that maps it to `has_revenue` (binding proper is the subject of Chapter 9). The worked solution is in the chapter notebook.

```
UNGROUNDED_TL = {"What is Northwind's topline?":
                 '[{"head":"northwind","relation":"topline","tail":"?"}]'}
GROUNDED_TL = {"What is Northwind's topline?":
               '[{"head":"total","relation":"has_revenue","tail":"?"}]'}

miss = QueryTripleExtractor(ScriptedBackend(UNGROUNDED_TL)).extract(
    "What is Northwind's topline?")[0]
fix = QueryTripleExtractor(ScriptedBackend(GROUNDED_TL)).extract(
```

```
    "What is Northwind's topline?")[0]
print("ungrounded:", miss.as_tuple(), "-> binds?", relation_binds(miss.relation))
print("grounded  :", fix.as_tuple(), "-> binds?", relation_binds(fix.relation))
assert not relation_binds(miss.relation)
assert relation_binds(fix.relation) and fix.tail == ASKED
```

8.7 Limitations

Query-triple extraction translates a question into a checkable form; it does not itself *answer* the question. Two limitations follow. First, extraction can be syntactically perfect and still bind the wrong entity: a paraphrase or alias ("topline", a segment nickname) requires the binding machinery of Chapter 9 to resolve, and an ambiguous term is refused rather than guessed. Second, grounding helps only for attributes the graph actually holds. If a question asks about something the document never recorded as a triple (an MD&A narrative claim, for example), no degree of schema grounding will produce a binding relation, and the appropriate outcome is abstention (Chapter 13), not a forced answer. The extractor is deliberately instructed *not* to substitute a listed relation for an attribute the entity does not have—an interest rate is not a fee—so that downstream binding and coverage checks surface the gap rather than mask it.

8.8 Self-check

The chapter's claim is that natural-language questions become query triples with exactly one `?` asked slot and that schema grounding produces a relation that binds. The notebook's final cell demonstrates both for a known question.

```
qts = grounded_extractor.extract("How many people work in Logistics?")
assert len(qts) == 1
t = qts[0]
assert t.tail == ASKED                      # the asked slot
assert t.head == "logistics"                # the subject was extracted
assert t.relation == "has_headcount"        # grounded to a real graph relation
assert t.relation in expected_vocab["relations"]   # ... that binds
print("OK  query-triple extraction yields a '?' slot that binds:", t.as_tuple())
```

With the question encoded as a binding-compatible query triple, the next chapter takes up *binding* (Chapter 9): resolving paraphrases, nicknames and the `has_<slug>` convention to the graph's canonical entities, and refusing rather than mis-resolving an ambiguous term.

Chapter 9

Binding query terms to the graph

Chapter 8 turned a question into query triples: a pattern like `("cloud", "revenue", "?")` with the asked value marked `?`. Those triples are written in the *user's* vocabulary, not the graph's. A user typing *"topline"* addresses a store that holds `has_revenue`; a user saying *"consumer hardware"* addresses a graph that knows `devices`. **Binding** is the step that resolves each non-variable slot to a term the store actually contains, or, when the term is genuinely ambiguous, refuses rather than guess.

A retrieval system that silently mis-resolves *"the unit"* to whichever segment sorts first returns a confident, wrong, well-provenanced answer — the most damaging failure mode, because the provenance makes it appear trustworthy. We treat an unbound slot as a designed outcome rather than a defect. It feeds the bind-check abstention of Chapter 13, so the pipeline declines instead of fabricating. The provenance ledger that makes a binding auditable was introduced in Chapter 3.

We use the real `knowlytix.knowledge.rag.binding` module throughout. To keep the chapter deterministic and CPU-only, the listings run against a small fake store that exposes exactly the interface `TripleBinder` consumes — `store.adapter.entity_to_idx`, `store.adapter.relation_to_idx` and the shipped `store.fuzzy_match_entity`. The trained Northwind store from the foundation build has the same vocabulary; nothing about the binder changes when it runs against the production store.

Every notebook begins with the library bootstrap so imports resolve to the working branch:

```
import os, sys
KNOWLYTIX_SRC = os.environ.get("KNOWLYTIX_SRC", "/home/asudjianto/jupyterlab/GMS-
    knowlytix")
sys.path.insert(0, KNOWLYTIX_SRC)
```

9.1 A stand-in store with the Northwind vocabulary

`TripleBinder` reads only two vocabulary maps and the entity matcher. We reuse the shipped `GMSExpertStore.fuzzy_match_entity` verbatim so the matching rules (unique exact / case-insensitive / substring) are the production ones, then populate the adapter with the Northwind FY2025 entities and relations from the corpus.

```
from knowlytix.knowledge.store import GMSExpertStore


class _Adapter:
    """Minimal stand-in for GraphToGMS: just the two vocab maps TripleBinder reads."""
```

```
    def __init__(self, entities: list[str], relations: list[str]):
        self.entity_to_idx = {e: i for i, e in enumerate(entities)}
        self.relation_to_idx = {r: i for i, r in enumerate(relations)}


class FakeStore:
    """Northwind FY2025 vocabulary with the real fuzzy_match_entity behaviour."""
    def __init__(self):
        self.adapter = _Adapter(
            entities=["cloud platform", "devices", "logistics", "retail",
                      "total", "technology", "operations", "revenue"],
            relations=["has_revenue", "has_headcount", "has_division",
                       "has_region", "has_head", "in_section"],
        )

    # Reuse the shipped bank-grade matcher verbatim (unique exact / ci / substring).
    fuzzy_match_entity = GMSExpertStore.fuzzy_match_entity


store = FakeStore()
sorted(store.adapter.entity_to_idx), sorted(store.adapter.relation_to_idx)
```

9.2 Fuzzy binding: substring resolves, paraphrase does not

`TripleBinder(mode="fuzzy")` resolves entities with `store.fuzzy_match_entity` and relations with a `has_<slug>` matcher — the table-column convention that maps a column heading *Revenue* to the relation `has_revenue`. It handles a substring nickname (`"cloud"` → `cloud platform`) and the column relation. A true paraphrase such as `"topline"` or `"consumer hardware"` shares no substring with any graph term, so fuzzy binding returns `None`: the slot is unbound and `BoundTriple.bound` is `False`.

```
from knowlytix.knowledge.rag import TripleBinder
from knowlytix.knowledge.rag.query_triples import QueryTriple

fuzzy = TripleBinder(store, mode="fuzzy")

# (a) substring nickname + table-column relation -> binds
ok = fuzzy.bind(QueryTriple("cloud", "revenue", "?"))
print("cloud/revenue ->", ok.head, ok.relation, ok.tail, "| bound:", ok.bound)

# (b) "topline" alias for revenue -> fuzzy cannot reach has_revenue
top = fuzzy.bind(QueryTriple("cloud platform", "topline", "?"))
print("topline       ->", top.head, top.relation, top.tail, "| bound:", top.bound)

# (c) "consumer hardware" synonym for devices -> no substring, no bind
syn = fuzzy.bind(QueryTriple("consumer hardware", "headcount", "?"))
print("consumer hw   ->", syn.head, syn.relation, syn.tail, "| bound:", syn.bound)
```

Output:

```
cloud/revenue -> cloud platform has_revenue ? | bound: True
topline       -> cloud platform None ? | bound: False
consumer hw   -> None has_headcount ? | bound: False
```

`"cloud"` is a unique substring of `cloud platform`; `"revenue"` slugs to `has_revenue`. `"topline"` and `"consumer hardware"` are genuine paraphrases that string matching cannot reach. Note that a bound triple requires *every* non-variable slot to resolve — the `?` slot is a variable and never counts against binding.

9.3 Embedding binding: the injectable encoder

`mode="embedding"` keeps exact and fuzzy matches authoritative, then falls back to a nearest-neighbor cosine match over an injectable encoder (`list[str] -> (N, d)`). In production the default is the repository's MiniLM encoder (`knowlytix.core.graph.encoders.encode_texts`, downloaded on first use); for a reproducible, GPU-free chapter we inject a deterministic 4-dimensional encoder whose axes are `[revenue, devices, cloud, headcount]`. Synonyms land on the same axis as the canonical Northwind term.

```
import torch

# Deterministic semantic axes: [revenue, devices, cloud, headcount].
# Synonyms land on the same axis as the canonical Northwind term.
def fake_encode(texts: list[str]) -> torch.Tensor:
    rows = []
    for t in texts:
        tl = t.lower()
        rows.append([
            1.0 if ("revenue" in tl or "topline" in tl or "sales" in tl) else 0.0,
            1.0 if ("devices" in tl or "hardware" in tl or "gadget" in tl) else 0.0,
            1.0 if ("cloud" in tl or "platform" in tl) else 0.0,
            1.0 if ("headcount" in tl or "staff" in tl or "employees" in tl) else 0.0,
        ])
    return torch.tensor(rows)

emb = TripleBinder(store, mode="embedding", encoder=fake_encode)

# "topline" -> has_revenue (the GMS box), "consumer hardware" -> devices
bt = emb.bind(QueryTriple("consumer hardware", "topline", "?"))
print("consumer hardware / topline ->", bt.head, bt.relation, bt.tail)
print("bound:", bt.bound)
```

Output:

```
consumer hardware / topline -> devices has_revenue ?
bound: True
```

Embedding binding succeeds exactly where fuzzy failed: `"topline"` shares the revenue axis with `has_revenue`, `"consumer hardware"` shares the devices axis with `devices`, and the asked slot `?` passes through untouched. Exact and fuzzy hits still win first — the embedding space is consulted only when string matching returns nothing (`binding.py`, `_bind_entity`). The production form omits the injected encoder so the binder lazily wraps the real MiniLM encoder; it is run in CI rather than here because the authoring GPU is shared:

```
# CI-ONLY (downloads MiniLM): real default encoder, no fake injected.
# from knowlytix.knowledge.rag import TripleBinder
# real = TripleBinder(store, mode="embedding")          # encoder=None -> encode_texts
# bt = real.bind(QueryTriple("topline", "topline", "?")) # paraphrase of revenue
```

```
# print(bt.head, bt.relation)  # expect a revenue-family relation
```

GMS in practice — "topline" → `has_revenue`

A chunk-and-pray pipeline treats *"what is the topline?"* as an opaque query string, embeds it whole and retrieves whichever chunk scores highest, on the expectation that the word *revenue* appears nearby. GEODE-RAG instead binds the *relation slot* `"topline"` to the graph term `has_revenue`, then answers *through* the graph. The alias is resolved once, explicitly, and the answer is the ENM-exact figure with provenance — not a guess drawn from text that happened to rank well. The chunk-and-pray baseline this contrasts with is the subject of Chapter 1.

9.4 Refuse on ambiguity, do not mis-resolve

Two safeguards constrain binding. First, a match below `bind_threshold` cosine similarity is rejected as out-of-vocabulary. Second, when the top two candidates fall within `bind_margin` of each other, the match is refused as a near tie: the system abstains rather than choose one of two plausible terms. Here `"the unit"` matches nothing (a zero vector, cosine 0) and a deliberately ambiguous encoder makes `"growth"` tie between `has_revenue` and `has_headcount`.

```
# (a) out-of-vocabulary term -> below threshold -> None
oov = emb.bind(QueryTriple("the unit", "revenue", "?"))
print("the unit ->", oov.head, "| bound:", oov.bound)

# (b) a term that ties between two relations -> refuse rather than guess.
# This encoder puts "growth" equidistant from has_revenue and has_headcount.
def tie_encode(texts: list[str]) -> torch.Tensor:
    rows = []
    for t in texts:
        tl = t.lower()
        if "growth" in tl:
            rows.append([0.7, 0.7, 0.0, 0.0])            # ambiguous query
        else:
            rows.append([
                1.0 if "revenue" in tl else 0.0,
                1.0 if "headcount" in tl else 0.0,
                0.0, 0.0,
            ])
    return torch.tensor(rows)

tie = TripleBinder(store, mode="embedding", encoder=tie_encode, bind_margin=0.1)
amb = tie.bind(QueryTriple("cloud platform", "growth", "?"))
print("growth   ->", amb.relation, "| bound:", amb.bound)
```

Output:

```
the unit -> None | bound: False
growth   -> None | bound: False
```

`"the unit"` encodes to the zero vector, cosine $0 <$ `bind_threshold` (0.5) — unbound. `"growth"` sits halfway between two relations; the gap to the runner-up is below `bind_margin` — refused. Both

feed the bind-check abstention of Chapter 13: an unbound query triple means the pipeline declines to answer rather than fabricate a binding.

9.5 Limits

Binding decides *whether* a term resolves, not whether the chosen `bind_threshold` and `bind_margin` are appropriate for a given corpus. The values above (0.5 / 0.05) are defaults rather than fixed truths: too low and *"the unit"* begins resolving to a real segment; too high and a legitimate paraphrase is refused. The binder also has no notion of whether a bound relation *applies* to its bound entity — `("retail", "has_region", "?")` binds cleanly even though region is asserted on divisions, not segments; that relevance check occurs downstream during retrieval (Chapter 10). Embedding binding is only as good as its encoder: the injected fake here is a teaching device, whereas the real MiniLM encoder makes different, occasionally surprising, neighbor choices. Choosing the operating thresholds from a labeled cohort rather than by inspection is the subject of Chapter 14.

Anti-patterns

Silent mis-resolution on a tie. Picking the first or highest-scoring candidate when two are within a hair of each other turns an ambiguous question into a confident wrong answer. Refuse and abstain instead.

Keyword-only entity matching. Substring matching cannot reach a paraphrase that shares no characters with the canonical term (*"consumer hardware"* vs. `devices`). Treating fuzzy matching as sufficient silently drops every synonym question.

Embedding everything by default. Embedding binding is a fallback, not the front door. Consulting it before exact/fuzzy matching lets a fuzzy neighbour override an exact graph term and erodes the determinism that makes answers auditable.

Exercise — Add a synonym and confirm embedding binding picks it up

Northwind's `logistics` segment is sometimes called *"shipping"* internally. Extend the encoder with a logistics/shipping axis, bind `("shipping", "topline", "?")`, and confirm it resolves to `logistics` / `has_revenue`. The worked solution is the final code cell of `notebooks/07_binding.ipynb`:

```
def exercise_encode(texts: list[str]) -> torch.Tensor:
    rows = []
    for t in texts:
        tl = t.lower()
        rows.append([
            1.0 if ("revenue" in tl or "topline" in tl) else 0.0,
            1.0 if ("logistics" in tl or "shipping" in tl or "freight" in tl)
    else 0.0,
        ])
    return torch.tensor(rows)

ex = TripleBinder(store, mode="embedding", encoder=exercise_encode)
sol = ex.bind(QueryTriple("shipping", "topline", "?"))
assert sol.head == "logistics" and sol.relation == "has_revenue"
print("shipping -> ", sol.head, "/", sol.relation)
```

9.6 Self-check

The chapter's claim — *a paraphrase binds to the canonical Northwind entity, and an ambiguous term is refused* — is proved by the notebook's final assertion:

```
# Paraphrase binds to the canonical entity + relation.
bound = emb.bind(QueryTriple("consumer hardware", "topline", "?"))
assert bound.head == "devices", bound.head
assert bound.relation == "has_revenue", bound.relation
assert bound.bound is True

# Ambiguous / OOV terms refuse rather than mis-resolve.
assert tie.bind(QueryTriple("cloud platform", "growth", "?")).relation is None
assert emb.bind(QueryTriple("the unit", "revenue", "?")).head is None

# Exact match stays authoritative even in embedding mode.
exact = emb.bind(QueryTriple("cloud platform", "has_revenue", "?"))
assert exact.head == "cloud platform" and exact.relation == "has_revenue"

print("OK: paraphrases bind, ambiguity refused, exact match authoritative.")
```

With query terms reliably bound to graph vocabulary, the next chapter turns the bound triples into answers — single-hop and multi-hop — with provenance attached per fact (Chapter 10). The verification step that subsequently checks each answer against its supporting triples follows in Chapter 12.

Chapter 10

Answering through the GMS

Binding (Chapter 9) turned a question's words into the graph's own vocabulary: `topline` became `has_revenue` and a segment nickname became its canonical entity. What we hold after binding is a list of *bound query triples* — patterns over real graph terms with the asked value left as the bare variable `?`. This chapter performs the last step before language re-enters the loop: it turns those bound triples into *facts*.

A vector RAG *retrieves passages*; a GEODE-RAG system *answers through the graph*. The retriever resolves each bound triple against the trained store, prefers an asserted edge over a predicted one, chains multi-hop questions through a variable environment and attaches a `file:line:char` provenance span to every fact it returns. No LLM runs here — retrieval is geometry alone. The trained store on which retrieval operates is the geometric memory introduced in Chapter 2 and populated from documents in Chapter 4. Synthesis (Chapter 11) and self-verification (Chapter 12) come after, and they read only the facts this stage produces.

10.1 Bound triples become facts

The retriever is `knowlytix.knowledge.rag.retrieve.Retriever`. It takes the store and (optionally) a `ProvenanceLedger`, and exposes one method, `retrieve(bound)`, returning a `RetrieveResult` of `RetrievedFact` records and an `answers` list. Each fact carries its `head`, `relation`, `tail`, a geodesic `score` (0.0 for an asserted edge), a `confidence` in $[0, 1]$, a `source` tag (`"triple"`, `"link_predict"` or `"score"`) and — the part chunk-and-pray (Chapter 1) cannot provide — a `location` and the `raw` source span.

We reload the store trained in stage F2. This chapter runs no GPU work beyond this reload; the retrieval below is geometry alone.

```
# Reload the trained store built by scripts/build_store.py (F2). This is the only
# GPU/Qwen-touching part of the chapter at build time -- the lead runs it in CI;
# retrieval itself is pure geometry and needs no LLM.
import os
import torch

from knowlytix.knowledge.config import DocGMSConfig
from knowlytix.knowledge.store import GMSExpertStore
from knowlytix.knowledge.geode.provenance import ProvenanceLedger

REPO_ROOT = os.path.join(os.path.dirname(os.getcwd()), "code") if os.path.basename(os.
    getcwd()) == "notebooks" else os.getcwd()
```

```
STORE = os.environ.get("GMS_STORE",
                       os.path.join(REPO_ROOT, "data", "gms_annual_report_store"))

store = GMSExpertStore(DocGMSConfig(store_path=STORE),
                       device=torch.device("cpu"))
assert store.load(), f"no trained store at {STORE}; run scripts/build_store.py first"
ledger = ProvenanceLedger.from_text(store.markdown)
print("entities:", store.adapter.num_entities, " relations:", store.adapter.
    num_relations)
```

10.2 Single-hop retrieval with provenance

A factual question — "What is the Cloud Platform segment's revenue?" — binds to one triple, `(cloud platform, has_revenue, ?)`. The retriever sees a known head and relation with an unknown tail, so it runs a tail query: it checks `store.query_triples` first, finds the asserted edge and returns it as a hard fact at score 0.0. The number is the byte-exact ENM figure, and the ledger (Chapter 3) resolves the edge to the table cell it came from.

```
# Single-hop: bind a question's terms, then answer it *through the graph*.
# The asked value is the bare "?" slot; the retriever returns the asserted edge,
# never a parsed number, and attaches the source span.
from knowlytix.knowledge.rag import Retriever, TripleBinder
from knowlytix.knowledge.rag.query_triples import QueryTriple

binder = TripleBinder(store)
retriever = Retriever(store, ledger)

bt = binder.bind(QueryTriple("cloud platform", "has_revenue", "?"))
assert bt.bound  # every non-variable slot resolved to real graph vocabulary

result = retriever.retrieve([bt])
for f in result.facts:
    print(f"{f.head} {f.relation} {f.tail}  "
          f"[{f.source} conf={f.confidence:.2f}]  @ {f.location}")
    print("   raw:", f.raw)
print("answers:", result.answers)
```

```
cloud platform has_revenue 120.0  [triple conf=1.00]  @ <report>:15:553-558
   raw: <the table cell text the triple resolved to>
answers: [('120.0', 1.0)]
```

The source is `"triple"` — an asserted edge, not a guess — so the score is 0.0 and confidence 1.0. The value `120.0` is the figure ENM holds byte-exact (`segment_performance / Cloud Platform/Technology/Revenue = 120.0` in the corpus facts), carried with a span, not scraped from prose. A vector RAG, by contrast, would hand the synthesizer a *chunk* and rely on the model to read the correct cell.

GMS in practice — answers carry their evidence

The chunk-and-pray default returns the top-k passages and trusts the LLM to locate the number. The geometric counterpart returns a `RetrievedFact` whose `location` is `file:line:char` and whose `raw` is the exact span. Provenance is not added afterward; it is a field on every fact the retriever emits. Synthesis (Chapter 11) then reads *spans*, and verification (Chapter 12) checks claims against *facts* — both downstream of this guarantee.

10.3 Multi-hop through a variable environment

The annual report's hierarchy — segment → division → region — is exactly what a single-edge lookup cannot answer. "Which region hosts the highest-revenue segment?" needs two hops: find the leading segment's division, then that division's region. Cloud Platform leads revenue (120.0), its division is `technology`, and `technology` sits in `north america`.

The retriever resolves this through a *variable environment*. The first triple binds `?x` to the division; the second triple reads `?x` from the environment and asks for the region in the bare `?` slot. The chain is resolved greedily: a variable bound by one hop feeds the next.

```
# Multi-hop: "which region hosts the highest-revenue segment?"
# Cloud Platform leads revenue (120.0). We resolve its division (?x), then the
# region of that division (?). The intermediate variable ?x links the two hops:
# the retriever resolves it from hop 1 and feeds it into hop 2 (variable env).
chain = [
    binder.bind(QueryTriple("cloud platform", "has_division", "?x")),
    binder.bind(QueryTriple("?x", "has_region", "?")),
]
assert all(b.bound for b in chain)

res = retriever.retrieve(chain)
print("hops:")
for f in res.facts:
    print(f"  {f.head} {f.relation} {f.tail}  [{f.source}]  @ {f.location}")
print("answer:", res.answers)
```

```
hops:
  cloud platform has_division technology  [triple]  @ <report>:..
  technology has_region north america     [triple]  @ <report>:..
answer: [('north america', 1.0)]
```

Both hops are recorded as separate, provenance-bearing facts. That list *is* the audit trail: a reader sees why the answer is `north america` — segment, then division, then region — and can check each edge against its source span. The chain matches the cohort label for this question, and no edge was predicted.

10.4 Asserted edges, not predictions

The retriever's tail query checks `query_triples` before it ever calls `link_predict`. An edge the graph asserts is returned as a hard fact at score 0.0; `link_predict` is the fallback for genuinely

missing edges and is scored lower (and flagged `source="link_predict"`). This ordering is the difference between answering and guessing.

```
# Asserted edges are preferred over link_predict (Anti-pattern: guessing over a
# fact you hold). For an edge the graph asserts, source == "triple", score 0.0;
# link_predict only fills genuinely missing edges and is scored lower.
asserted = store.query_triples(head="cloud platform", relation="has_revenue")
print("asserted in graph:", asserted)

# A missing edge falls back to link_predict (a ranked guess, lower confidence):
guess = store.link_predict("cloud platform", "has_region", top_k=3)
print("link_predict (no asserted edge):", guess)
```

```
asserted in graph: [('cloud platform', 'has_revenue', '120.0')]
link_predict (no asserted edge): [(<entity>, <distance>), ...]
```

The segment-to-region link is not asserted directly — region lives on the *division*, not the segment — which is precisely why the region question must be answered multi-hop rather than with a single-edge `link_predict` guess. The chain follows the asserted edges that are held and does not predict the one that is absent.

10.5 Limits

This stage answers only what the graph asserts or can chain, subject to three boundaries. First, the retriever resolves the *first* binding for an intermediate variable (`env[tail_var] = asserted[0][2]`); a fan-out where `?x` legitimately has several values is not enumerated — the chain follows one path. Second, the both-unknown pattern (a triple with two variable slots) is unsupported and silently skipped; questions must be phrased so each hop fixes one end. Third, retrieval does not *rank* the answer set by world-knowledge plausibility — it returns what the graph holds. "Highest revenue" became answerable only because the chain was written starting from the segment already known to lead revenue; selecting that segment is a separate query (an ENM `argmax` over `has_revenue`), not something the retriever infers. When the bound triples do not resolve, the result is empty, and that emptiness drives the abstention calibrated in Chapter 13 — the retriever never invents a fact to fill the gap.

Anti-patterns

- **Predicting an edge already held.** Calling `link_predict` (or any similarity ranker) before checking `query_triples` returns a plausible guess where an asserted fact exists. The retriever's tail query checks the graph first *by design*; this ordering must not be reversed.
- **Dropping provenance between retrieval and synthesis.** A fact without its `location`/`raw` is unverifiable downstream. Retrieval results constructed by hand must never omit the span — it is what Chapter 12 checks claims against.
- **Forcing a one-hop answer onto a multi-hop question.** Asking `(segment, has_region, ?)` when region lives on the division yields a `link_predict` guess, not a fact. Chain the asserted edges instead.

Exercise — A three-hop chain over the hierarchy

Extend the two-hop pattern to three hops. Starting from the `logistics` segment, resolve its division into `?x`, that division's region into `?y`, and finally ask for the head of that division in the `?` slot. Confirm the answer is `sam reyes` and that all three hops come back as asserted facts (`source == "triple"`). The worked solution is in the chapter notebook; the expected facts are `logistics has_division operations`, `operations has_region europe` and `operations has_head sam reyes`.

10.6 What this gives the rest of the pipeline

After this chapter, a question is a set of `RetrievedFacts`, each an asserted edge (or an explicitly flagged prediction) with a confidence and a source span. Chapter 3 explained how those spans are resolved; Chapter 11 hands the facts and spans — not the document — to a local Qwen and requires it to answer *only* from them; and Chapter 12 decomposes the synthesized answer back into claim triples and checks each against the facts this stage produced. The downstream coverage and abstention behavior is calibrated in Chapter 13 and Chapter 14, and the end-to-end results are reported in Chapter 15. Every downstream stage depends on this stage having returned facts rather than passages.

Part VI

Generation and Trust

Chapter 11

Grounded synthesis

Retrieval (Chapter 10) already produced the answer *values* through the graph: bound query triples resolved to asserted facts, each carrying its source span. What remains is to turn those facts into prose a reader can use. This is the step at which a conventional RAG pipeline departs from its premise: it hands the model a body of retrieved text and relies on it to write the answer, so the model is free to fill any gap from parametric memory. The chunk-and-pray default of Chapter 1 embodies exactly this risk. The GEODE-RAG synthesizer follows the opposite course: it sees *only* the GMS-verified facts and their exact source excerpts, under a system prompt that forbids reaching beyond them. The values were decided by the geometry; synthesis is constrained to phrase them.

11.1 The evidence block

The `Assembler` (`knowlytix.knowledge.rag.assemble`) is deliberately thin. It formats the retrieved facts into an `<evidence>` block — one `FACT` line per triple plus the verbatim `SOURCE` span and its `file:line:char` location — and calls the synthesis LLM with a fixed system prompt. The prompt instructs the model to answer using *only* the supplied facts and excerpts, to cite values exactly and to decline when the evidence does not contain the answer.

Every notebook begins with the library bootstrap so imports resolve to the working branch:

```
import os, sys
KNOWLYTIX_SRC = os.environ.get("KNOWLYTIX_SRC", "/home/asudjianto/jupyterlab/GMS-
    knowlytix")
sys.path.insert(0, KNOWLYTIX_SRC)
```

```
from knowlytix.knowledge.rag.assemble import Assembler
from knowlytix.knowledge.rag.retrieve import RetrievedFact
from knowlytix.knowledge.rag.config import RagConfig
from knowlytix.knowledge.llm_backend import LLMBackend
from knowlytix.knowledge.geode import QWEN_3B

print('synthesis model (real path):', QWEN_3B)
```

We start from the facts that the `Retriever` of Chapter 10 returns for "cloud platform revenue" over the Northwind FY2025 store: the asserted triple `(cloud platform, has_revenue, 120.0)` with its provenance span. The exact figure, 120.0, is the value stored in the Exact Numerical Memory under `segment_performance / Cloud Platform/Technology/Revenue`.

```
# Facts as Ch8's Retriever would return them, with provenance spans.
```

```
# Source: data/corpus_facts.md sample retrieval for "cloud platform revenue".
facts = [
    RetrievedFact(
        head="cloud platform", relation="has_revenue", tail="120.0",
        score=0.0, confidence=1.0, source="triple",
        location=":15:553-558", raw="Cloud Platform | Technology | 120.0 | 340",
    ),
]
ENM_VALUE = "120.0"   # segment_performance / Cloud Platform/Technology/Revenue
print(Assembler._context("What was Cloud Platform revenue?", facts))
```

The printed context is what the synthesizer actually receives:

```
<evidence>
FACT: cloud platform | has_revenue | 120.0
SOURCE (:15:553-558): Cloud Platform | Technology | 120.0 | 340
</evidence>

Question: What was Cloud Platform revenue?
```

There is no document body here, no neighboring paragraph, no top-k chunk: only the verified fact and the cell it came from. The model cannot drift onto a plausible-sounding figure because no other figure is present.

11.2 Synthesizing and refusing

For deterministic CI we drive synthesis with a scripted backend that grounds in the evidence string; the local-Qwen path follows. Any object implementing the `LLMBackend` interface (one `call` method, one `model_name` property) can be substituted in its place.

```
class FakeBackend(LLMBackend):
    """Deterministic stand-in for Qwen: grounds in the evidence string.

    Mirrors a well-behaved synthesizer: it answers from the FACT line when
    the asked value is present, otherwise it declines. Used so the chapter's
    claim is checkable in CI without a GPU.
    """
    def __init__(self, model="fake-grounded"):
        self._model = model

    def call(self, system: str, user: str, max_tokens: int = 2048) -> str:
        if "120.0" in user:
            return "Cloud Platform reported revenue of 120.0."
        return "I cannot answer that from the available evidence."

    @property
    def model_name(self) -> str:
        return self._model
```

```
assembler = Assembler(FakeBackend())
grounded = assembler.assemble("What was Cloud Platform revenue?", facts)
print(grounded)
```

The grounded answer carries the ENM figure and nothing else:

```
Cloud Platform reported revenue of 120.0.
```

The production synthesis path is identical apart from the backend: local Qwen2.5-3B-Instruct on GPU, with no API key. This cell is tagged for the lead to run in CI, as it loads the model:

```python
# [GPU/Qwen  CI] Real synthesis path. The lead runs this in CI; it is the
# per-role LLM choice (local Qwen2.5-3B-Instruct, no API key).
from knowlytix.knowledge.llm_backend import LocalTransformersBackend

qwen = LocalTransformersBackend(QWEN_3B)
real_answer = Assembler(qwen).assemble("What was Cloud Platform revenue?", facts)
print(real_answer)
# Expected: prose stating revenue of 120.0, no other figure.
```

The other half of grounded synthesis is refusal. When retrieval returns no facts — as it does for an Outlook question, a prose section that carries no triples and is a measured coverage blind spot (Chapter 13) — the evidence block is empty and the synthesizer declines rather than invents a forecast:

```python
# No fact answers an Outlook question (a coverage blind spot, Ch11): the
# evidence block is empty, so a grounded synthesizer must decline.
no_facts: list[RetrievedFact] = []
refusal = Assembler(FakeBackend()).assemble(
    "What does the Outlook section forecast for FY2026?", no_facts)
print(refusal)
```

```
I cannot answer that from the available evidence.
```

Note. *Refusal here is the* synthesizer *declining on empty evidence. The pipeline as a whole reaches the same outcome earlier and more cheaply: with nothing bound,* `RagPipeline` *abstains before synthesis ever runs. The end-to-end abstention story — bind-check, accept/abstain/escalate, the audit trail — is Chapter 13.*

GMS in practice — evidence block vs. the whole document

The chunk-and-pray default (Chapter 1) fills the prompt with the top-k retrieved passages and relies on the model to select the correct number from the prose. GEODE-RAG hands the synthesizer a closed set of GMS-verified facts with their exact source cells. The model's task narrows from *find and judge the answer* to *phrase the answer that the geometry already decided*, which is why the synthesized text can be required to contain the ENM figure and no other number.

11.3 Per-role LLMs

`RagConfig` keeps the three LLM roles independent: `llm` for synthesis, `llm_extract` for the NL-to-query-triples step (Chapter 9) and `llm_verify` for answer self-verification (Chapter 12). Unset roles fall back: `verify` defaults to `extract`, and `extract` defaults to the synthesis `llm`. Swapping the synthesizer is a single-field change and leaves the other roles unchanged.

```
extract_be = FakeBackend("fake-extract")
synth_be = FakeBackend("fake-synth")

cfg = RagConfig(llm=synth_be, llm_extract=extract_be)
# Synthesis uses the swapped backend; verify defaults to the extract role.
assert cfg.llm.model_name == "fake-synth"
assert cfg.extract_llm().model_name == "fake-extract"
assert cfg.verify_llm().model_name == "fake-extract"   # verify -> extract
print("synthesis :", cfg.llm.model_name)
print("extract   :", cfg.extract_llm().model_name)
print("verify    :", cfg.verify_llm().model_name)
```

This is the deployment knob of Chapter 16: a small local model for extraction and verification, a larger one for synthesis or the reverse — each role chosen on its own merits.

11.4 Self-check

The chapter's claim is that a grounded answer contains the ENM figure and no other number, and that the synthesizer refuses when the evidence lacks the answer. The notebook's final cell asserts exactly this:

```
import re

# (a) the grounded answer carries the exact ENM figure ...
assert ENM_VALUE in grounded
# ... and introduces no other number.
nums = re.findall(r"\d+(?:\.\d+)?", grounded)
assert set(nums) == {ENM_VALUE}, nums
# (b) with no supporting facts, the synthesizer declines.
assert "cannot answer" in refusal.lower()
print("Ch9 self-check passed: grounded synthesis carries only the ENM figure;"
      " empty evidence -> refusal.")
```

Anti-patterns

- **Handing the model the whole document.** Stuffing the prompt with retrieved passages re-opens every door triple-mediation closed: the model can pick a wrong nearby number, average two cells or quote stale prose. Synthesis sees a closed evidence block, not the corpus.
- **Filling gaps from parametric memory.** An LLM will gladly produce a confident FY2026 outlook it was never given. Grounded synthesis forbids it; empty evidence yields a refusal, not a guess.
- **Re-deriving numbers in prose.** Do not let the synthesizer recompute or round a figure. The ENM value is authoritative and byte-exact (Chapter 2); the answer cites it verbatim.

Exercise — Per-role synthesis LLM

Using `RagConfig`, swap *only* the synthesis backend (`llm`) and confirm the extract and verify roles fall back correctly: `extract_llm()` returns the extract backend (or `llm` if unset), and `verify_llm()` returns the verify backend (or the extract role if unset). The worked solution is the per-role cell in the chapter notebook; extend it by also setting `llm_verify` and re-checking the chain.

Limits. Grounded synthesis constrains *where* the answer comes from; it does not *prove* that the prose faithfully restates the facts. A sufficiently wayward model could still mis-phrase a value even with the evidence in front of it, and the Assembler does not parse its own output. That check is a separate stage: answer self-verification decomposes the generated text back into claim triples and tests them against the GMS (Chapter 12). Synthesis also inherits the coverage of retrieval: it can only ground what the graph holds, so a blind-spot question yields a refusal here and an abstention upstream (Chapter 13), never a dense guess unless the distrusted fallback is explicitly enabled (Chapter 16). For the geometric internals behind the ENM and the verified facts, see Chapter 2 and Appendix C; for the agent that consumes this answer as a tool, see the capstone bridge (Appendix A) and the `search_policy` of the companion agent book.

Chapter 12

Self-verification: the GMS as a hallucination detector

Chapter 11 produced a *grounded* draft: the synthesizer was handed retrieved facts plus their spans and instructed to answer only from them. That constrains the model; it does not *prove* that the model obeyed. A small local model can still drop a digit, swap a segment or paste a half-remembered figure from pre-training. Constraining generation reduces the hallucination rate; it does not drive it to zero, and a single confident wrong number is precisely the failure a trustworthy financial-RAG system cannot ship.

We decompose the *answer itself* back into claim triples and check each claim against the trained store with `AnswerVerifier`. The check is **calibration-free**: a numeric-aware *contradiction* test (the answer says t, the graph asserts $t' \neq t$) and, for cap-trained stores, an *admissibility* test against the relation's learned cap radius ρ_r. No threshold is set by hand, and the verifier is not an LLM judging an LLM — the adjudicator is the geometry. The LLM is used only to *decompose* the draft into triples; the verdict comes from the store. This continues the triple-mediated discipline introduced in Chapter 8 and the provenance ledger of Chapter 3, now applied to the generated answer rather than the retrieved evidence.

We run two drafts over Northwind Industries FY2025: a faithful one (every claim *supported*) and a tampered one with a hallucinated total-revenue figure (*contradicted*). With `on_verify_fail="abstain"` the pipeline refuses the tampered answer rather than emitting it.

12.1 A deterministic claim-extraction backend

`AnswerVerifier` calls an LLM only to *decompose* the draft answer into claim triples; the adjudication that follows is pure geometry. For a deterministic, GPU-free demonstration we therefore inject a scripted `LLMBackend` that returns the claim triples a competent extractor would produce for each draft. The real Qwen path is shown at the end of the chapter and is identical except for this stub.

```
import json
from knowlytix.knowledge.llm_backend import LLMBackend


class ScriptedBackend(LLMBackend):
    """Deterministic stand-in for the verify-LLM.

    Maps each draft answer (matched by substring) to the JSON claim-triple
```

```
    array a real extractor would emit. No GPU, no network -- the geometry,
    not this stub, decides supported vs contradicted.
    """

    def __init__(self, script: dict[str, list[dict]]):
        self._script = script

    def call(self, system: str, user: str, max_tokens: int = 2048) -> str:
        for needle, claims in self._script.items():
            if needle in user:
                return json.dumps(claims)
        return '[]'

    @property
    def model_name(self) -> str:
        return "scripted-verify"
```

12.2 The facts we verify against

The verifier reads the *trained* store. Training a real `GMSExpertStore` needs the GPU (Chapter 4), so to keep this chapter offline and deterministic we back the verifier with a tiny *fixture store* that exposes exactly the methods `AnswerVerifier` and `TripleBinder` call — `query_triples`, `fuzzy_match_entity`, `cap_radius`, `score_triple`, and an `adapter` with the relation/entity vocabulary. We seed it with the Northwind triples the two drafts touch, taken verbatim from the corpus facts.

```
from dataclasses import dataclass, field


@dataclass
class _Adapter:
    relation_to_idx: dict
    entity_to_idx: dict


class FixtureStore:
    """CPU-only stand-in exposing the store surface the verifier/binder use."""

    def __init__(self, triples):
        self._triples = [tuple(t) for t in triples]
        ents = {h for h, _, _ in self._triples} | {t for _, _, t in self._triples}
        rels = {r for _, r, _ in self._triples}
        self.adapter = _Adapter(
            relation_to_idx={r: i for i, r in enumerate(sorted(rels))},
            entity_to_idx={e: i for i, e in enumerate(sorted(ents))},
        )

    def query_triples(self, head=None, relation=None, tail=None):
        return [t for t in self._triples
                if (head is None or t[0] == head)
                and (relation is None or t[1] == relation)
                and (tail is None or t[2] == tail)]
```

```
    def fuzzy_match_entity(self, name):
        return name if name in self.adapter.entity_to_idx else None

    def cap_radius(self, rel):
        return None            # fixture has no learned cap geometry

    def score_triple(self, head, rel, tail):
        return None            # not scorable without a trained model


triples = [
    ("total", "has_revenue", "355.0"),
    ("cloud platform", "has_revenue", "120.0"),
    ("logistics", "has_revenue", "95.0"),
]
store = FixtureStore(triples)

# Ground truth (corpus_facts.md): total revenue is exactly 355.0
print(store.query_triples(head="total", relation="has_revenue"))
```

```
[('total', 'has_revenue', '355.0')]
```

In a real deployment we would load the store built in Chapter 4 and pass it directly to `AnswerVerifier`; the verification logic is identical, only the store is real. The fixture is a contradiction-only stand-in: `cap_radius` returns `None`, so the admissibility branch reports `unverifiable` rather than fabricating a geometry the fixture does not have.

12.3 A faithful draft verifies as supported

The first draft states the correct figure. `AnswerVerifier.verify` decomposes it to the claim `(total, has_revenue, 355.0)`, binds it to the graph and finds the asserted edge, and numeric-aware equality holds — so the claim is `supported` and `report.ok` is `True`.

```
from knowlytix.knowledge.rag.verify import AnswerVerifier

good_draft = "Total revenue for FY2025 was 355.0 million."
good_script = {
    good_draft: [{"head": "total", "relation": "has_revenue", "tail": "355.0"}],
}
verifier = AnswerVerifier(store, ScriptedBackend(good_script))

report = verifier.verify(good_draft)
print("ok       :", report.ok)
print("verdicts :", [(v.triple.as_tuple(), v.status) for v in report.verdicts])
print("failures :", report.failures)
```

```
ok       : True
verdicts : [(('total', 'has_revenue', '355.0'), 'supported')]
```

```
failures : []
```

The claim matched an *asserted* edge, so neither calibration nor a cap radius was needed — the contradiction test alone settled it.

12.4 A hallucinated figure verifies as contradicted

Consider next a draft that reports `455.0` — a plausible-looking but wrong total, the kind of single-digit slip a 3B model makes. The graph asserts `355.0`; numeric-aware equality fails; the verdict is `contradicted` and `report.ok` is `False`. The `detail` names what the graph actually holds.

```
bad_draft = "Total revenue for FY2025 was 455.0 million."
bad_script = {
    bad_draft: [{"head": "total", "relation": "has_revenue", "tail": "455.0"}],
}
bad_verifier = AnswerVerifier(store, ScriptedBackend(bad_script))

bad_report = bad_verifier.verify(bad_draft)
print("ok      :", bad_report.ok)
for v in bad_report.verdicts:
    print(v.triple.as_tuple(), "->", v.status, "|", v.detail)
```

```
ok      : False
('total', 'has_revenue', '455.0') -> contradicted | graph asserts ['355.0']
```

The verifier did not *judge* whether 455 is reasonable — it found a directly asserted fact that disagrees. This is the difference between a faithfulness check and an opinion.

GMS in practice — geodesic contradiction vs. an LLM judge

The chunk-and-pray default for "does this answer hallucinate?" is a second LLM prompt — "rate this answer's faithfulness 1–5" — which is one stochastic model grading another, with no ground truth and no provenance. The approach introduced in Chapter 1 as the baseline is replaced here by a *lookup*: the draft's claim triple is compared against the store's asserted edge, exactly and numeric-aware. A wrong total is not "rated low"; it is `contradicted` by a fact that carries its own span. The adjudicator is the geometry, not a judge that can itself hallucinate.

12.5 The pipeline abstains on a failed verification

Verification is a pipeline stage, not a one-off call. With `verify_llm_output=True` and `on_verify_fail="abstain"`, a contradicted claim forces `RagPipeline` down the abstain branch: `decision == "abstain"`, the answer becomes the standard refusal, and the `verification` audit record carries `ok=False`. We point both the synthesis LLM and the verify LLM at scripted backends so the whole `query()` runs offline.

```
from knowlytix.knowledge.rag.config import RagConfig
from knowlytix.knowledge.rag.pipeline import RagPipeline

question = "What was total revenue in FY2025?"
```

```
# Extractor: question -> a query triple with the asked slot '?'.
extract_script = {
    question: [{"head": "total", "relation": "has_revenue", "tail": "?"}],
}
# Synthesizer: emits the tampered draft (455.0).
synth_script = {question: bad_draft}


class SynthBackend(LLMBackend):
    def __init__(self, text):
        self._text = text
    def call(self, system, user, max_tokens=2048):
        return self._text
    @property
    def model_name(self):
        return "scripted-synth"


cfg = RagConfig(
    llm=SynthBackend(bad_draft),
    llm_extract=ScriptedBackend(extract_script),
    llm_verify=ScriptedBackend(bad_script),
    verify_llm_output=True,
    on_verify_fail="abstain",
    relevance_gate=False,       # offline: skip the extra LLM relevance call
    ground_extraction=False,    # offline: scripted extractor needs no schema
)
pipe = RagPipeline.from_store(store, cfg)
ans = pipe.query(question)

print("decision     :", ans.decision)
print("answer       :", ans.answer)
print("verified ok? :", ans.verification.get("ok"))
print("notice       :", ans.notice)
```

```
decision     : abstain
answer       : I cannot answer this from the available grounded evidence.
verified ok? : False
notice       : Answer failed GMS self-verification (unsupported claims).
```

The tampered figure never reaches the end user. By contrast, a vanilla pipeline would return `455.0` with confident prose and no signal that it is wrong.

12.6 The real path: Qwen2.5-3B as the verify-LLM

In production the verify-LLM is the same local model that synthesizes — only its *role* differs (decompose, not generate). The listing below is the real Qwen wiring; it loads the model on the GPU and is executed in CI, not while authoring. The geometry-based adjudication is identical to the scripted runs above; only the claim extractor changes. The choice of Qwen2.5-3B as the local runtime model is discussed further in Appendix A.

```
# CI-ONLY: requires the shared GPU. Do not run while authoring.
from knowlytix.knowledge.geode.agent_llm import QWEN_3B
from knowlytix.knowledge.llm_backend import LocalTransformersBackend

qwen = LocalTransformersBackend(QWEN_3B)        # Qwen/Qwen2.5-3B-Instruct
qwen_verifier = AnswerVerifier(store, qwen)
qwen_report = qwen_verifier.verify(bad_draft)
assert qwen_report.ok is False                  # 455.0 contradicts 355.0
```

12.7 Limits

Self-verification catches what the *graph* can dispute, subject to three boundaries. First, it is only as good as claim decomposition: if the verify-LLM fails to extract a claim the draft actually makes, that claim is never checked — the verifier reduces, but does not eliminate, dependence on a small model. Second, a claim that touches nothing in the graph is reported `unverifiable`, not safe; for a store without learned caps (cap radius `None`) the admissibility test cannot fire, so the only hard signal is direct contradiction against an asserted edge. Third, this is a *faithfulness* check, not a *truth* check: it confirms that the answer agrees with the store, so a wrong fact in the store will be faithfully echoed. Verification presumes that the GEODE-corrected graph (Chapter 5) is itself sound; it is the last gate, not the first. The admissibility test against learned cap radii, which the fixture store cannot exercise, is developed in Chapter 14.

Anti-patterns

- **LLM-judging-an-LLM.** Asking a second model to "rate faithfulness" substitutes one stochastic opinion for another, with no ground truth and no provenance. Adjudicate against the graph, not against a judge that can itself hallucinate.
- **Verifying prose without claim alignment.** Comparing the draft's raw text to a retrieved passage by similarity does not detect a swapped number — the strings stay similar. Decompose to claim triples and check each tail against the asserted tail; that is what makes the check numeric-aware and exact.
- **Treating `unverifiable` as `supported`.** A claim that binds to nothing is not validated; it is unchecked. Surfacing it as advisory rather than as a pass keeps the abstain decision sound.

Exercise — Regenerate before abstaining

Set `on_verify_fail="regenerate"` and re-run the tampered query. With a fixed scripted synthesizer the second draft is identical, so verification fails again and the pipeline *annotates* (confidence driven to `0.0`, a notice naming `total.has_revenue=455.0`) rather than abstaining. Then wire a `SynthBackend` whose second `call` returns the correct `355.0` draft and confirm the regenerate path now accepts a `supported` answer. The worked solution is in the chapter notebook.

Self-verification is the gate that turns a grounded draft (Chapter 11) into a *trustworthy* one: a hallucinated figure is contradicted by the geometry and refused, not rated and shipped. The next

chapter (Chapter 13) generalizes the refusal — moving from "this answer is wrong" to "we do not cover this question at all" — and the evaluation chapter (Chapter 15) measures the resulting *zero confident-wrong* guarantee across the whole cohort. For the agent-side counterpart, see the draft-verifier discussion in Appendix A.

Chapter 13

Abstention and coverage

A trustworthy RAG system must know, and *say*, what it does not know. Triple-mediated retrieval (Chapter 8) provides a measurable notion of *do not know*: if a question does not bind to a known entity or relation, or no grounded fact matches, the pipeline **abstains** instead of guessing (Chapter 10). Because every answerable fact derives from a triple with provenance (Chapter 3), the regions of a document that carry **body text but no triples** are *measurable blind spots* rather than a silent failure mode.

This chapter demonstrates three things over the Northwind Industries FY2025 annual report:

1. a prose question (Management Discussion, Risk Factors, Outlook) **abstains with a notice**;
2. `coverage_report(store)` flags exactly the prose sections as blind spots;
3. `RagAnswer.audit()` emits a per-query audit trail.

The companion notebook is `notebooks/11_abstention_and_coverage.ipynb`. Its first cell is the standard `KNOWLYTIX_SRC` bootstrap; every listing below is copied verbatim from a notebook cell.

13.1 The coverage monitor

`coverage_report` walks the document section by section, buckets each content triple by the source line its provenance resolves to and reports which sections have body text but zero triples. These are the blind spots that a triple-mediated pipeline cannot reach.

Coverage is computable from the markdown plus the store's triple list alone, with no model forward pass, so this cell runs on CPU. The trained store ships its triple list as `triples.json`; we read it directly and wrap it in a small stand-in carrying `.markdown` and `.triples`, which is all that `coverage_report` reads. In production one passes the real `GMSExpertStore`.

```
import json
from dataclasses import dataclass, field
from knowlytix.knowledge.rag import coverage_report

# The trained store's triple list, read from disk (CPU-only, no model load).
with open(os.path.join(STORE_PATH, "triples.json")) as f:
    TRIPLES = [tuple(t) for t in json.load(f)]
print(f"{len(TRIPLES)} triples loaded from the store (incl. structural in_section)")

@dataclass
```

```
class _StoreView:
    """Minimal view coverage_report needs: .markdown and .triples."""
    markdown: str
    triples: list = field(default_factory=list)
    store_path: str = ""

store_view = _StoreView(markdown=markdown, triples=TRIPLES, store_path=STORE_PATH)
report = coverage_report(store_view)

print(f"coverage_ratio = {report.coverage_ratio:.2f}")
for r in report.regions:
    mark = "x" if r.covered else " "
    flag = "  <-- BLIND SPOT" if r.blind_spot else ""
    print(f"[{mark}] {r.title} ({r.triple_count} triples, "
          f"{r.body_lines} body lines){flag}")
print()
print("blind spots:", [r.title for r in report.blind_spots])
```

The output matches the canonical `data/corpus_facts.md`:

```
coverage_ratio = 0.56
[ ] Northwind Industries -- Annual Report FY2025 (0 triples, 4 body lines)  <-- BLIND
    SPOT
[x] 1. Segment Performance (15 triples, 9 body lines)
[x] 2. Divisions (4 triples, 5 body lines)
[x] 3. Income Statement (8 triples, 7 body lines)
[x] 4. Balance Sheet (3 triples, 6 body lines)
[x] 5. Corporate Facts (4 triples, 6 body lines)
[ ] 6. Management Discussion and Analysis (0 triples, 9 body lines)  <-- BLIND SPOT
[ ] 7. Risk Factors (0 triples, 8 body lines)  <-- BLIND SPOT
[ ] 8. Outlook (0 triples, 5 body lines)  <-- BLIND SPOT

blind spots: ['Northwind Industries -- Annual Report FY2025',
 '6. Management Discussion and Analysis', '7. Risk Factors', '8. Outlook']
```

Roughly half the document's content-bearing regions carry no triple. This is not a defect: those sections are qualitative prose (MD&A, Risk Factors, Outlook) for which the report itself defers to the tables for authoritative figures. The gap is **named and counted** rather than hidden.

13.2 A prose question abstains

We now turn to the online path. We load the trained store and run a Risk-Factor question through `RagPipeline`. The Risk Factors and Outlook sections have no triples, so nothing binds: the bind-check fires and the pipeline abstains with a notice rather than fabricating a paragraph.

CI-only cell

This cell loads the store and runs Qwen2.5-3B-Instruct on the GPU. The library is the authoritative source for every symbol used here.

```
# === CI-only (loads store + Qwen on GPU) ===
from knowlytix.knowledge.store import GMSExpertStore
```

```
from knowlytix.knowledge.config import DocGMSConfig
from knowlytix.knowledge.llm_backend import LocalTransformersBackend
from knowlytix.knowledge.rag import RagConfig, RagPipeline
from knowlytix.knowledge.geode import QWEN_3B

store = GMSExpertStore(DocGMSConfig(store_path=STORE_PATH))
assert store.load(), "store failed to load"

qwen = LocalTransformersBackend(QWEN_3B)
rag = RagConfig(llm=qwen)            # bank-grade defaults: dense off, abstain-not-
    guess
pipe = RagPipeline.from_store(store, rag)

prose_q = "What are the company's main risk factors?"
ans = pipe.query(prose_q)
print("decision:", ans.decision)
print("route:   ", ans.route)
print("notice:  ", ans.notice)
print("answer:  ", ans.answer)
assert ans.decision == "abstain"
assert ans.notice is not None
```

Expected: `decision: abstain`, `route: triple` and a notice such as *"Question did not bind to known entities/relations."* The answer is the standard refusal string, not a guess. Contrast a chunk-and-pray baseline (Chapter 1), which would summarize the Risk Factors prose and present it as an answer with no means of verifying it.

GMS in practice — abstain instead of summarizing prose

Chunk-and-pray default: retrieve the top-k prose chunks for the Risk Factors question and let the LLM summarize them; an answer is always produced and it is unverifiable.

Geometric counterpart: the question's terms fail the bind-check against the graph, so `RagPipeline` abstains with `decision="abstain"` and a notice. There is no grounded fact and no answer. The blind spot that caused the abstention is independently measurable via `coverage_report` (Section 13.1).

13.3 The per-query audit trail

Every answer, whether accepted or abstained, carries a structured audit record: the decision, the route, the query triples, what bound, the source facts and their provenance spans, the verification verdicts and the notice. The record is wired to `RagConfig.audit_sink`, which captures every query.

```
# === CI-only (uses 'ans' from the previous listing) ===
import json
audit = ans.audit()
print(json.dumps(audit, indent=2, default=str))
assert audit["decision"] == "abstain"
assert audit["route"] == "triple"
assert audit["verified"] is True   # abstaining IS a verified outcome
```

The audit shows `decision: abstain` with an empty `sources` list: there was no grounded fact to cite, which is why the system declined. Note `verified: true`: an abstention is a *verified* outcome,

not an error. We compare this trail to a dense-retrieval answer in Chapter 16, which carries `verified=False` and a quarantine notice.

Anti-patterns

- **Answering on a miss.** When nothing binds, returning the best-effort top-k summary is the central error of chunk-and-pray: it converts "do not know" into a confident, unverifiable paragraph. The geometric default abstains.
- **Hiding blind spots behind dense retrieval.** Silently routing every unbound query to a vector index makes the coverage gap invisible: the system always answers, so no one measures what it cannot ground. Dense retrieval should remain opt-in and flagged (Chapter 16) and the coverage report should remain visible.
- **Treating abstention as an error.** An abstain with `verified=True` is a correct, auditable outcome. Logging it as a failure, or alerting on it, trains operators to "fix" the system by making it guess.

Exercise — Clear a blind spot

Add a triple to a blind-spot section and observe coverage clear. The library enforces a subtlety: `coverage_report` counts a triple toward a section only if its *provenance resolves* into that section. The `ProvenanceLedger` (Chapter 3) resolves table cells, section headers (`in_section`) and schema bullets; a triple invented out of thin air resolves to line `-1` (`unaligned`) and counts toward nothing. Show that an unanchored content triple does not clear the `8. Outlook` blind spot, but that an `in_section` triple keyed to the real Outlook header does (pass `exclude_relations=()` so the structural triple is counted).

Worked solution (in the notebook):

```
from knowlytix.knowledge.geode.provenance import ProvenanceLedger
ledger = ProvenanceLedger.from_text(markdown)

# An invented content triple does NOT resolve -> it cannot clear a blind spot.
fake = ("cloud platform", "has_outlook", "continued investment")
print("invented triple resolves to line:", ledger.resolve(*fake).line_no,
      "(unaligned)")

# An anchored triple keyed to the real Outlook header DOES resolve.
outlook_triple = ("outlook", "in_section", "outlook")
print("anchored triple resolves to line:", ledger.resolve(*outlook_triple).line_no)

augmented = _StoreView(markdown=markdown,
                       triples=TRIPLES + [outlook_triple],
                       store_path=STORE_PATH)
# Count structural relations too, so the anchored triple registers.
after = coverage_report(augmented, exclude_relations=())
titles = [r.title for r in after.blind_spots]
print("blind spots after:", titles)
print(f"coverage_ratio: {report.coverage_ratio:.2f} -> {after.coverage_ratio:.2f}")
assert "8. Outlook" not in titles
```

`8. Outlook` drops out of `blind_spots` and the coverage ratio rises (0.56 → 0.67). The lesson is twofold. First, the remedy for a blind spot is *more anchored triples*, obtained by re-ingesting the section or broadening extraction, rather than enabling distrusted dense retrieval (the opt-in escape hatch in Chapter 16). Second, the coverage monitor cannot be gamed: an unanchored claim resolves to `-1` and clears nothing, so the ratio moves only when real evidence backs the new triple.

13.4 Limitations

The coverage monitor measures *triple presence per section*, not *answer quality*. A section with triples can still fail to answer a specific question, because the triple exists but the asked attribute differs; that is the relevance gate's task (Chapter 12). A covered section's triples can also be wrong if extraction erred, which is the concern of GEODE (Chapter 5). Coverage says nothing about whether a blind spot *should* have been triplified: qualitative prose such as Risk Factors legitimately carries no authoritative fact, so a 0.56 ratio is not a defect to drive to 1.0. Finally, abstention here is binary; the pipeline does not rank *how close* it came to binding, so a near-miss paraphrase and a wholly off-topic question both abstain. Calibrating the bind threshold so that paraphrases resolve while off-topic queries still abstain is the subject of Chapter 14.

13.5 Self-check

The chapter's claim is that the prose sections are measurable blind spots and that the coverage monitor names them. The notebook's final cell establishes this CPU-only against the real library (no store, no Qwen):

```
blind = {r.title for r in report.blind_spots}
assert "7. Risk Factors" in blind
assert "8. Outlook" in blind
assert "6. Management Discussion and Analysis" in blind
# The fact-bearing tables are NOT blind spots.
assert "1. Segment Performance" not in blind
assert "3. Income Statement" not in blind
assert abs(report.coverage_ratio - 0.56) < 0.01
print("OK: prose sections are measurable blind spots; tables are covered.")
```

Part VII

Calibration and Evaluation

Chapter 14

Calibrate, Don't Guess

A threshold picked by eye is a number that cannot be defended in a review. The embedding binder of Chapter 9 turns on a single similarity cut: above it a paraphrase is accepted as a graph entity, below it the term is refused and the query abstains (Chapter 13). Choosing that cut by inspecting a few examples is the same "chunk and pray" instinct that Chapter 1 argues against — it produces a system whose behavior nobody can reconstruct.

This chapter sets the cut *from data*. We label a small cohort: paraphrases that *should* resolve to a Northwind segment are positives and unrelated terms are negatives. `calibrate_bind_threshold` then sweeps a grid and writes back the threshold that best separates the two. The same discipline — cohort, sweep and operating point under a cost ceiling — governs the accept/abstain cuts downstream and is the shared method of Appendix C; we apply it here to one knob rather than re-derive it.

14.1 A deterministic, corpus-grounded encoder

The production binder downloads a MiniLM encoder on first use, which requires the GPU and a network round-trip. For a calibration sweep that runs in CI we *inject* a deterministic encoder. It maps each Northwind revenue segment and its paraphrases onto the same axis; the four axes are the four segments from the corpus (`cloud platform`, `devices`, `logistics` and `retail`). Unrelated terms land at the origin and bind to nothing. The *procedure* below is identical with a real encoder — only the vectors change.

```
import torch

# Axes: [cloud_platform, devices, logistics, retail].
# Each row activates the axis whose segment the term denotes (canonical name or
    paraphrase).
_AXES = {
    0: ("cloud platform", "saas", "cloud"),       # cloud platform
    1: ("devices", "hardware", "gadget"),         # devices
    2: ("logistics", "shipping", "freight"),      # logistics
    3: ("retail", "stores", "storefront"),        # retail
}


def fake_encode(texts: list[str]) -> torch.Tensor:
    """4-dim semantic encoder over Northwind's revenue segments (no network)."""
    rows = []
```

```
    for t in texts:
        tl = t.lower()
        rows.append([1.0 if any(k in tl for k in keys) else 0.0
                     for _, keys in sorted(_AXES.items())])
    return torch.tensor(rows)


# Sanity: a paraphrase and its segment share an axis; an unrelated term is zero.
print("saas    ->", fake_encode(["saas"]).tolist()[0])
print("cloud platform ->", fake_encode(["cloud platform"]).tolist()[0])
print("weather ->", fake_encode(["weather"]).tolist()[0])
```

The paraphrase `saas` and the canonical `cloud platform` both produce `[1, 0, 0, 0]`; `weather` produces `[0, 0, 0, 0]`. The cosine between a paraphrase and its segment is therefore 1.0, and between noise and any segment is 0.0 — a clean separation the sweep will find.

14.2 An embedding-mode binder over the trained store

`calibrate_bind_threshold` tunes a `TripleBinder` in *embedding* mode. Exact and fuzzy string matches still take precedence; the embedding fallback fires only for terms fuzzy cannot resolve (Chapter 9). We load the store that ships under `data/gms_annual_report_store/` read-only.

```
import torch
from knowlytix.knowledge.config import DocGMSConfig
from knowlytix.knowledge.store import GMSExpertStore
from knowlytix.knowledge.rag import TripleBinder

STORE = os.environ.get("GMS_STORE", "data/gms_annual_report_store")
store = GMSExpertStore(DocGMSConfig(store_path=STORE), device=torch.device("cpu"))
assert store.load(), f"no trained store at {STORE}; run scripts/build_store.py first"

binder = TripleBinder(store, mode="embedding", encoder=fake_encode,
                      bind_threshold=0.5, bind_margin=0.05)
print("entities:", sorted(store.adapter.entity_to_idx)[:8], "...")
print("starting bind_threshold:", binder.bind_threshold)
```

14.3 Positives, negatives and the sweep

Positives are `(term, expected_entity)` pairs that *should* bind to a segment; negatives are terms that should bind to *nothing*. The grid sweep maximizes (true positives + true negatives)/total and writes the winning threshold back onto the binder.

```
from knowlytix.knowledge.rag import calibrate_bind_threshold
from knowlytix.knowledge.rag.query_triples import QueryTriple

# Paraphrases that fuzzy string match misses -> must resolve via embeddings.
positives = [
    ("saas",     "cloud platform"),
    ("hardware", "devices"),
    ("shipping", "logistics"),
    ("stores",   "retail"),
```

```
]
# Unrelated terms -> must NOT bind to any segment.
negatives = ["weather", "moon", "guitar"]

best_th, acc = calibrate_bind_threshold(binder, positives, negatives)
print(f"chosen bind_threshold = {best_th:.3f}")
print(f"separation accuracy   = {acc:.3f}")
print("binder now set to     =", binder.bind_threshold)
```

The chosen threshold is the *operating point*: above it the binder accepts a paraphrase as a segment match, below it the term is refused. On this cohort the cosine gap is binary (1.0 for a paraphrase, 0.0 for noise), so any threshold in $(0, 1]$ achieves `acc == 1.0`; the sweep returns the smallest grid point, 0.05. The binder mutates in place — subsequent `bind` calls use the calibrated value.

GMS in practice — a defensible cut instead of a guessed one

The chunk-and-pray default is a magic number: `if similarity > 0.8`, set by whoever wrote the line. The geometric counterpart is the same line of code *with provenance* — the threshold is the output of a recorded sweep over a labeled cohort, so the question "why 0.8?" has an answer ("it maximized separation on cohort X"), and re-calibration is a re-run rather than a debate.

14.4 The calibrated binder resolves paraphrases and refuses noise

```
def bind_head(term: str) -> str | None:
    return binder.bind(QueryTriple(term, "has_revenue", "?")).head


for term in ["saas", "shipping", "stores"]:
    print(f"{term:10s} -> {bind_head(term)!r}")     # paraphrase -> canonical segment
for term in ["weather", "moon"]:
    print(f"{term:10s} -> {bind_head(term)!r}")     # noise -> None (refused)
```

After calibration `saas` binds to `cloud platform`, `shipping` to `logistics` and `stores` to `retail`; both `weather` and `moon` bind to `None` and feed the bind-check abstention of Chapter 13.

14.5 Limits

Calibration is only as good as the cohort. With four segments and three noise terms the grid finds a clean cut, but a threshold tuned on this cohort does *not* generalize to a relation band it never saw — per-relation bands need per-relation labels, and a single global cut will be too loose for some relations and too tight for others. The deterministic encoder here makes the axes orthogonal by construction; the real MiniLM space is less separable, so we expect a softer separation, a non-trivial `bind_margin` and a threshold strictly inside $(0, 1)$ rather than at the grid edge. The principal limitation is that calibration sets *where* to draw the line — it cannot manufacture signal the encoder does not carry. If a paraphrase is closer to the wrong segment in embedding space, no threshold recovers it; that is a limitation of the encoder (Chapter 9), not of the sweep.

Anti-patterns

- **Thresholds by eye.** Hand-picking a cut from a handful of examples produces a number nobody can defend or reproduce. Sweep a labeled cohort and record the operating point.
- **Calibrating on the system's own output.** Building positives and negatives from answers the pipeline already accepted bakes its current bias into the threshold and reports inflated separation. The cohort should be labeled independently of the system under test.
- **One global cut for every relation.** A threshold tuned on segment names is not valid for, say, region or division relations. Calibrate per-relation bands when the cost of a mis-bind differs across relations.

Exercise — tighten the false-allow ceiling

The default sweep maximizes raw accuracy, treating a false-allow (binding noise) the same as a false-refuse. A bank-grade system pays more for a false-allow. Re-rank the grid under a *hard ceiling of zero false-allows* and report how the chosen threshold moves. The worked solution is in the notebook; the listing below is its core. Carrying these calibrated cuts into a measurable accuracy and coverage report is the subject of Chapter 15.

```
def calibrate_under_ceiling(binder, positives, negatives, *,
                            max_false_allow=0,
                            grid=tuple(i / 20 for i in range(1, 20))):
    """Pick the lowest threshold (max recall) whose false-allows <= ceiling."""
    feasible = []
    for th in grid:
        binder.bind_threshold = th
        false_allow = sum(binder.bind(QueryTriple(t, "_", "?")).head is not None
                          for t in negatives)
        if false_allow <= max_false_allow:
            tp = sum(binder.bind(QueryTriple(t, "_", "?")).head == exp
                     for t, exp in positives)
            feasible.append((th, tp))
    if not feasible:
        raise ValueError("no threshold meets the false-allow ceiling")
    # lowest threshold that still admits the most positives
    best_th = max(feasible, key=lambda x: (x[1], -x[0]))[0]
    binder.bind_threshold = best_th
    return best_th
```

Self-check

The notebook's final cell proves the chapter's claim: the calibrated threshold separates the labeled cohort, the binder is updated in place, and after calibration a paraphrase binds to its canonical segment while noise is refused.

```
# Re-run the data-accuracy calibration so the assert is self-contained.
```

```
binder.bind_threshold = 0.5
best_th, acc = calibrate_bind_threshold(binder, positives, negatives)

# Claim of the chapter: the calibrated threshold separates pos from neg.
assert acc == 1.0, f"calibration failed to separate cohort: acc={acc}"
assert 0.0 < best_th <= 1.0, f"threshold out of range: {best_th}"
assert binder.bind_threshold == best_th, "binder not updated in place"

assert binder.bind(QueryTriple("saas", "has_revenue", "?")).head == "cloud platform"
assert binder.bind(QueryTriple("weather", "has_revenue", "?")).head is None

print("OK: calibrated bind_threshold separates the labeled cohort.")
```

Chapter 15

Evaluating the RAG with a Designed Experiment

A single accuracy number is the wrong instrument for a grounded system. It rewards a confident guess as much as a verified answer, says nothing about whether the retriever ranked the right fact, and cannot tell *which* kinds of question break the system. This chapter evaluates the assembled RAG with the designed experiment built in Chapter 6: the DoE cohort is the test set, the GMS is the oracle, and four metrics — retrieval precision@k and recall@k, correctness and completeness — are read off every answer. A logistic analysis then attributes the failures to the presentation factors that caused them, and the whole battery is run a second time against the traditional chunk-and-pray baseline of Appendix E so the two architectures can be compared on identical questions.

15.1 The cohort is the design, the GMS is the oracle

Evaluation reuses `data/enrichment/rag_cohort.json` — the 150 contextual questions designed in Chapter 6, each carrying a graph-derived answer. Because the answers come from the store, no human labeling is involved and every score is checked against ground truth the GMS itself holds. The five presentation factors travel with each case, so any metric can be sliced by factor.

```
import json
cohort = json.load(open("data/enrichment/rag_cohort.json"))
print(len(cohort), "designed cases; factors:", list(cohort[0]["_factors"]))
```

```
150 designed cases; factors: ['clarity', 'style', 'length', 'expertise',
'paraphrase_depth']
```

15.2 Four metrics, all against the GMS

Each question is run through the pipeline and scored four ways, all with library functionality and the GMS as the oracle. *Precision@k* and *recall@k* ask whether the ground-truth value is among the top-k retrieved units, by GMS value-equality rather than string overlap. *Correctness* asks, of the value the answer asserts for the *asked* relation, whether it is grounded under the store's *calibrated* per-relation cut — the `HallucinationOracle`, not a raw distance: a value sitting where the committed fact sits passes, a real-but-wrong figure (the prior year, a different line item) is scored fabricated. *Completeness* is the `CompletenessEvaluator` reading: of the store-grounded

atoms the answer should contain, how many did it cover (recall). Correctness and completeness are measured on answered cases, so coverage (abstention) is a separate axis, not folded in.

```
from knowlytix.harness.testing.hallucination import HallucinationOracle
from knowlytix.harness.testing.completeness import CompletenessEvaluator
oracle = HallucinationOracle(store=store)      # calibrated per-relation cut
comp = CompletenessEvaluator(store)

for c in cohort:
    ans = pipe.query(c["question"])
    gt = gt_value(c["expected_answer"])
    hits = [f for f in ans.sources[:K] if num_eq(f.tail, gt)]
    precision.append(len(hits) / max(1, len(ans.sources[:K])))
    recall.append(1.0 if hits else 0.0)
    if ans.decision == "accept":               # answered-only
        head, rel, _ = golden(store, c)[0]
        correctness.append(oracle.assess_claim(head, rel, asserted(ans.answer, gt)).
    passed)
        completeness.append(comp.evaluate(ans.answer, [gt], {"head": head}).score)
```

```
=== GEODE-RAG DoE test (GMS as ground truth, n=150, k=3) ===
  precision@3    : 0.617
  recall@3       : 0.720
  correctness    : 0.780
  completeness   : 0.752
  abstention     : 0.153
```

The retriever places the right cell in the top three for 72% of questions, and of the answers it commits to, 78% assert a value the calibrated GMS confirms as grounded. The pipeline declines 15% of this cohort — it binds nothing, or self-verification rejects the draft — which on an all-answerable cohort is a cost; the same reflex is the safety property on unanswerable prose (Chapter 13).

15.3 Attributing the weakness

Because the cohort is a designed experiment with a fixed answer, a failure is a signal about the *presentation factor* that moved it. We fit a logistic regression of correctness on each factor with Benjamini-Hochberg correction — the shipped `suite.evaluate.attribute`, which delegates to the GraphDOE analyzer — and read off which factors significantly drive the failures.

```
from knowlytix.harness.suite import evaluate as ev
table = ev.attribute(rows, FACTORS, metric="correct")    # logistic + BH
for r in sorted(table, key=lambda r: r["p_value"]):
    print(f"  {r['factor']:16} p={r['p_value']:.3f}"
          f"{'  <-- significant' if r['significant'] else ''}")
```

```
  style            p=0.017  <-- significant
  paraphrase_depth p=0.169
  length           p=0.320
```

```
expertise        p=0.687
clarity          p=0.930
```

Only `style` significantly moves correctness: the GEODE-RAG is insensitive to clarity, length and expertise but sensitive to register (formal vs casual vs technical phrasing), which is where the next round of encoder tuning (Chapter 7) would focus.

15.4 GEODE-RAG versus chunk and pray

The same cohort and the same four metrics are run against the traditional baseline — fixed-size chunks, cosine top-k, a generator that answers from the passages and never abstains (Appendix E). The GMS is the oracle for both, so the comparison is exact.

metric	GEODE-RAG	chunk-and-pray
precision@3	0.617	0.351
recall@3	0.720	0.733
correctness	0.780	0.380
completeness	0.752	0.710
abstention	0.153	0.000
weakness (logistic)	style	clarity

Two differences stand out. The first is precision: the GEODE-RAG retrieves a *cell*, the baseline retrieves a *chunk*. A 60-word chunk holds many co-occurring numbers — FY2024 next to FY2025, cost next to revenue — so the baseline's top-k is nearly twice as polluted (precision 0.35 vs 0.62) even though recall ties. The second, and larger, is correctness: 0.78 against 0.38. Both systems answer with a real number copied from the report, but the baseline routinely copies the *wrong* real number — the prior-year figure, a neighboring line item — and when that value is scored against the asked relation under the store's calibrated cut, it reads as *fabricated*. The GEODE-RAG binds the asked attribute to its cell, so the value it commits to is the grounded one. Completeness is close (0.75 vs 0.71): both usually surface the expected atom. And only the GEODE-RAG can decline — abstention 0.153 against 0.000; the baseline's always-answer reflex is harmless on this all-answerable cohort but is exactly what produces a confident wrong answer on a question it should refuse. The two systems also break differently: the baseline's significant factor is `clarity`, because a misleadingly phrased question drags cosine retrieval to the wrong chunk, whereas the GEODE-RAG, binding through the graph, is unmoved by clarity and sensitive only to register.

GMS in practice — precision, and the right to abstain

Recall measures whether the answer is somewhere in the top-k; precision measures whether the retrieved unit *is* the answer. Chunk-and-pray can match recall because a wide window usually contains the figure — but the figure sits among several others, with nothing marking which is correct, which is why its precision collapses and why it cannot abstain. The GEODE-RAG retrieves the asserted cell and can decline when it binds nothing. The comparison is not that the baseline is worse at finding text; it is that finding text is not the same as grounding an answer, and a system that cannot abstain cannot be trusted on a question it should refuse.

15.5 Limits

The cohort is answerable by construction: the generators produce questions the graph can answer, so it measures retrieval quality on in-scope questions, not the abstention behavior on out-of-scope prose — that is measured separately in Chapter 13, and it is where the baseline's zero abstention is a liability rather than an advantage. The metrics are run with the accept gate open (`accept_threshold=0`) so that retrieval and answer quality are measured directly; the accept/abstain operating point is calibrated separately in Chapter 14. Groundedness and completeness are scored on the claims a small model's answer asserts, so they fold in the generator's fidelity as well as the retriever's; the precision/recall figures isolate retrieval. Finally, the logistic attribution has the power of 150 runs over five factors — enough to surface a dominant factor, not enough to resolve fine interactions; a larger design would refine it.

Anti-patterns

- **Reporting recall without precision.** A wide retrieval window inflates recall while burying the answer among distractors. Precision@k is what exposes that a chunk is not a cell.
- **Reading correctness or completeness alone.** A system that never abstains can score well on both by always copying a plausible figure. They must be read alongside the abstention rate and against an out-of-scope cohort.
- **Leaving weakness to anecdote.** "It seems worse on hard questions" is not actionable. A logistic attribution over the design factors names the factor with a corrected p-value, which is where tuning should go.

Exercise — add an out-of-scope block

Append the report's Outlook and Risk-Factors prose questions (no graph answer) to the cohort and re-run both systems. Confirm the GEODE-RAG abstains on them while the baseline answers, and watch the baseline's confident-wrong count rise while its headline metrics barely move. The worked solution is in the chapter notebook.

15.6 Self-check

The notebook asserts the chapter's claim: on identical questions the GEODE-RAG retrieves the right cell far more precisely than chunk-and-pray, and is the only one of the two that can decline.

```
assert geode["precision_at_k"] > baseline["precision_at_k"] + 0.2
assert geode["abstention_rate"] > 0.0 and baseline["abstention_rate"] == 0.0
print(f"OK: precision {geode['precision_at_k']:.2f} vs "
      f"{baseline['precision_at_k']:.2f}; only GEODE abstains")
```

Cross-references. The cohort and the encoders it scores come from Chapters 6 and 7; threshold selection is Chapter 14; the self-verifier behind correctness is Chapter 12; out-of-scope abstention is Chapter 13; and the baseline under comparison is Appendix E. The capstone (Chapter 18) draws the conclusion these metrics support.

Part VIII

Production

Chapter 16

Pluggable LLMs and the distrusted dense fallback

A GEODE-RAG deployment has three LLM-shaped jobs: **extract** the query triples from the natural-language question (Chapter 8), **synthesize** the grounded answer from retrieved facts and spans (Chapter 11) and optionally **verify** the answer's own claims against the store (Chapter 12). These are distinct jobs with distinct cost and quality trade-offs, so the pipeline assigns a *different* backend to each role. This chapter describes the mechanism and then turns to the one component that departs from strict triple-mediated retrieval: the **dense vector fallback**.

Retrieval in GEODE-RAG is triple-mediated and the dense vector index is distrusted (Chapter 1). The dense fallback is where "distrusted" becomes operational. It is *off by default*; once enabled, it answers questions the graph cannot reach, but the answer is quarantined — flagged `verified=False`, routed as `dense_fallback` and stamped with a notice. A `strict_mode` switch allows a bank-grade deployment to refuse even that. A dense hit is a lead rather than proof. We return to coverage measurement in Chapter 13 and to calibration of the verification signal in Chapter 14.

16.1 The three roles, one config

`RagConfig` carries three backend slots. Only `llm` (the synthesizer) is required; `llm_extract` falls back to `llm`, and `llm_verify` falls back to `llm_extract`. This arrangement pairs a small, inexpensive extractor with a stronger synthesizer, or pins every role to one local Qwen for a fully offline deployment. For deterministic CI we use a scripted fake backend; the library accepts any `LLMBackend`.

Listing 16.1: Per-role backend assignment.

```
from knowlytix.knowledge.rag import RagConfig
from knowlytix.knowledge.llm_backend import LLMBackend


class ScriptedBackend(LLMBackend):
    # Deterministic fake backend for CI -- returns a fixed reply.
    # The library accepts any LLMBackend, so a scripted one makes the
    # notebook reproducible without a GPU. The real Qwen path is shown below.

    def __init__(self, name: str, reply: str = ""):
        self._name = name
```

```
        self._reply = reply

    def call(self, system: str, user: str, max_tokens: int = 2048) -> str:
        return self._reply

    @property
    def model_name(self) -> str:
        return self._name


# Per-role assignment: a different backend for extract / synthesize / verify.
cfg = RagConfig(
    llm=ScriptedBackend("synth"),
    llm_extract=ScriptedBackend("extract"),
    llm_verify=ScriptedBackend("verify"),
)
print("synthesize :", cfg.llm.model_name)
print("extract    :", cfg.extract_llm().model_name)
print("verify     :", cfg.verify_llm().model_name)
```

The resolution helpers `extract_llm()` and `verify_llm()` implement the fallback chain. An unassigned slot inherits from the one above it, so a single-model deployment requires a single argument.

Listing 16.2: The fallback chain: one backend, all roles.

```
# One backend for synthesis + extraction; verify inherits the extractor.
shared = ScriptedBackend("qwen-shared")
cfg2 = RagConfig(llm=shared)
assert cfg2.extract_llm() is shared      # llm_extract defaults to llm
assert cfg2.verify_llm() is shared       # llm_verify defaults to extract_llm()
print("all roles ->", cfg2.verify_llm().model_name)
```

For a production deployment every role is a local Qwen2.5-3B-Instruct. The agent runtime is Qwen-locked for research integrity (Chapter 5); the same model can back all three roles. The next listing requires the GPU and is executed in CI.

Listing 16.3: The real local-Qwen path (CI).

```
# CI ONLY -- loads Qwen on GPU. Do not run during authoring.
from knowlytix.knowledge.llm_backend import LocalTransformersBackend
from knowlytix.knowledge.geode import QWEN_3B

qwen = LocalTransformersBackend(QWEN_3B, device="cuda")
qwen_cfg = RagConfig(llm=qwen)        # all three roles -> local Qwen
assert qwen_cfg.extract_llm().model_name == QWEN_3B
```

16.2 Loading the trained store

The pipeline runs over the store that Foundation F2 built. We load it from disk rather than rebuild it, because rebuilding runs GEODE self-correction and training on the GPU (Chapter 4). The load is a CI step; the listings that follow describe the grounded behavior recorded in the corpus facts sheet.

Listing 16.4: Load the trained store from disk (CI).

```
# CI ONLY -- loads the trained store from disk (GPU for the model tensors).
import torch
from knowlytix.knowledge.store import GMSExpertStore
from knowlytix.knowledge.config import DocGMSConfig

REPO_ROOT = os.path.join(os.path.dirname(os.getcwd()), "code") if os.path.basename(os.
    getcwd()) == "notebooks" else os.getcwd()
STORE = os.path.join(REPO_ROOT, "data", "gms_annual_report_store")
store = GMSExpertStore(DocGMSConfig(store_path=STORE),
                       device=torch.device("cuda" if torch.cuda.is_available() else "cp
    u"))
assert store.load(), "store not found -- run scripts/build_store.py first"
print("loaded store at", store.store_path)
```

16.3 Default: a prose question abstains

Sections 6–8 of the Northwind annual report — Management Discussion and Analysis, Risk Factors and Outlook — carry *no triples*. They are the corpus's coverage blind spots (`coverage_ratio = 0.56`; see Chapter 13). A question that can only be answered from that prose has nothing to bind to, so the default triple-mediated pipeline abstains rather than guess. The MD&A paragraph, for instance, states that "Management does not consider any single customer to be material to consolidated results" — a qualitative claim with no authoritative figure.

Listing 16.5: The default config: dense quarantine off.

```
from knowlytix.knowledge.rag import RagPipeline

# Default config: dense_fallback is False (the bank-grade default).
default_cfg = RagConfig(llm=ScriptedBackend("synth"))
assert default_cfg.dense_fallback is False
assert default_cfg.strict_mode is False
```

A prose question run through this default abstains. The route stays `triple` and the answer remains `verified=True`: an abstention is a verified outcome, not a failure.

Listing 16.6: Default pipeline abstains on a prose question (CI).

```
# CI ONLY -- runs the pipeline against the loaded store.
PROSE_Q = "What does management say about customer concentration risk?"

pipe_default = RagPipeline(store, default_cfg)
ans_default = pipe_default.query(PROSE_Q)

# The MD&A prose has no triples to bind, so the default route abstains.
assert ans_default.decision == "abstain"
assert ans_default.route == "triple"
assert ans_default.verified is True          # an abstention is still verified-true
print(ans_default.notice)                    # bind-check abstention notice
```

16.4 Opt-in dense fallback — answered, but quarantined

When `dense_fallback=True`, the pipeline indexes the document's paragraph spans (via `build_spans`) into an `InMemoryVectorBackend`. When nothing binds, it searches that index and synthesizes from the raw passages. The result is quarantined: routed as `dense_fallback`, flagged `verified=False` and stamped with the notice *"Answer from dense fallback; NOT GMS-verified."*

Listing 16.7: Opt in to the distrusted dense path.

```
# Same prose question, dense fallback enabled.
dense_cfg = RagConfig(
    llm=ScriptedBackend(
        "synth",
        reply=("Management states it does not consider any single customer "
               "material to consolidated results."),
    ),
    dense_fallback=True,        # opt in to the distrusted path
    strict_mode=False,          # actually return the (flagged) dense answer
)
assert dense_cfg.dense_fallback is True
assert dense_cfg.vector_backend is None     # defaults to InMemoryVectorBackend
```

Listing 16.8: Dense fallback answers, flagged unverified (CI).

```
# CI ONLY -- dense fallback answers the prose question, flagged unverified.
pipe_dense = RagPipeline(store, dense_cfg)
ans_dense = pipe_dense.query(PROSE_Q)

assert ans_dense.route == "dense_fallback"
assert ans_dense.verified is False                          # quarantined
assert ans_dense.notice == "Answer from dense fallback; NOT GMS-verified."
assert ans_dense.dense_sources                              # the spans it used
# The dense answer points at MD&A prose (section 6), a known blind spot.
print(ans_dense.dense_sources[0].location)
print(ans_dense.answer)
```

GMS in practice — why dense is quarantined

The chunk-and-pray default (Chapter 1) returns the top-k nearest chunks and lets the model write an answer over them, with no signal that the answer was never verified against structured knowledge. GEODE-RAG keeps the dense index available but *quarantines* its output: a separate route, a falsy `verified` flag and a notice that travels with the answer into the audit record (Chapter 13). The geometric path is the source of truth; dense is an explicitly labeled exception, so a downstream consumer cannot mistake a vector hit for a verified fact.

16.5 `strict_mode` — refuse even the dense answer

A bank-grade deployment may decide that an unverifiable answer is worse than no answer. `strict_mode =True` keeps the dense index wired but downgrades the dense route back to an abstention, retaining the coverage signal without ever returning an unverified figure.

Listing 16.9: Strict mode: dense route downgraded to abstain.

```
strict_cfg = RagConfig(
    llm=ScriptedBackend("synth"),
    dense_fallback=True,
    strict_mode=True,           # dense route -> abstain
)
assert strict_cfg.dense_fallback is True
assert strict_cfg.strict_mode is True
```

Listing 16.10: Strict mode abstains despite dense being on (CI).

```
# CI ONLY -- strict mode abstains on the prose question despite dense being on.
pipe_strict = RagPipeline(store, strict_cfg)
ans_strict = pipe_strict.query(PROSE_Q)

assert ans_strict.decision == "abstain"
assert ans_strict.route == "triple"          # never reaches the dense route
assert ans_strict.verified is True
print(ans_strict.notice)
```

16.6 Pointing the fallback at a custom backend

The dense index is pluggable. A deployment implements `VectorBackend` — two methods, `index` and `search` — and passes it as `vector_backend`. The library default is `InMemoryVectorBackend` with an injectable encoder, so the fallback is offline-testable and requires no external service. The same `build_spans` machinery the pipeline uses internally gives each span a line-range provenance location (Chapter 3), even though the answer over it is unverified.

Listing 16.11: The VectorBackend contract and span provenance.

```
from knowlytix.knowledge.rag import InMemoryVectorBackend, build_spans
from knowlytix.knowledge.rag.dense import VectorBackend, DenseSpan

# build_spans chunks markdown into paragraph spans with line-range provenance.
report_md = "## 6. Management Discussion and Analysis\n\nManagement does not consider
    any single customer to be material.\n"
spans = build_spans(report_md, source_path="data/annual_report.md")
assert spans and spans[0].location.startswith("data/annual_report.md:")
assert spans[0].line_start >= 1                # header-only blocks are skipped
print(spans[0].location, "->", spans[0].text[:48])

# Any VectorBackend subclass can replace the default -- e.g. an external DB.
assert issubclass(InMemoryVectorBackend, VectorBackend)
```

16.7 Limits

The dense fallback does not make prose verifiable. It retrieves text the graph never triplified and lets the synthesizer write over it, so its output carries exactly the failure modes that Chapter 1 cataloged: top-k is not relevance, retrieved prose is not evidence and any number in a dense answer was parsed from text, not read from the Exact Numerical Memory. The `verified=False` flag and

the notice label this limitation; they do not remove it. The appropriate response to a recurring dense hit is not to trust it but to *close the blind spot* by extracting triples for that section (Chapter 13); the fallback is a stopgap for coverage, not a second source of truth. Per-role LLM selection, likewise, changes *who* extracts, synthesizes and verifies; it does not change *what* is verifiable, since only the triple route is GMS-verified, whatever model drives it.

Anti-patterns

- **One model for everything by reflex.** Using the same large hosted model for extraction, synthesis and verification is convenient but wasteful: extraction is inexpensive and benefits little from a frontier model, whereas verification calls for a model that will not rationalize the draft it is checking. Roles should be assigned deliberately rather than left to a single default.
- **Trusting a dense hit.** A high cosine score indicates textual similarity, not factual support. Treating a `dense_fallback` answer as if it were verified — dropping the `verified=False` flag or the notice before it reaches a consumer — reintroduces exactly the hallucination surface the triple route eliminates.
- **Dense on by default.** Shipping with `dense_fallback=True` (or, worse, ignoring `strict_mode`) turns an opt-in stopgap into the silent primary path. Coverage gaps then disappear into plausible-sounding prose answers instead of surfacing as abstentions that can be measured and closed.

Exercise — Point the fallback at your own backend

Implement a `VectorBackend` subclass — it needs only `index` and `search` — backed by a store of any kind (even a trivial keyword scorer). Pass it to `RagConfig(vector_backend=...)` with `dense_fallback=True`, query the MD&A question and confirm that the answer still returns `verified=False` on the `dense_fallback` route. Then set `strict_mode=True` and confirm that the custom backend is bypassed in favor of an abstention. The worked solution is in the chapter notebook.

Self-check. The notebook's final cell asserts the config-level contract for all three modes on CPU — default (`dense_fallback` off), opt-in (dense on, strict off) and strict (dense on, downgraded to abstain) — plus the per-role fallback chain. The CI-marked cells assert the runtime behavior against the trained store: default abstains; dense answers with `verified=False` and a notice; and strict abstains once more. Together they substantiate the chapter's claim. Chapter 17 takes up persistence of verified outputs to an external store.

Chapter 17

Persisting to an External Store (KAL / Postgres)

A trained GMS store is a directory on local disk: a model checkpoint, an adapter, an ENM table and the source markdown. That suffices for a single notebook, but a deployed system serves many callers, survives restarts and is audited by people who never see the training machine. For that, the *graph*—the triples, their provenance and their GEODE verification—must live in a real database.

KAL (`knowlytix.kal`, the Knowledge Adapter Layer) is the backend-agnostic knowledge-graph store: a single `KnowledgeAdapter` contract over Postgres/pgvector, an in-memory mock, JSONL and others. The module `knowlytix.knowledge.rag.kal_sink` is the one-way bridge from a built RAG store into KAL. This chapter converts the Northwind FY2025 store's triples to `KALTriple` records, persists them through the offline mock adapter, reads them back, stamps GEODE verification and shows the gated Postgres path. It builds on the retrieval results of Chapter 10 (answering through the GMS) and on the binding of entities established in Chapter 9; persisting changes *where* the graph lives, not *what* it asserts.

Note. *T*

he trained GMS model checkpoint is a separate artifact and is **not** KAL's concern. Only the graph (triples + provenance + verification) is persisted. ENM numeric values ride along as triple `object_literal`s.

17.1 What KAL stores

A `KALTriple` carries more than `(subject, predicate, object)`. Its object is *either* a `KALNode` (an entity) *or* a `KALLiteral` (a scalar)—exactly one, enforced by a model validator. A `TripleProvenance` records `source`, `extractor` and the `source_text` the triple was read from. A `VerificationMetadata` records GEODE's `confidence` and `verification_status` and travels with the triple across federation. Because every fact carries quantified verification as first-class data, a downstream consumer filters on confidence without re-running GEODE.

17.2 A corpus-grounded fixture

`store_to_kal_triples` reads only three attributes of a built store: `store.triples` (an iterable of `(head, relation, tail)` tuples), `store.markdown` (for provenance recovery) and `store.store_path`

(the default `source`). We substitute a small object with exactly those attributes, populated from the canonical corpus facts, so the chapter runs offline—no GPU, no Qwen, no trained store. In CI this is replaced with the real store; the conversion code is identical.

```
from dataclasses import dataclass, field

# (head, relation, tail) tuples  verbatim from data/corpus_facts.md.
# Numeric tails are exact ENM values; entity tails are graph nodes.
CORPUS_TRIPLES = [
    ("cloud platform", "has_revenue", "120.0"),
    ("cloud platform", "has_headcount", "340.0"),
    ("cloud platform", "has_division", "technology"),
    ("devices", "has_revenue", "80.0"),
    ("devices", "has_division", "technology"),
    ("logistics", "has_revenue", "95.0"),
    ("logistics", "has_division", "operations"),
    ("retail", "has_revenue", "60.0"),
    ("retail", "has_division", "operations"),
    ("total", "has_revenue", "355.0"),
    ("technology", "has_region", "north america"),
    ("technology", "has_head", "dana cole"),
    ("operations", "has_region", "europe"),
    ("revenue", "has_fy2025", "355.0"),
    # in_section triples exist in the real store but are excluded by
    # store_to_kal_triples' default exclude_relations=('in_section',):
    ("cloud platform", "in_section", "segment performance"),
]

# A minimal markdown stand-in so the ProvenanceLedger can recover
# source_text for the table-cell triples (line/char spans).
FIXTURE_MD = (
    "## 1. Segment Performance\n"
    "| Segment | Division | Revenue | Headcount |\n"
    "| --- | --- | --- | --- |\n"
    "| Cloud Platform | Technology | 120.0 | 340.0 |\n"
    "| Devices | Technology | 80.0 | 210.0 |\n"
    "| Total | All | 355.0 | 1500.0 |\n"
)

@dataclass
class FixtureStore:
    """Stands in for a built RAG store (offline). Same attribute surface
    that kal_sink reads."""
    triples: list = field(default_factory=lambda: list(CORPUS_TRIPLES))
    markdown: str = FIXTURE_MD
    store_path: str = "data/gms_annual_report_store"

store = FixtureStore()
len(store.triples)
```

17.3 Listing 1 — Converting triples to KALTriple

`store_to_kal_triples` maps each `(h, r, t)` to a `KALTriple`. Numeric tails become typed literals; entity tails become object nodes; the `in_section` relation is dropped by the default `exclude_relations`.

```
from knowlytix.knowledge.rag.kal_sink import store_to_kal_triples

kal_triples = store_to_kal_triples(store, extractor="geode")

print(f"converted {len(kal_triples)} triples "
      f"(from {len(store.triples)} store triples; in_section excluded)")

# A numeric (literal) triple vs an entity (node) triple.
rev = next(t for t in kal_triples
           if t.subject.normalized_name == "cloud platform"
           and t.predicate == "has_revenue")
div = next(t for t in kal_triples
           if t.subject.normalized_name == "cloud platform"
           and t.predicate == "has_division")

print("numeric tail  ->", rev.object, "| literal:", rev.object_literal)
print("entity tail   ->", div.object, "| literal:", div.object_literal)
print("provenance    ->", rev.provenance)
```

Fourteen of the fifteen store triples convert (the single `in_section` triple is excluded). The revenue triple lands with `object_literal=KALLiteral(value='120.0', datatype='number')` and `object=None`; the division triple lands with `object=KALNode(name='technology')` and `object_literal=None`. The exact figure was never re-parsed from prose—it carried through from the ENM value as a typed literal, and its `provenance.source_text` is the table cell it was read from (recovered via a `ProvenanceLedger` built from `store.markdown`; the provenance ledger is introduced in Chapter 3).

17.4 Listing 2 — Persist offline, then read back

`MockKnowledgeAdapter` is KAL's in-memory adapter; it lives in production source, not `tests/`, so that the offline persistence path is real code. `persist_store_to_kal_sync` is the async→sync bridge around the async `insert_triples`. We seed the mock's `fixed_triples` with the converted records so a subsequent `query_triples` reads them back—a full round trip.

```
import asyncio
from knowlytix.kal import KALQuery
from knowlytix.kal.adapters.mock import MockKnowledgeAdapter
from knowlytix.knowledge.rag.kal_sink import persist_store_to_kal_sync

# Seed the mock with what we will persist, so reads return them.
adapter = MockKnowledgeAdapter("geode-mock", fixed_triples=kal_triples)

# Write through the sync bridge (async insert_triples under the hood).
inserted = persist_store_to_kal_sync(adapter, store, tenant_id="northwind")
print(f"persisted {inserted} triples")

# Read them back through the adapter's async query API.
result = asyncio.run(adapter.query_triples(KALQuery(), tenant_id="northwind"))
```

```
read_back = result.triples
print(f"read back {len(read_back)} triples; errors={result.errors}")

for t in read_back[:3]:
    tail = (t.object_literal.value if t.object_literal
            else t.object.name)
    print(f"  {t.subject.name} -- {t.predicate} --> {tail}")
```

`inserted` equals the converted-triple count, and the read-back set has the same length: the graph survives the round trip with its structure intact. The mock does not persist across process restarts—that is the Postgres adapter's role—but it exercises the same `KnowledgeAdapter` Protocol the real backend implements, so the persist code is validated offline.

GMS in practice —

Chunk-and-pray default: dump chunks and embeddings into a vector database; provenance and verification are lost at the boundary.
Geometric counterpart: persist `KALTriples`—each carries its `source_text` span and a `VerificationMetadata` confidence, so the external store remains auditable. The vector DB stores blobs; KAL stores a verifiable graph.

17.5 Listing 3 — Stamping GEODE verification

Pass `confidence=...` and every triple gets a `VerificationMetadata` with `verification_status='verified'` and `verifier='geode'`.

```
verified_triples = store_to_kal_triples(store, extractor="geode",
                                        confidence=0.97)
v_adapter = MockKnowledgeAdapter("geode-verified",
                                 fixed_triples=verified_triples)
v_result = asyncio.run(v_adapter.query_triples(KALQuery()))

sample = v_result.triples[0]
vm = sample.verification
print("confidence        ->", vm.confidence)
print("verification_status ->", vm.verification_status)
print("verifier          ->", vm.verifier)

# Without confidence, verification is None (the default in Listing 1).
print("unstamped verification ->", kal_triples[0].verification)
```

The stamped triple reads back `VerificationMetadata(confidence=0.97, verification_status='verifie verifier='geode')`; the unstamped Listing-1 triples have `verification=None`. Because verification travels with the triple, a consumer can filter on `min_confidence` at query time without re-running the GEODE self-correction loop of Chapter 5, which produces these confidence values.

17.6 Listing 4 — The Postgres path (gated)

The real backend is `kal_postgres_adapter`, which builds a pgvector-backed adapter through KAL's `AdapterFactory`. It requires a live database with KAL's migrations applied, so this cell

is gated on a KAL_PG_DSN environment variable and is a no-op in offline CI. The conversion and persist calls are byte-identical to the mock path; only the adapter differs.

```
import os
from knowlytix.knowledge.rag.kal_sink import (
    kal_postgres_adapter, persist_store_to_kal_sync)

PG = os.environ.get("KAL_PG_DSN")  # e.g. host=...;db=...;user=...;pw=...
if PG:
    parts = dict(kv.split("=", 1) for kv in PG.split(";"))
    pg_adapter = kal_postgres_adapter(
        host=parts["host"], database=parts["db"],
        username=parts["user"], password=parts["pw"])
    n = persist_store_to_kal_sync(pg_adapter, store,
                                  tenant_id="northwind", confidence=0.97)
    print(f"persisted {n} triples to Postgres")
else:
    print("KAL_PG_DSN not set -- skipping live Postgres persist (offline CI)")
```

With no DSN, the cell prints the skip notice and CI remains offline. Directed at a migrated database, the same `store` persists into a tenant-scoped graph with pgvector similarity search available on the nodes. Credentials are passed only to the builder; KAL encrypts them at rest.

Anti-patterns

- **Writing bespoke connectors.** Hand-written psycopg/Neo4j glue duplicates KAL's tenant scoping, deduplication, provenance and verification handling, and drifts from the contract. Persist through the `KnowledgeAdapter` Protocol; change backends by changing the adapter, not by rewriting the sink.
- **Storing model blobs in the graph database.** The trained GMS checkpoint is not a graph artifact. Inserting tensors into KAL enlarges the store and complicates the audit surface. Persist triples, provenance and verification only; version the checkpoint separately.
- **Dropping provenance and verification at the boundary.** Persisting bare `(s, p, o)` discards the metadata that makes the graph auditable. Always carry `source_text` and, where GEODE ran, a `confidence`.

Exercise — P

ersist the store with a stamped `confidence` and read the verification back from a round-tripped triple. Confirm that confidence is preserved, that the `verification_status` is `"verified"` and that an unstamped conversion leaves `verification` as `None`. (Worked solution: Listing 3 in the companion notebook.)

17.7 Limits

This bridge is one-way and lossy by design. It persists the graph, not the trained geometry: the GMS model cannot be reconstructed from KAL, so the store checkpoint must be versioned separately. The mock adapter validates the persist *code* but is not a database; it does not survive

a process restart, deduplicate inserts or enforce tenant isolation, all of which only the Postgres backend provides. The `KALTriplePattern` object field matches node objects by name only, so literal-object (numeric ENM) triples are not reachable by an object-pattern query; they are filtered by subject and predicate instead. Finally, persistence does not re-verify: a stamped `confidence` is the value GEODE produced at extraction time, not a fresh check against the live database.

Cross-references. The provenance recovered here is built in Chapter 3; the verification confidence originates in the GEODE self-correction loop of Chapter 5 and the answer verifier of Chapter 12. The triple-mediated representation persisted here is introduced in Chapter 8. For the KAL contract and adapter design in full, see the `kal-knowledge-adapter-layer` reference and Appendix D (reproducing the artifacts). The end-to-end system that consumes this external store is assembled in Chapter 18. This chapter does not duplicate KAL's internal storage schema; that reference is authoritative.

Part IX

Capstone

Chapter 18

Capstone: Summary, Conclusion and Lessons Learned

Every preceding chapter built one part of a complete GEODE-RAG over Northwind Industries' annual report, and ran it on the same corpus so the parts compose. This chapter introduces no new pipeline. It recapitulates the design, states the verdict the designed experiment supports, and distills the lessons worth carrying to a corpus that is not this one.

18.1 The system in one view

The system is a three-stage build followed by an online answer path, and each stage is a chapter.

- **Build the graph.** Regex ingestion turns the report into typed triples and exact figures; GEODE self-correction cleans the graph before it is indexed; the geometric model and Exact Numerical Memory are trained and persisted (Chapters 4, 5, with the substrate of Chapter 2 and the provenance ledger of Chapter 3).

- **Generate data and tune the encoders.** The store is its own oracle: a designed experiment mines it for questions with known answers and enriches them across presentation factors (Chapter 6); that data tunes the v-encoder to bind a paraphrase to the right attribute and the u-encoder to separate a true claim from a contradictory one (Chapter 7).

- **Answer through the graph.** A question is parsed into query triples, bound to graph vocabulary, retrieved as asserted facts with provenance, and synthesized into a grounded answer (Chapters 8, 9, 10, 11).

- **Refuse to be wrong.** The answer is self-verified against the geometry, the system abstains on a prose blind spot rather than guess, and every decision gate reads a calibrated operating point (Chapters 12, 13, 14).

- **Measure and ship.** The assembled system is evaluated with the same designed experiment and compared head-to-head with the traditional baseline (Chapter 15); pluggable backends and external persistence make it deployable (Chapters 16, 17).

18.2 The verdict

On the 150-question designed cohort, scored with the GMS as the oracle, the GEODE-RAG outperforms the chunk-and-pray baseline of Appendix E on the axes that bear on trust (Chapter 15). It retrieves a *cell* at precision@3 0.62 against the baseline's 0.35: a chunk holds many co-occurring numbers, so a wide window finds the figure (recall ties, 0.72 against 0.73) but cannot say which figure is the answer. The gap widens at correctness — 0.78 against 0.38. Both systems answer with a real number from the report, but the baseline routinely commits to the *wrong* real number (the prior-year figure, a neighboring line item), which the store's calibrated cut scores as fabricated; the GEODE-RAG binds the asked attribute to its cell, so the value it commits to is the grounded one. Completeness is close (0.75 against 0.71). And only the GEODE-RAG can decline — abstention 0.153 against 0.000; the baseline's always-answer reflex is harmless only because every question here is answerable, and is exactly the mechanism that yields a confident wrong answer on a question that should be refused. The two systems even fail differently: the baseline is significantly degraded by *misleading* phrasing, which drags cosine retrieval to the wrong chunk, while the GEODE-RAG, binding through the graph, is unmoved by clarity and sensitive only to register.

18.3 Lessons learned

1. **Retrieve a cell, not a chunk.** The measurable edge is precision. Similarity search returns text that contains the answer; triple-mediated retrieval returns the asserted fact. Recall is easy; precision is the trust metric.

2. **Ingest once, recall exact.** An authoritative number parsed once into Exact Numerical Memory is recalled byte-exact and never re-read from text at answer time — the single change that removes the most common confident-wrong failure.

3. **Let the graph be the oracle.** Generating the test and training data from the store, with a designed experiment over presentation factors, gives provably correct answers and a factor structure that attributes failures — neither of which a hand-written cohort provides.

4. **Tune encoders in context, and respect the geometry.** Bind on the contextual question, not a keyword; and learn contradiction from genuine conflicting values with a transform that can move tension — a rotation cannot.

5. **Calibrate every gate; abstain over guess.** Each decision reads an operating point fit from data, and a question the graph cannot answer is refused. The right to abstain is the property the baseline structurally lacks.

6. **Verify with geometry, not a second model.** A confident wrong number is caught because its value is distant from the asserted fact in the store's exact memory — a measurement, not one language model judging another.

7. **Cite the cell.** A trustworthy answer carries provenance back to the source table cell, not to a span of surrounding prose.

18.4 Limits, and when chunk and pray is enough

This system was demonstrated on one small, clean, single-document report whose authoritative numbers live in tables that cross-check each other. It does not establish that the technique scales

unchanged to long, multi-document, noisy corpora, and the designed cohort is answerable by construction, so the precision and abstention advantages shown here are the in-scope half of the story; the out-of-scope half — where the baseline's zero abstention becomes a liability — is argued in Chapter 13. The converse also holds: for exploratory search over prose, where there is no authoritative figure to get wrong and no requirement to abstain, the chunk-and-pray baseline is simpler and adequate. The case for the GEODE-RAG is specific — it is the architecture for answers that must be exact, cited and refusable.

18.5 The bridge onward

The pipeline exposes a single grounded tool: a query that returns an answer with its decision, its provenance and its audit trail. That is the surface a governed agent's executor expects, and wiring it into an agent loop is the hand-off to the agent book, sketched in Appendix A. The substrate and calibration method are recapped self-contained in Appendix C; a comparison with alternative frameworks is in Appendix B; and the one-command path from clone to a runnable book is Appendix D.

Appendix A

Plugging GEODE-RAG into a governed agent

The capstone (Chapter 18) assembled a complete GEODE-RAG system over the Northwind Industries FY2025 annual report: ingest and GEODE-correct, trained store, triple-mediated query with binding, multi-hop, provenance, self-verification and abstention. That system answers questions. An *agent* does more: it plans, calls tools and acts. This appendix shows how a retrieval system designed to be verifiable plugs into an agent designed to be governed.

The mechanism is a single tool. We wrap the entire pipeline as `search_report`: a typed, gated tool whose output carries the grounded answer, its provenance and its accept/abstain decision. This is the bridge to the agent book, *Beyond Prompt and Pray*, whose capstone agent calls a structurally identical `search_policy` tool over a banking-policy GMS. We cross-refer that book for the full `GovernedToolExecutor` and its gate protocol (agent book Ch 6, Ch 15); here we reproduce the *shape* in a few lines so this appendix stands alone, and we do not duplicate the agent material. The trust properties that this tool propagates were established earlier: provenance in Chapter 3, self-verification in Chapter 12 and abstention in Chapter 13.

Everything below is grounded in `data/corpus_facts.md`. The only GPU/Qwen-touching cell is the store reload plus the real-Qwen `RagConfig`, which the lead runs in CI; the deterministic path uses a scripted fake `LLMBackend` so the notebook self-checks without a model.

The retrieval substrate beneath this tool was developed across the body of the book: the document-to-graph transformation in Chapter 4, GEODE self-correction in Chapter 5, triple-mediated retrieval in Chapter 8 and grounded synthesis in Chapter 11.

A.1 A scripted backend for deterministic CI

The pipeline requires an `LLMBackend` for query-triple extraction and grounded synthesis. For a reproducible notebook we script one: it answers the single question this appendix asks, returning the canonical query triple and an answer that quotes only the retrieved fact. The real Qwen path is shown in §A.5. The interface is the real one — `knowlytix.knowledge.llm_backend.LLMBackend`.

```
from knowlytix.knowledge.llm_backend import LLMBackend


class ScriptedBackend(LLMBackend):
    """Deterministic stand-in for Qwen so the bridge self-checks offline.
```

```
    extract: maps the appendix's question to its canonical query triple
             (cloud platform, has_revenue, ?). synthesize: echoes the one
             retrieved figure verbatim -- no parametric memory, no other number.
    """

    @property
    def model_name(self) -> str:
        return "scripted-fake"

    def call(self, system: str, user: str, max_tokens: int = 2048) -> str:
        low = (system + " " + user).lower()
        if "query triple" in low or "extract" in low or "relation" in low:
            return "cloud platform | has_revenue | ?"
        # synthesis: answer ONLY from the evidence block we were handed
        return "Cloud Platform reported revenue of 120.0 for FY2025."
```

A.2 The pipeline behind the tool

We reload the store that Foundation F2 trained and wrap it bank-grade: dense fallback off, strict mode on, self-verification on. This is the only cell that touches the store on disk — the lead runs it in CI. `RagConfig` and `RagPipeline` are the real library symbols from `knowlytix.knowledge.rag`; the bank-grade configuration is the one Chapter 18 settled on.

```
# CI-ONLY: loads the trained store + builds the pipeline. Skipped at authoring
# time (one shared GPU). The bridge logic below does not depend on this cell.
import torch
from knowlytix.knowledge.query import DocGMSConfig, GMSExpertStore
from knowlytix.knowledge.rag import RagConfig, RagPipeline

STORE_PATH = os.environ.get("GMS_RAG_STORE", os.path.join(
    os.path.join(os.path.dirname(os.getcwd()), "code") if os.path.basename(os.getcwd())
     == "notebooks" else os.getcwd(),
    "data", "gms_annual_report_store"))


def build_pipeline(llm: LLMBackend) -> RagPipeline:
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    config = DocGMSConfig(store_path=STORE_PATH, ingest_mode="regex")
    store = GMSExpertStore(config, device=device)
    if not store.load():
        raise RuntimeError(f"no store at {STORE_PATH}; run scripts/build_store.py")
    rag = RagConfig(
        llm=llm,
        binding="embedding",      # paraphrases bind (Chapter~\ref{ch:embedding-sft})
        dense_fallback=False,      # quarantined dense index never used
        strict_mode=True,          # graph-only; abstain rather than guess
        verify_llm_output=True,    # GMS self-verification (Chapter~\ref{ch:answering})
        on_verify_fail="abstain",  # a contradicted claim never ships
    )
    return RagPipeline.from_store(store, rag)
```

A.3 The tool: typed in, typed out, provenance carried

The agent book registers tools as typed records with pydantic input/output schemas and a risk level (agent book Ch 6). We mirror that contract here without importing the private `agentlab` package — the schema *is* the contract.

The tool's output carries the answer *and* its provenance (the `file:line:char` span and the raw source text) *and* the decision/verified flags. An agent that calls `search_report` receives a citable, abstention-aware result, not a bare string. When the pipeline abstains the tool returns `decision="abstain"` with an empty `sources` list, and the agent branches on that instead of fabricating.

```
from pydantic import BaseModel, Field


class SearchReportInput(BaseModel):
    question: str = Field(..., description="A natural-language question about the
    annual report.")


class Citation(BaseModel):
    triple: tuple[str, str, str]
    location: str | None = None   # file:line:char span
    raw: str | None = None        # the source text the fact came from


class SearchReportOutput(BaseModel):
    answer: str
    decision: str                 # accept | abstain | escalate
    verified: bool                # GMS-verified, not LLM-judged
    confidence: float
    route: str                    # triple | dense_fallback
    sources: list[Citation] = Field(default_factory=list)
    notice: str | None = None
```

The tool function itself is thin: the pipeline does the work, and the wrapper only adapts a `RagAnswer` into the agent's output schema, preserving provenance and the abstain decision.

```
def search_report(pipe: RagPipeline, question: str) -> SearchReportOutput:
    """Answer a question through GEODE-RAG; return a typed, cited result.

    The pipeline does the work (extract -> bind -> retrieve -> synthesize ->
    verify -> decide). This wrapper only adapts RagAnswer into the agent's
    tool-output schema, preserving provenance and the abstain decision.
    """
    ans = pipe.query(question)
    citations = [
        Citation(triple=(f.head, f.relation, f.tail), location=f.location, raw=f.raw)
        for f in ans.sources
    ]
    return SearchReportOutput(
        answer=ans.answer,
        decision=ans.decision,
        verified=ans.verified,
        confidence=ans.confidence,
```

```
        route=ans.route,
        sources=citations,
        notice=ans.notice,
    )
```

A.4 A minimal governed loop

The agent book runs every tool call through a `GovernedToolExecutor`: gates fire first (schema, policy, plausibility), then the tool runs, and the result is recorded for the audit trail (agent book Ch 6, Ch 15). We reproduce the shape in a few lines so the appendix stands alone; the production executor with its full gate protocol lives in the agent book — we do not duplicate it.

Two gates suffice to show the contract: a **syntax** gate that validates the arguments against the tool's input schema and a **risk** gate that records the tool's risk level. A real deployment adds policy gates (agent book Ch 6).

```
from dataclasses import dataclass, field
from typing import Any, Callable


@dataclass(frozen=True)
class Tool:
    """Typed, gated tool record -- the agent book's Tool contract, minimal form."""
    name: str
    description: str
    input_schema: type[BaseModel]
    fn: Callable[..., BaseModel]
    risk: str = "low"


@dataclass
class ToolResult:
    tool_name: str
    output: Any = None
    error: str | None = None
    allowed: bool = False
    gates: list[str] = field(default_factory=list)


class GovernedExecutor:
    """Validate args against the schema, record risk, then run the tool.

    Mirrors agent book GovernedToolExecutor.execute: gates first, tool second,
    everything captured for the audit trail. A deny short-circuits before fn.
    """

    def __init__(self, tools: dict[str, Tool]):
        self._tools = tools
        self.audit: list[dict] = []

    def execute(self, name: str, arguments: dict) -> ToolResult:
        tool = self._tools.get(name)
        if tool is None:
```

```
            return ToolResult(name, error=f"unknown tool {name!r}")
        try:
            parsed = tool.input_schema(**arguments)  # syntax gate
        except Exception as e:
            return ToolResult(name, error=f"schema: {e}", gates=["syntax:deny"])
        gates = ["syntax:allow", f"risk:{tool.risk}"]
        output = tool.fn(**parsed.model_dump())
        result = ToolResult(name, output=output, allowed=True, gates=gates)
        self.audit.append({
            "tool": name, "gates": gates,
            "decision": getattr(output, "decision", None),
            "verified": getattr(output, "verified", None),
        })
        return result
```

Registering the tool binds the pipeline into the tool's `fn`, so the registry holds a callable that takes only schema arguments — exactly as the agent book wires a stateful tool.

```
# CI-ONLY: needs the real pipeline. The deterministic self-check (below) injects
# a fake pipeline so this same code runs offline.
def make_executor(pipe: RagPipeline) -> GovernedExecutor:
    tool = Tool(
        name="search_report",
        description="Search the Northwind FY2025 annual report via GEODE-RAG; returns a
     grounded, cited answer or abstains.",
        input_schema=SearchReportInput,
        fn=lambda question: search_report(pipe, question),
        risk="low",
    )
    return GovernedExecutor({"search_report": tool})
```

A.5 The real Qwen path

Everything above runs on the scripted backend for determinism. In production the synthesis and extraction LLM is local Qwen2.5-3B-Instruct — no API key. We swap the backend and the bridge is unchanged: that interchangeability is the reason for routing the model through `LLMBackend` (Chapter 16 develops per-role LLM selection). The lead runs this in CI; we show it as a listing because one GPU is shared across authors.

```
# CI-ONLY (real Qwen). Build the pipeline with a local Transformers backend
# and run the governed tool end to end.
def run_with_qwen(question: str) -> ToolResult:
    from knowlytix.knowledge.llm_backend import LocalTransformersBackend
    qwen = LocalTransformersBackend("Qwen/Qwen2.5-3B-Instruct")
    pipe = build_pipeline(qwen)
    executor = make_executor(pipe)
    return executor.execute("search_report", {"question": question})


# result = run_with_qwen("What was Cloud Platform's revenue?")
# print(result.output.answer, result.output.sources[0].location)
```

A.6 Self-check: a grounded, cited answer through the executor

We exercise the bridge without a GPU by injecting a fake pipeline that returns the corpus-true fact for the canonical question: `('cloud platform', 'has_revenue', '120.0')` (see `data/corpus_facts.md`). The fake exposes the same `.query()` → `RagAnswer` contract the real `RagPipeline` does, so the tool and executor code paths are exercised verbatim. The assertion checks that the answer flows through the governed executor *with* provenance and *without* losing the verified/decision flags — which is what the bridge exists to guarantee.

```
from knowlytix.knowledge.rag.pipeline import RagAnswer
from knowlytix.knowledge.rag.retrieve import RetrievedFact

# Corpus-true fact (data/corpus_facts.md): cloud platform has_revenue 120.0.
_FACT = RetrievedFact(
    head="cloud platform", relation="has_revenue", tail="120.0",
    score=0.0, confidence=1.0, source="triple",
    location="annual_report.md:15:553-558", raw="Cloud Platform | Technology | 120.0 |
    340",
)


class _FakePipeline:
    """Stands in for RagPipeline.query() so the bridge self-checks offline."""
    def query(self, question: str) -> RagAnswer:
        return RagAnswer(
            answer="Cloud Platform reported revenue of 120.0 for FY2025.",
            confidence=1.0, decision="accept", route="triple", verified=True,
            sources=[_FACT],
        )


_pipe = _FakePipeline()
_tool = Tool(
    name="search_report",
    description="GEODE-RAG over the annual report.",
    input_schema=SearchReportInput,
    fn=lambda question: search_report(_pipe, question),
    risk="low",
)
executor = GovernedExecutor({"search_report": _tool})

result = executor.execute("search_report", {"question": "What was Cloud Platform's
    revenue?"})
out = result.output

# 1. the call passed the governance gates and ran
assert result.allowed and result.error is None, result
assert "syntax:allow" in result.gates and "risk:low" in result.gates
# 2. the answer is grounded: it quotes the corpus figure, carries provenance,
#    and keeps the verified/decision flags through the executor
assert "120.0" in out.answer
assert out.decision == "accept" and out.verified is True and out.route == "triple"
assert out.sources, "the tool dropped provenance"
assert out.sources[0].triple == ("cloud platform", "has_revenue", "120.0")
```

```
assert out.sources[0].location == "annual_report.md:15:553-558"
# 3. the audit trail recorded the gated, verified call
assert executor.audit[-1]["verified"] is True
print("OK: search_report returned a grounded, cited answer through the governed executo
    r")
print(out.answer, "<-", out.sources[0].location)
```

A.7 Abstention crosses the bridge too

A governed agent must be able to distinguish *"I do not know"* from a wrong answer. Because the pipeline abstains on a prose blind-spot question (Chapter 13; the coverage blind spots in `data/corpus_facts.md` are MD&A, Risk Factors and Outlook), the tool surfaces `decision="abstain"` with empty `sources` and a notice. The agent reads that and routes to a human or a different tool — it never turns a miss into a fabricated number.

```
class _AbstainPipeline:
    def query(self, question: str) -> RagAnswer:
        return RagAnswer(
            answer="I cannot answer this from the available grounded evidence.",
            confidence=0.0, decision="abstain", route="triple", verified=True,
            notice="Question did not bind to known entities/relations.",
        )


_abstain_tool = Tool(
    name="search_report", description="GEODE-RAG.",
    input_schema=SearchReportInput,
    fn=lambda question: search_report(_AbstainPipeline(), question),
)
abstain_exec = GovernedExecutor({"search_report": _abstain_tool})
ab = abstain_exec.execute("search_report", {"question": "What does the Outlook section
    say?"})
assert ab.output.decision == "abstain" and not ab.output.sources
print("OK: the tool abstains on a blind-spot question, carrying the notice:")
print(ab.output.notice)
```

A.8 Exercise solution: a policy gate

The minimal executor has only a syntax gate and a risk tag. The subclass below adds a **policy** gate that denies an empty or over-long question *before* the tool runs — a coarse prompt-injection and cost guard. It mirrors the agent book's `Gate` protocol (Ch 6): check first, a deny short-circuits.

```
class PolicyGovernedExecutor(GovernedExecutor):
    """GovernedExecutor + a policy gate on the question argument."""

    MAX_LEN = 500

    def execute(self, name: str, arguments: dict) -> ToolResult:
        q = str(arguments.get("question", "")).strip()
        if not q or len(q) > self.MAX_LEN:
            res = ToolResult(name, error="policy: empty or over-long question",
```

```
                                allowed=False, gates=["policy:deny"])
            self.audit.append({"tool": name, "gates": ["policy:deny"],
                               "decision": None, "verified": None})
            return res
        return super().execute(name, arguments)


_pol = PolicyGovernedExecutor({
    "search_report": Tool(
        name="search_report", description="GEODE-RAG.",
        input_schema=SearchReportInput,
        fn=lambda question: search_report(_FakePipeline(), question),
    )
})
# a normal question still passes
ok = _pol.execute("search_report", {"question": "What was Cloud Platform's revenue?"})
assert ok.allowed and "120.0" in ok.output.answer
# an empty question is denied before the tool runs
bad = _pol.execute("search_report", {"question": "   "})
assert not bad.allowed and bad.gates == ["policy:deny"]
assert _pol.audit[-1]["gates"] == ["policy:deny"]
print("OK: policy gate denies an empty question, allows a real one")
```

A.9 Limitations

The bridge is an adapter, not a new guarantee. It carries through exactly the trust properties the pipeline already established and adds none of its own: if the store is uncalibrated, `search_report` will still accept a `link_predict` guess (Chapter 14 fits the `accept_threshold` that prevents this); if a question's answer lives in a coverage blind spot, the tool abstains rather than reaching into a dense index, which is correct but unhelpful to an agent that needed an answer. The minimal `GovernedExecutor` here is deliberately a sketch — it has a syntax gate and a risk tag, not the policy gates, plausibility checks, retry budget or fault-injection hooks of the agent book's real executor. We treat this appendix as the wiring diagram; the production governance lives in *Beyond Prompt and Pray*. Finally, registering the tool does not make the *agent* safe: a governed loop can still call the right tool with the wrong arguments or ignore an abstention. The tool gives the agent a citable, abstention-aware fact; whether the agent uses it well is the agent book's subject, not this one's.

Anti-patterns

Flattening the result to a string. Returning only `ans.answer` from the tool discards the provenance span, the `verified` flag and the abstain decision — the agent loses every signal that distinguished GEODE-RAG from chunk-and-pray, and a downstream reviewer can no longer cite the source. The structured output should always be returned.

Treating an abstention as a tool failure. If the executor maps `decision="abstain"` to an error or an empty retry, the agent will keep re-calling the tool or fall through to a model that guesses. Abstention is a *successful* answer that states "not in evidence"; the loop must branch on it, not retry it.

Re-deriving numbers in the agent. Once the tool returns `120.0` with its span, an agent

that "rounds" or "reconciles" that figure in a later step reintroduces the parsed-number hallucination the entire pipeline exists to prevent. The cited figure should pass through unchanged.

Exercise — Add a policy gate to the executor

The minimal `GovernedExecutor` has only a syntax gate and a risk tag. Add a `policy` gate that *denies* a `search_report` call whose `question` is empty or longer than 500 characters (a coarse prompt-injection / cost guard), returning a `ToolResult` with `allowed=False` and `gates=["policy:deny"]` *before* the tool runs. Confirm a normal question still passes and the audit trail records the denial. The worked solution is in the notebook; the gate's signature may be compared to the agent book's `Gate` protocol (Ch 6) — the production version checks the same way against richer policies.

GMS in practice — A retrieval tool an agent can be governed around

A chunk-and-pray retriever hands an agent a list of passages and a similarity score; the agent (or its LLM) then decides what the passages mean, and any number it reports was parsed from prose. The agent cannot cite a span, cannot distinguish a confident answer from a guess and has nothing to abstain on. The GEODE-RAG tool hands the agent a *decision*: a cited fact with provenance, a GMS-*verified* flag and an explicit `abstain` when the graph holds no answer. Governance moves from "hope the model read the passages correctly" to "gate on a typed, verifiable result" — which is the only kind of retrieval a governed agent can be held accountable for.

This closes the loop with *Beyond Prompt and Pray*: that book builds the governed agent and assumes a trustworthy retrieval tool; this book built the retrieval tool and, here, showed it integrating into the agent's executor. The capstone (Chapter 18) forward-referenced this appendix as the final step — ingest to corrected store to triple-mediated, verified, abstaining answers, and now to a tool a governed agent calls. The framework comparison in Appendix B and the GMS substrate in Appendix C situate this design among alternatives, and Appendix D documents how to reproduce the artifacts used throughout.

Appendix B

GEODE-RAG versus vector-RAG frameworks

This appendix compares contracts. LangChain and LlamaIndex are well suited to what they were built for: assembling chunk-and-retrieve pipelines quickly, with a large catalog of loaders, splitters, vector stores and prompt templates. GEODE-RAG is built for a different contract — the *grounding* discipline introduced in Chapter 1: retrieval is triple-mediated, the dense index is distrusted, every answer is verifiable against asserted facts and their source spans, and the system abstains rather than guess. The two are not interchangeable, and the table below is intended to help the reader decide which contract a given application requires.

Every row that makes a claim about a competing framework cites that framework's own documentation or default behavior as of this writing. Where a framework *can* be configured to approximate a GEODE-RAG property, the table records this in the "with effort" column rather than scoring it as absent — the distinction this appendix draws is between *default, verifiable* behavior and behavior that must be assembled and then trusted by the integrator.

B.1 What each framework optimizes for

A vector-RAG framework optimizes *recall over passages*: given a query embedding, it returns the top-k chunks whose embeddings are nearest, then hands those chunks to an LLM with an instruction to answer from them. The retrieval unit is a span of text; relevance is cosine similarity; provenance, where present, is "which chunk did this come from"; and verification, where present, is a second LLM call that judges the first — the "model judging a model" failure mode discussed in Chapter 1.

GEODE-RAG optimizes *verifiable grounding*. The retrieval unit is an asserted triple in a trained `GMSExpertStore`; relevance is whether a query triple *binds* to graph vocabulary (Chapter 9) and a fact is actually held (Chapter 10); provenance is a `file:line:char` span resolved through a `ProvenanceLedger` (Chapter 3); exact numbers come from Exact Numerical Memory (ENM), never parsed from a chunk; and verification is a geometric contradiction check of the answer's own claim triples against the asserted facts (Chapter 12), not a second LLM opinion.

B.2 Side-by-side: the same question, two contracts

Running the same question — "What was Northwind's total revenue in FY2025?" — through each contract shows the difference. The listings below are *illustrative*: the vector-RAG snippet is the canonical LlamaIndex/LangChain shape (pseudo-real; it does not run in this book's CI and is not

pinned to a framework version), while the GEODE-RAG snippet is library code copied verbatim from the pipeline used throughout this book.

Listing B.1: The vector-RAG contract (illustrative; not run in this book).

```
# Illustrative LlamaIndex/LangChain shape. Retrieval unit = a text chunk;
# relevance = cosine similarity; the answer's number is *parsed from prose*.
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader

docs  = SimpleDirectoryReader(input_files=["annual_report.md"]).load_data()
index = VectorStoreIndex.from_documents(docs)          # chunk + embed
qe    = index.as_query_engine(similarity_top_k=4)      # top-k passages
resp  = qe.query("What was Northwind's total revenue in FY2025?")

print(resp)                 # an LLM completion grounded in 4 retrieved chunks
print(resp.source_nodes)    # chunk-level provenance: which passage, not which cell
# No ENM: "355.0" reaches the user only if the model copies it faithfully from a
# chunk. No claim-level verification. No abstention when the chunk loses a tie.
```

The GEODE-RAG snippet below is the same pipeline assembled in Chapter 10, configured for the verifiable triple route only. Every property the table credits to GEODE-RAG is visible in the returned `RagAnswer`: the figure comes from ENM (`355.0`, byte-exact), the `sources` carry a `file:line:char` location, `verified` is `True`, and `audit()` emits the full evidence trail. This cell is library code; the lead executes it in CI (it loads the trained store and a local Qwen, so it is not run inside this appendix draft).

Listing B.2: The GEODE-RAG contract (real library; CI-executed by the lead).

```
from knowlytix.knowledge.store import GMSExpertStore
from knowlytix.knowledge.rag import RagConfig, RagPipeline

store = GMSExpertStore.load("data/gms_annual_report_store")  # trained, GEODE-corrected
rag   = RagConfig()                  # defaults: triple route only, dense OFF, abstain-
    on-fail
pipe  = RagPipeline.from_store(store, rag)

ans = pipe.query("What was Northwind's total revenue in FY2025?")

print(ans.answer)        # grounded synthesis containing the ENM figure 355.0
print(ans.decision)      # "accept"
print(ans.verified)      # True  -- triple route, not the distrusted dense fallback
for f in ans.sources:    # cell-level provenance, not chunk-level
    print(f.head, f.relation, f.tail, f.location, repr(f.raw))
# e.g. total has_revenue 355.0  :15:553-558  '355.0'
import json
print(json.dumps(ans.audit(), indent=2, default=str))   # full, replayable evidence
    trail
```

The vector-RAG engine returns a completion and a list of source chunks; nothing in its return value lets a downstream consumer assert that `355.0` is the authoritative figure rather than a number the model copied imperfectly, or hallucinated, from a nearby passage. The GEODE-RAG `RagAnswer` carries the asserted triple, the exact span it resolves to, the verification verdict and a decision the caller can gate on. This is the central difference: one returns text about the answer; the other returns the answer together with a proof that can be replayed (Chapter 11).

B.3 The comparison table

The columns separate three states: **Default** (what the framework does out of the box), **With effort** (achievable by assembling extra components that must then be trusted and maintained) and **GEODE-RAG** (default behavior of the library in this book). "N/A by design" marks a property a chunk-and-retrieve architecture cannot offer without changing its retrieval unit.

Property	**LangChain / LlamaIndex (default)**	**With effort**	**GEODE-RAG (default)**
Retrieval unit	Text chunk (passage)	Sub-document nodes, metadata filters	Asserted triple in a trained store
Grounding signal	Cosine similarity of embeddings	Hybrid / reranker scores	Triple *binding* + fact held in graph
Provenance	Chunk / node id	Line-level if the loader preserves it	`file:line:char` span via ledger
Exact numbers	Parsed from retrieved prose	Output parsers / tool calls (still LLM-mediated)	ENM, byte-exact, never parsed
Verification	None / LLM-judges-LLM	Self-consistency, second judge model	Geometric contradiction on claim triples
Abstention	Answers anyway (returns top-k)	Custom "no answer" prompt heuristic	Bind-check + accept/abstain decision
Multi-hop	Re-query / agent loop over chunks	Knowledge-graph index (LlamaIndex)	Variable-env triple chain (`?x`→`?`)
Coverage blind spots	Not measured	Manual eval	`coverage_report`: body-text-but-no-triples
Audit trail	Source chunks list	Callbacks / tracing add-ons	`RagAnswer.audit()`, replayable
Dense index trust	Trusted (it is the system)	—	Distrusted: off by default, flagged `verified=False`

Table B.1: GEODE-RAG versus vector-RAG frameworks. "With effort" means assemble-and-trust, not a default guarantee.

B.4 Cost and operational profile

The cost profile is not one-sided. A vector-RAG pipeline has one heavy offline step (embed every chunk) and one inexpensive online step (one embedding lookup plus one LLM call). GEODE-RAG front-loads more work: ingestion runs GEODE self-correction (Chapter 5) to train the store, which is more expensive than embedding chunks. Online, GEODE-RAG can make *more* LLM calls than a naive top-k pipeline — query-triple extraction, the schema-grounded repair retry, synthesis and optional self-verification — though each runs on a small local model (Qwen2.5-3B-Instruct), with no API key and no per-token bill. The trade-off is deliberate: more compute is spent to obtain a verdict that can be defended, and it is spent on hardware under the operator's control rather than a metered endpoint. For applications where a wrong number is inexpensive, the vector-RAG profile is preferable; for applications where a confident-wrong answer is the expensive failure (Chapter 15), the GEODE-RAG profile is the appropriate one.

A second operational difference is that the GEODE-RAG store is a queryable artifact, not an opaque embedding matrix. It can be persisted to an external knowledge-graph backend with provenance and verification intact (Chapter 17), any triple can be inspected and any answer can be audited after the fact. A vector index supports nearest-neighbor lookup and little else; explaining *why* a chunk surfaced requires re-running the similarity search.

GMS in practice — The default flips from "trust the top-k" to "prove it"

A vector-RAG framework's default is to retrieve the nearest chunks and answer. The GEODE-RAG default (`RagConfig()` with no arguments) is the verifiable triple route only: `dense_fallback=False`, `strict_mode` available, `on_verify_fail="abstain"` and an accept/abstain decision on every query (Chapter 13). Where the chunk-and-pray safe default is "return something," the geometric default is "return a grounded fact or decline." The dense index is not deleted — it is *quarantined*: opt-in, off by default and when used it is flagged `verified=False` with a notice (Chapter 16).

B.5 When a vector-RAG framework is the right choice

GEODE-RAG does not dominate every setting. A vector-RAG framework is the appropriate tool when the corpus is mostly unstructured prose with few authoritative facts to extract into triples; when approximate, semantically similar passages constitute an acceptable answer (open-domain Q&A, brainstorming, "find me relevant context"); when there are no exact numbers whose correctness is load-bearing; and when the cost of an occasional wrong-but-plausible answer is low. In those settings the GEODE-RAG ingestion cost and the effort of defining a schema yield little benefit, because there is no verifiable structure to ground against — and GEODE-RAG would abstain on most queries, which is correct behavior but not useful product behavior for that corpus.

GEODE-RAG is warranted when facts are structured (tables, hierarchies, functional relations), numbers must be exact, a wrong answer is expensive and "I do not know" is preferable to a confident guess — financial reports, compliance, clinical or legal lookup. The Northwind corpus is deliberately of the second kind.

Anti-patterns

- **Comparing on retrieval recall alone.** A benchmark that scores "did the right chunk appear in the top-k" rewards vector-RAG and is blind to the actual value of GEODE-RAG: provenance, exact numbers, verification and abstention. Compare on the trust metrics of Chapter 15 (provenance rate, abstention precision, zero confident-wrong), not on recall.
- **Bolting a "verifier" onto a vector pipeline and calling it verified.** A second LLM judging the first is the "model judging a model" failure described in Chapter 1; it is not the geometric contradiction check of Chapter 12. "With effort" in Table B.1 means *assemble and then trust*, not *equivalent guarantee*.
- **Reading exact numbers off a retrieved chunk.** Output parsers and tool-calling still take the figure from text the model produced. Only ENM yields a byte-exact number with a span; a parsed number must not be presented as authoritative.

Exercise — Map your corpus to the right contract

Take a corpus under one's own control. (1) Count how many of its load-bearing facts are *exact* (numbers in tables, functional relations such as one-CEO-one-FY-end) versus *approximate* (prose answered by paraphrase). (2) For each exact fact, decide whether a confident-wrong answer is inexpensive or expensive. (3) Score the corpus against Table B.1: if most rows that matter are in the GEODE-RAG column *and* confident-wrong is expensive, the geometric contract repays its ingestion cost; if the facts are mostly approximate prose, a vector-RAG framework is the appropriate choice and GEODE-RAG would mostly abstain. Record the deciding row. There is no notebook solution — the deliverable is the decision and the row that drove it.

B.6 Limits of this comparison

This appendix compares *contracts*, not tuned systems on a fixed benchmark. It does not claim that GEODE-RAG retrieves more passages, answers faster or beats a well-tuned vector pipeline on open-domain recall — on those axes a mature framework with a good reranker will often win, and GEODE-RAG does not attempt to compete there. The "with effort" column is necessarily coarse: LangChain and LlamaIndex are large, fast-moving toolkits, and a determined engineer can approximate several GEODE-RAG properties by composing their primitives. The distinction that survives is *default and verifiable* versus *assemble and trust*: GEODE-RAG ships the grounding, provenance, exact-number, verification and abstention guarantees as the default path, and exposes them in a single auditable `RagAnswer`; a vector-RAG framework leaves the integrator to build, wire and trust those guarantees. Finally, the comparison says nothing about corpora with no extractable structure — where there are no triples to mediate retrieval, GEODE-RAG offers no advantage and correctly abstains. For the document-to-graph extraction that produces those triples, see Chapter 4; for the geometry internals behind the verification and binding claims, see Appendix C and the GMS monograph rather than this appendix.

Appendix C

The GMS Substrate & Calibration

This book relies on the Geometric Memory Substrate (GMS) at the altitude of a RAG engineer: a trained `GMSExpertStore` is a register of facts that one can score, look up, query and contradict. The geometric memory model itself is introduced in Chapter 2; this appendix makes the book self-contained at the level of the store's interface. It recaps the six store primitives, shows how a trained store is assembled from a document and states the threshold-calibration method — cohort, sweep, operating point under a false-allow ceiling — once, so that the chapters that use it (Chapter 9, Chapter 14 and Chapter 13) can refer here instead of re-deriving it.

The geometry *behind* the primitives — rotors, admissibility caps and holonomy proofs — is deferred to the GMS monograph. It is not required in order to deploy a trustworthy RAG system, which needs only knowledge of what each primitive returns and when to apply it. The calibration sweep at the end of this appendix runs CPU-only with a deterministic injected encoder, so it reproduces a chosen operating point without a GPU, without the trained store and without Qwen.

C.1 The six store primitives

Everything the RAG pipeline does to the graph routes through six methods on `GMSExpertStore`.

Primitive	Question it answers	Return
`score_triple(h,r,t)`	How well does this asserted fact fit the trained geometry?	geodesic score, or `None`
`lookup_enm(cat,id)`	What is the *exact* stored number?	byte-exact float, or `None`
`query_triples(...)`	Which asserted edges match this pattern?	list of triples
`link_predict(...)`	What tail does the geometry *predict* (no asserted edge)?	ranked candidates
`check_holonomy(path)`	Is a relation composition path consistent?	residual / verdict
`tension_energy(a,b)`	How contradictory are two entities?	energy, or `None`

The design rule repeated throughout the book is the following: **`lookup_enm` for numbers, asserted `query_triples` before `link_predict` for facts and `tension_energy` for contradiction**. A predicted link is a hypothesis, not a fact (Chapter 10). The listing below asserts that the public surface exists; it does not load a trained store, because the GPU is shared across authoring runs.

```
from knowlytix.knowledge.store import GMSExpertStore
```

```
# The public surface this appendix recaps. We assert the methods exist;
# we do NOT load a trained store here (one GPU is shared across authors).
PRIMITIVES = (
    "score_triple", "lookup_enm", "query_triples",
    "link_predict", "check_holonomy", "tension_energy",
)
for name in PRIMITIVES:
    assert callable(getattr(GMSExpertStore, name)), name
print("six primitives present:", ", ".join(PRIMITIVES))
```

Output:

```
six primitives present: score_triple, lookup_enm, query_triples, link_predict,
    check_holonomy, tension_energy
```

Chapter 2 works these primitives against the live Northwind store; this appendix only fixes the contract.

C.2 Building a store

Two entry points assemble a trained store, both in `knowlytix.knowledge.geode.rag`:

- `build_rag_store(md_path, config, ...)` — the full ingest path: parse the document, extract triples, populate Exact Numerical Memory (ENM), run the GEODE self-correction loop (Chapter 5), then train. This is what `scripts/build_store.py` calls to produce `data/gms_annual_report_store/`.
- `store_from_triples(md_path, triples, config, ...)` — skip extraction and train directly from a triple list that is already trusted.

Both are GPU/training operations, so the listing is gated behind an environment flag and executed only in CI. The figures it checks are the canonical FY2025 values: segment total revenue `355.0`, FY2025 net income `70.0`.

```
# CI-ONLY (GPU): build the trained store from the annual report.
# Do not run during authoring; the lead executes this in CI.
RUN_GPU = os.environ.get("GMS_RUN_GPU") == "1"
if RUN_GPU:
    from knowlytix.knowledge.geode.rag import build_rag_store
    from knowlytix.knowledge.geode.rag import DocGMSConfig  # config dataclass
    code = (os.path.join(os.path.dirname(os.getcwd()), "code")
            if os.path.basename(os.getcwd()) == "notebooks" else os.getcwd())
    store = build_rag_store(os.path.join(code, "data", "annual_report.md"),
    DocGMSConfig())
    # Numbers come from data/corpus_facts.md (byte-exact ENM).
    assert store.lookup_enm("segment_performance", "Total/All/Revenue") == 355.0
    assert store.lookup_enm("income_statement", "Net Income/FY2025") == 70.0
    print("built store: total revenue =",
          store.lookup_enm("segment_performance", "Total/All/Revenue"))
else:
    print("skipped GPU build (set GMS_RUN_GPU=1 in CI)")
```

Chapter 4 treats ingestion in full; the import names and the ENM keys are the load-bearing detail here.

C.3 The threshold-calibration method

Embedding binding (Chapter 9) resolves a paraphrase such as *topline* to the canonical relation `has_revenue` by cosine similarity. That comparison requires a *threshold*: bind above it, refuse below it. Setting the threshold by inspection is the anti-pattern. The method comprises three steps:

1. **Cohort.** Collect labeled `(term, expected_entity)` *positives* that should bind and *negatives* that should bind to nothing.
2. **Sweep.** Walk a grid of candidate thresholds.
3. **Operating point.** Pick the threshold that best separates the two classes. Tightening the false-allow ceiling (refusing more negatives) pushes the threshold up.

`calibrate_bind_threshold(binder, positives, negatives, grid=...)` runs the sweep, returns `(best_threshold, accuracy)` and sets the value on the binder. This is the same operating-point procedure used for the accept/abstain cuts in Chapter 14 and for the governed agent's gates in the agent book (*Beyond Prompt and Pray*, App C).

GMS in practice — a defensible cut instead of a guessed one

The chunk-and-pray default (Chapter 1) sets a similarity cutoff by inspecting a handful of examples. Here the cutoff is the *argmax* of accuracy over a labeled cohort under a stated false-allow ceiling — a number that can be recorded in a review and reconstructed from data.

The listing below is fully runnable on CPU. It injects a small deterministic encoder and a stub adapter, so that no GPU, no trained store and no Qwen are touched. The vocabulary mirrors the Northwind store's relations. The negatives are placed deliberately close to `has_revenue` so that the sweep must *raise* the threshold to refuse them — this is the false-allow ceiling in action.

```
import numpy as np
from knowlytix.knowledge.rag.binding import TripleBinder
from knowlytix.knowledge.rag.eval import calibrate_bind_threshold

# A deterministic toy encoder: each text maps to a fixed unit vector.
# Synonyms sit near their canonical relation; unrelated terms sit far away.
_VEC = {
    # canonical graph relations (mirror data/corpus_facts.md)
    "has_revenue":   [1.0, 0.0, 0.0],
    "has_headcount": [0.0, 1.0, 0.0],
    "has_region":    [0.0, 0.0, 1.0],
    # positive synonyms -> should bind to has_revenue / has_headcount
    "topline":       [0.97, 0.10, 0.0],
    "sales":         [0.95, 0.05, 0.0],
    "staff count":   [0.05, 0.98, 0.0],
    # negatives -> should bind to nothing (placed deliberately close to
    # has_revenue so the sweep has to RAISE the threshold to refuse them)
    "weather":       [0.60, 0.55, 0.0],
    "breakfast":     [0.62, 0.50, 0.0],
}

def toy_encoder(texts):
```

```
    out = []
    for t in texts:
        v = np.asarray(_VEC.get(t.strip().lower(), [0.33, 0.33, 0.33]),
                       dtype=np.float32)
        out.append(v / (np.linalg.norm(v) + 1e-9))
    return np.vstack(out)

class _StubAdapter:  # the binder only needs the relation/entity vocab
    relation_to_idx = {"has_revenue": 0, "has_headcount": 1, "has_region": 2}
    entity_to_idx = {"has_revenue": 0, "has_headcount": 1, "has_region": 2}

class _StubStore:
    adapter = _StubAdapter()
    def fuzzy_match_entity(self, name):
        # force the embedding path: no fuzzy hit for paraphrases
        return name if name in _StubAdapter.entity_to_idx else None

binder = TripleBinder(_StubStore(), mode="embedding", encoder=toy_encoder,
                      bind_threshold=0.5, bind_margin=0.05)
print("binder mode:", binder.mode, "| start threshold:", binder.bind_threshold)
```

```
# The labeled cohort: terms that SHOULD bind, and terms that should NOT.
positives = [
    ("topline", "has_revenue"),
    ("sales", "has_revenue"),
    ("staff count", "has_headcount"),
]
negatives = ["weather", "breakfast"]

best_th, acc = calibrate_bind_threshold(binder, positives, negatives)
print(f"chosen operating point: threshold={best_th:.2f}  accuracy={acc:.2f}")
print("binder threshold now set to:", binder.bind_threshold)
```

Output (deterministic on CPU):

```
chosen operating point: threshold=0.80  accuracy=1.00
binder threshold now set to: 0.8
```

The negatives sit at cosine ≈ 0.74–0.78 from `has_revenue`, so a lax threshold of 0.50 would *false-allow* them. The sweep raises the cut to 0.80 — above the negatives, below the synonyms (≈ 0.99) — which binds every synonym and refuses every negative (accuracy 1.00). On the real Northwind store the encoder is MiniLM and the cohort comes from `data/eval_cohort.json`, but the method is identical. The same operating-point procedure governs the accept/abstain decision examined in Chapter 13.

C.4 Limits

Calibration tunes a decision boundary; it does not create separability. If the positives and negatives overlap in embedding space, no threshold recovers 100% accuracy — the sweep returns the *least-bad* point, not a correct one. A calibrated threshold is only as reliable as its cohort: calibration must use hand-labeled pairs, never the system's own accepted outputs, since the latter encodes the current set of mistakes. Calibration also says nothing about whether a *fact* is true — that determination

belongs to the verifier (Chapter 12), not to the binder. Finally, this appendix recaps the substrate's *interface*; the geometric guarantees behind `score_triple` and `check_holonomy` reside in the GMS monograph, not here.

Anti-patterns

- **Threshold by inspection.** A similarity cut chosen from a handful of examples is unreconstructable and drifts silently. Calibrate against a labeled cohort with a stated false-allow ceiling.
- **Calibrating on one's own output.** Fitting the threshold to the answers the system has already accepted encodes its current errors. Use independent, hand-labeled pairs.
- **Treating the store as a black-box vector database.** The six primitives are not nearest-neighbor lookups; `lookup_enm` is byte-exact and `score_triple` and `tension_energy` are geometric. Applying a cosine index where an asserted edge exists discards the substrate's guarantees.

Exercise — move the operating point under a tighter ceiling

Add a third negative, `"forecast"`, with embedding `[0.70, 0.45, 0.0]` (closer still to `has_revenue`). Re-run `calibrate_bind_threshold` and observe whether the chosen threshold rises. Then move `"topline"` toward `[0.78, 0.40, 0.0]` so that it overlaps the negative region and observe that accuracy drops below 1.00 — calibration cannot separate what the encoder has already merged. The worked solution is in the companion notebook.

C.5 Self-check

Re-running the calibration must reproduce the same operating point, and that point must perfectly separate the labeled cohort.

```
# Reproduce the operating point and prove it separates the cohort.
binder.bind_threshold = best_th
from knowlytix.knowledge.rag.query_triples import QueryTriple

pos_ok = all(binder.bind(QueryTriple(t, "_", "?")).head == exp
             for t, exp in positives)
neg_ok = all(binder.bind(QueryTriple(t, "_", "?")).head is None
             for t in negatives)

assert acc == 1.0, f"cohort not separable at acc={acc}"
assert pos_ok, "a positive synonym failed to bind at the chosen threshold"
assert neg_ok, "a negative term bound when it should have been refused"
print("OK: operating point", round(best_th, 2),
      "separates the labeled cohort (acc=1.00)")
```

This is the appendix's claim made executable: the calibration method is reproducible and the chosen operating point is defensible against its cohort. For the substrate's role inside the full pipeline, see Chapter 2; for calibration applied to the live accept/abstain decision, see Chapter 14; and for the binding step that consumes the calibrated threshold, see Chapter 9.

Appendix D

Reproducing the Artifacts

Every number printed in Chapters 1–18 derives from one trained store over one corpus, and a reader must be able to rebuild that store from a fresh clone and obtain the same answers. This appendix documents the reproduction pipeline: the artifact tree, the single GPU step that builds the store, the notebook builders that keep every listing byte-identical to the cell it documents and a CPU-only verification that the *shipped* store reproduces the corpus facts exactly. The aim is a one-command path from clone to a runnable book, with an inexpensive check that a reader can run before spending a GPU minute.

This appendix has no dependency on the chapters' content—it depends only on the repository skeleton (Foundation F1) and the corpus and store (Foundation F2). It is the operational counterpart to Appendix C: that appendix makes the *method* self-contained; this one makes the *artifacts* self-contained. The geometric store whose reproduction we verify here is the substrate introduced in Chapter 2 and elaborated through Chapter 4.

D.1 The artifact tree

As in every chapter, the notebook's first cell makes `knowlytix` resolve to the branch source under `KNOWLYTIX_SRC`.

```
import os, sys
KNOWLYTIX_SRC = os.environ.get("KNOWLYTIX_SRC", "/home/asudjianto/jupyterlab/GMS-
    knowlytix")
sys.path.insert(0, KNOWLYTIX_SRC)
```

Everything needed to rebuild the book is checked in. There are two kinds of files in `data/`: hand-authored inputs (`annual_report.md`, `eval_cohort.json`) and generated outputs (`corpus_facts.md` and the trained store). The generated outputs are committed so the book is runnable from a clone without a GPU; `build_store.py` regenerates them.

```
# The reproducible artifact tree. Everything a reader needs to rebuild the book
# is checked in; nothing here is generated by hand.
#
#   beyond-chunk-and-pray/
#     book/                          # the LaTeX manuscript
#     notebooks/                     # one runnable .ipynb per chapter
#     code/                          # REPO_ROOT below: all code + data live here
#       pyproject.toml               # deps: knowlytix, torch, transformers, pandas,
    jupyter
```

```
#       book_kit.py                  # shared bootstrap: repo paths + store loader
#       scripts/
#         _bootstrap.py              # makes 'knowlytix' resolve to the branch source
#         build_store.py             # corpus  -> trained store + data/corpus_facts.md
#         build_nb_*.py              # one builder per chapter -> notebooks/*.ipynb
#         build_nb_appendix_D.py     # this appendix's builder
#       data/
#         annual_report.md           # the corpus (source of truth for content)
#         corpus_facts.md            # ground-truth facts sheet (generated)
#         eval_cohort.json           # labeled evaluation questions
#         gms_annual_report_store/   # the trained GMS store (generated)
#
# Two kinds of files live in code/data/: hand-authored inputs (annual_report.md,
# eval_cohort.json) and generated outputs (corpus_facts.md, the store). Generated
# outputs are committed so the book is runnable from a fresh clone without a GPU;
# build_store.py regenerates them byte-for-byte. The notebooks run from notebooks/,
# so REPO_ROOT resolves to the sibling code/ directory.
import os

REPO_ROOT = os.path.join(os.path.dirname(os.getcwd()), "code") if os.path.basename(os.
    getcwd()) == "notebooks" else os.getcwd()
for sub in ("data/annual_report.md", "data/eval_cohort.json",
            "data/gms_annual_report_store", "scripts/build_store.py"):
    path = os.path.join(REPO_ROOT, sub)
    print(f"{'OK ' if os.path.exists(path) else 'MISSING':8} {sub}")
```

The trained store is itself a directory of small artifacts—a model checkpoint, an adapter, the ENM table and the source markdown. It is portable: copying the directory loads it anywhere, with no retraining.

```
# The trained store is a directory of small artifacts -- a model checkpoint, an
# adapter, the ENM table, and the source markdown. It is portable: copy the
# directory and the store loads anywhere, no retraining. This is what build_store.py
# writes and what every chapter reloads.
STORE = os.path.join(REPO_ROOT, "data", "gms_annual_report_store")
for name in sorted(os.listdir(STORE)):
    print(name)
```

```
adapter.json
documents
enm.json
metadata.json
model.pt
model_dims.json
phase.json
triples.json
```

D.2 The one GPU step: `build_store.py`

`scripts/build_store.py` is the single command that turns the corpus into the trained store, and the only GPU step in the whole book. Its configuration is the reproducible recipe: a fixed geometry,

a fixed training budget and `regex` ingest—a deterministic extraction path with no LLM in the loop, so the same corpus yields the same triples every time. The build returns a `RagBuildResult` (`converged`, `iterations`, `n_entities`, `n_triples`, `n_enm`, `corrections`, `anchor_violations`) and then regenerates `data/corpus_facts.md` from the trained store. The self-correction loop that runs inside this build is described in Chapter 5.

```
# scripts/build_store.py is the single command that turns the corpus into the
# trained store. It is the only GPU step in the whole book. The configuration
# below is copied verbatim from that script -- it is the reproducible recipe:
# a fixed geometry, a fixed training budget, and regex ingest (deterministic,
# no LLM in the extraction path). Running it twice produces the same store.
#
#   from knowlytix.core.config import GeometryConfig, TrainConfig
#   from knowlytix.knowledge.config import DocGMSConfig
#   from knowlytix.knowledge.geode.rag import build_rag_store
#   from knowlytix.knowledge.geode.loop import make_default_trainer
#
#   cfg = DocGMSConfig(
#       store_path="data/gms_annual_report_store",
#       ingest_mode="regex",                 # deterministic extraction
#       loss_mode="cap",
#       geometry=GeometryConfig(d_v=64, d_u=64, m=32, d=32),
#       train=TrainConfig(epochs=150, batch_size=64, neg_samples=16,
#                         lr=5e-3, lr_riemannian=2e-3),
#   )
#   res = build_rag_store("data/annual_report.md", cfg, device=dev,
#                         geode_trainer=make_default_trainer(dev, epochs=80))
#
# res is a RagBuildResult with: converged, iterations, n_entities, n_triples,
# n_enm, corrections, anchor_violations. build_store.py prints those, then
# regenerates data/corpus_facts.md from the trained store. DO NOT run this cell
# in a shared-GPU session; the lead runs `python scripts/build_store.py` in CI.
print("build_store.py is the GPU step; see scripts/build_store.py. Skipped here.")
```

Note. *T*

he notebook does *not* run this cell. A single GPU is shared during authoring; the build is a CI step. A reader reproducing locally runs `python scripts/build_store.py` once, then never needs the GPU again—the remaining chapters load the store on CPU.

D.3 Notebooks are built, not hand-edited

Notebooks are emitted by `scripts/build_nb_*.py` builders that use `nbformat`, so the JSON is always valid and the first cell is always the exact `KNOWLYTIX_SRC` bootstrap from the global brief. Re-running a builder regenerates its notebook deterministically; that is the mechanism that keeps each `.tex` listing byte-identical to the cell it documents. This appendix's own notebook is built the same way, by `scripts/build_nb_appendix_D.py`.

```
# Notebooks are *built*, not edited by hand. Each scripts/build_nb_*.py emits one
# notebook with nbformat so the JSON is always valid and the first cell is always
# the exact KNOWLYTIX_SRC bootstrap. Re-running a builder regenerates its notebook
```

```
# deterministically; that is how a .tex listing stays byte-identical to the cell
# it documents. List the builders and the notebooks they own.
SCRIPTS = os.path.join(REPO_ROOT, "scripts")
builders = sorted(f for f in os.listdir(SCRIPTS)
                  if f.startswith(("build_nb", "build_notebook")))
print(f"{len(builders)} notebook builders:")
for b in builders:
    print(" ", b)
```

D.4 Verifying the reproduction (CPU only)

The reproducibility claim is concrete: a fresh clone's shipped store reproduces the corpus facts byte-exact. We load the store on CPU—no GPU, no Qwen—and read three Exact Numerical Memory figures back through `lookup_enm`. These are the exact values printed in `data/corpus_facts.md` and used throughout the book; if a rebuild ever drifts, this cell fails first. Exact Numerical Memory is introduced in Chapter 2.

```
# The reproducibility claim: a fresh clone's *shipped* store reproduces the
# corpus facts byte-exact. We load the store on CPU (cheap -- no GPU, no Qwen)
# and read three ENM figures back through lookup_enm. These are the exact values
# in data/corpus_facts.md; if a rebuild ever drifts, this cell fails first.
import torch
from knowlytix.knowledge.config import DocGMSConfig
from knowlytix.knowledge.store import GMSExpertStore

store = GMSExpertStore(DocGMSConfig(store_path=STORE),
                       device=torch.device("cpu"))
assert store.load(), f"no trained store at {STORE}; run scripts/build_store.py first"

# (category, entity_id, expected) -- verbatim from data/corpus_facts.md.
EXPECTED_ENM = [
    ("income_statement", "Revenue/FY2025", 355.0),
    ("segment_performance", "Cloud Platform/Technology/Revenue", 120.0),
    ("balance_sheet", "Total Assets", 540.0),
]
for category, entity_id, expected in EXPECTED_ENM:
    got = store.lookup_enm(category, entity_id)
    print(f"{category}/{entity_id} = {got}  (expected {expected})")
    assert got == expected, f"ENM drift: {category}/{entity_id} {got} != {expected}"
print("\nstore reproduces the shipped corpus facts byte-exact")
```

```
income_statement/Revenue/FY2025 = 355.0  (expected 355.0)
segment_performance/Cloud Platform/Technology/Revenue = 120.0  (expected 120.0)
balance_sheet/Total Assets = 540.0  (expected 540.0)

store reproduces the shipped corpus facts byte-exact
```

A single matching value could be coincidence. A stronger check is an invariant that holds across values: the four segment revenues must sum to the `Total` row. This is the numeric *sum anchor* that GEODE enforced during the build (Chapter 5); checking it here establishes that the shipped store is internally consistent, not merely that one cell happens to match.

```
# A second reproduction invariant, independent of any single value: the segment
# revenues must sum to the Total row. This is the numeric *sum anchor* GEODE
# enforces during the build (Chapter~\ref{ch:geode-self-correction}); checking it here
    proves the shipped
# store is internally consistent, not just that one cell happens to match.
SEGMENTS = ["Cloud Platform/Technology", "Devices/Technology",
            "Logistics/Operations", "Retail/Operations"]
parts = [store.lookup_enm("segment_performance", f"{s}/Revenue") for s in SEGMENTS]
total = store.lookup_enm("segment_performance", "Total/All/Revenue")
print("segments:", parts, " sum:", sum(parts), " total:", total)
assert sum(parts) == total, f"sum anchor broken: {sum(parts)} != {total}"
print("sum anchor holds: Total = sum(segments)")
```

```
segments: [120.0, 80.0, 95.0, 60.0]  sum: 355.0  total: 355.0
sum anchor holds: Total = sum(segments)
```

D.5 From clone to runnable book

The whole path is two commands. The first rebuilds the store (GPU, once); the second executes every notebook end to end, each of which concludes on a self-check `assert`.

```
git clone <repo> && cd beyond-chunk-and-pray/code
pip install -e .                          # knowlytix, torch, transformers, jupyter
python scripts/build_store.py             # GPU, once: store + corpus_facts.md
jupyter nbconvert --execute --to notebook --inplace ../notebooks/*.ipynb
```

Continuous integration runs exactly this. The workflow below installs the package, rebuilds the store, then executes every notebook; `nbconvert` exits non-zero if any cell—including a chapter's final self-check—raises, so a drifted figure or a broken invariant fails the build. The chapter self-checks that this workflow enforces are described in Chapter 12.

```
# .github/workflows/ci.yml -- executes every notebook on each push.
name: ci
on: [push, pull_request]
jobs:
  notebooks:
    runs-on: [self-hosted, gpu]        # build_store.py needs a GPU once
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with: { python-version: "3.12" }
      - run: pip install -e .
      - run: python scripts/build_store.py
      - run: jupyter nbconvert --execute --to notebook --inplace notebooks/*.ipynb
```

GMS in practice — a build that can be re-run, not a snapshot that must be trusted

Chunk-and-pray pipelines (Chapter 1) are reproducible only in the weak sense that re-running them re-embeds the same chunks. There is no invariant to check: a silent change

in the embedding model, the chunker or a parsed number leaves no signal. The geometric store ships an explicit recipe (`build_store.py`) *and* explicit invariants (byte-exact ENM, the sum anchor) that a CPU-only cell re-verifies. Reproduction is then a test that passes or fails, not a claim accepted on faith.

D.6 Limitations

This appendix verifies that the *shipped* store matches the shipped corpus facts; it does not establish that an independent rebuild on different hardware yields a bit-identical `model.pt`. Floating-point training is sensitive to BLAS versions, CUDA and thread counts, so checkpoints can differ at the bit level while the *facts* they encode remain invariant. Reproduction is therefore checked at the level that matters—ENM values and the sum anchor read back through the public API—rather than by hashing the checkpoint. The verification is also only as complete as the invariants encoded: it covers the numeric facts and one structural anchor, not every triple. Extending it to the full triple set and to the multi-hop retrievals of Chapter 10 is the exercise below. Finally, the GPU rebuild itself is out of scope for the CPU check; CI runs it, but a reader on a laptop verifies the shipped artifacts without reproducing the training run.

Anti-patterns

- **Committing the store but not the recipe.** A checked-in artifact with no script to regenerate it is a snapshot, not a reproducible build. Ship `build_store.py` alongside the store so the artifact can always be rebuilt and audited.
- **Hand-editing generated notebooks.** Editing a `.ipynb` directly desynchronizes it from its `.tex` listing and risks invalid JSON. Change the `build_nb_*.py` builder and regenerate.
- **Hashing the checkpoint as the reproducibility test.** A bit-exact `model.pt` is hardware-dependent and fragile. Verify the *facts*—byte-exact ENM values and structural anchors read through the public API—which are what the book actually depends on.

Exercise — Extend the reproduction check

The notebook verifies three ENM values and the revenue sum anchor. Strengthen it: (1) add the headcount sum anchor (segment headcounts must sum to the `Total/All/Headcount` value of `1500.0`); (2) reproduce the two-hop retrieval from `data/corpus_facts.md`—bind `("cloud platform", "has_division", "?x")` then `("?x", "has_region", "?")` with a `TripleBinder` and `Retriever`, and assert the answer is `"north america"`. Both run on CPU. The worked solution is in the appendix notebook's exercise cell. (See Chapter 10 for the multi-hop retrieval API.)

The worked solution, from the appendix notebook's exercise cell:

```
# Exercise (worked): strengthen the reproduction check with a second sum anchor
# and a two-hop retrieval -- both CPU-only, both grounded in data/corpus_facts.md.
# (1) headcounts must sum to the Total row.
head_parts = [store.lookup_enm("segment_performance", f"{s}/Headcount")
```

```
             for s in SEGMENTS]
head_total = store.lookup_enm("segment_performance", "Total/All/Headcount")
assert sum(head_parts) == head_total == 1500.0
print("headcount anchor holds:", sum(head_parts), "==", head_total)

# (2) reproduce the two-hop retrieval: cloud platform -> division -> region.
from knowlytix.knowledge.geode.provenance import ProvenanceLedger
from knowlytix.knowledge.rag import Retriever, TripleBinder
from knowlytix.knowledge.rag.query_triples import QueryTriple

ledger = ProvenanceLedger.from_text(store.markdown)
binder = TripleBinder(store)
retriever = Retriever(store, ledger)
chain = [binder.bind(QueryTriple("cloud platform", "has_division", "?x")),
         binder.bind(QueryTriple("?x", "has_region", "?"))]
result = retriever.retrieve(chain)
print("two-hop answers:", result.answers)
assert any(a[0] == "north america" for a in result.answers)
print("two-hop retrieval reproduces: north america")
```

The self-check that closes the appendix notebook asserts that the artifact tree is present, the store loads, every documented ENM value reads back exactly and the sum anchor holds—the same gates that CI enforces. The capstone in Chapter 18 exercises the same store end to end.

```
# Self-check: the chapter's claim is that a fresh clone reproduces the artifacts.
# We assert the artifact tree is present, the store loads, every documented ENM
# value reads back exactly, and the sum anchor holds -- the same gates CI runs.
assert os.path.exists(STORE), "store directory missing"
assert store.lookup_enm("income_statement", "Revenue/FY2025") == 355.0
assert sum(store.lookup_enm("segment_performance", f"{s}/Revenue")
           for s in SEGMENTS) == store.lookup_enm("segment_performance",
                                                  "Total/All/Revenue")
print("App D self-check passed: artifacts reproduce byte-exact from a clone.")
```

Appendix E

A Runnable "Chunk and Pray" Baseline

Chapter 1 argues against chunk-and-pray retrieval and Appendix B contrasts the two *contracts*. Neither runs the baseline. This appendix does: it builds a textbook dense-retrieval RAG end to end — fixed-size chunks, sentence-transformers embeddings, cosine top-k, stuff-and-generate — and grades it on the *same* labeled cohort and with the *same* trust metrics as the GEODE-RAG pipeline of Chapter 15. The comparison is then a table of numbers from one run on one corpus rather than a rhetorical claim.

The baseline is a yardstick: its generator is the same local Qwen2.5-3B-Instruct the GEODE-RAG pipeline uses, so the variable under test is the retrieval contract, not the model. Where chunk-and-pray answers correctly it is credited; where it fails the failure is the architecture's, which is what this appendix measures. Unlike the rest of the book, this appendix deliberately uses *none* of the `knowlytix.knowledge.rag` machinery for retrieval: it is hand-rolled from an embedding model and a cosine search, because that is what "chunk and pray" is.

E.1 The corpus and the cohort, unchanged

The first cell is the usual `KNOWLYTIX_SRC` bootstrap; it is needed only to load the shared local-Qwen backend, not for retrieval. The baseline then reads the same `annual_report.md` and the same `eval_cohort.json` the GEODE-RAG pipeline is graded on. There is no trained store — chunk-and-pray needs only raw text and an embedding model.

```
import os, sys
KNOWLYTIX_SRC = os.environ.get("KNOWLYTIX_SRC", "/home/asudjianto/jupyterlab/GMS-
    knowlytix")
sys.path.insert(0, KNOWLYTIX_SRC)
REPO_ROOT = os.path.join(os.path.dirname(os.getcwd()), "code") if os.path.basename(os.
    getcwd()) == "notebooks" else os.getcwd()
```

```
# The baseline reads the SAME corpus and the SAME labeled cohort the GEODE-RAG
# pipeline is graded on (Chapter~\ref{ch:abstention-coverage}). Nothing about the data
    changes; only the
# retrieval contract does. There is no trained store here -- chunk-and-pray needs
# only raw text and an embedding model.
import json
```

```
CORPUS = os.path.join(REPO_ROOT, "data", "annual_report.md")
COHORT = os.path.join(REPO_ROOT, "data", "eval_cohort.json")

report_md = open(CORPUS, encoding="utf-8").read()
cohort = json.loads(open(COHORT, encoding="utf-8").read())
print(f"corpus: {len(report_md)} chars, {report_md.count(chr(10)) + 1} lines")
print(f"cohort: {len(cohort)} cases; types:",
      sorted({r["type"] for r in cohort}))
```

The corpus is `3260 chars, 90 lines`; the cohort is the same twelve cases, spanning the same six types, used in Chapter 15.

E.2 Step 1 — chunk the document, blind to structure

The canonical splitter slides a fixed window of words over the document with a fixed overlap. It is deliberately blind to structure: it does not know a markdown table row from a sentence, so it cuts across both. This is the defining move of chunk-and-pray — the retrieval unit becomes a span of text, not an asserted fact. The `Chunk.location` property records a character-offset range, which is the best provenance a chunker can offer and is already weaker than the `file:line:char` cell span of Chapter 3.

```
# Step 1 -- chunk and pray. The canonical splitter: slide a fixed-size window of
# WORDS over the document with a fixed OVERLAP. This is deliberately blind to
# structure -- it does not know a markdown table from a paragraph -- so it cuts
# across table rows and section boundaries. That is the whole point: the retrieval
# unit becomes a span of text, not an asserted fact.
from dataclasses import dataclass

CHUNK_WORDS = 60          # window size, in whitespace tokens
OVERLAP_WORDS = 15        # stride = CHUNK_WORDS - OVERLAP_WORDS


@dataclass
class Chunk:
    id: int
    text: str
    char_start: int
    char_end: int

    @property
    def location(self) -> str:
        # Chunk-level provenance: an offset range over the file, NOT a cell.
        # Contrast the GEODE-RAG span ":15:553-558" (file:line:char of one cell).
        return f"chunk#{self.id}:{self.char_start}-{self.char_end}"


def chunk_text(text: str, size: int = CHUNK_WORDS,
               overlap: int = OVERLAP_WORDS) -> list[Chunk]:
    # Tokenize on whitespace, keeping each token's char offset so a chunk can
    # report *where* in the file it came from -- the best provenance a chunker has.
    toks, offsets, i = [], [], 0
    for tok in text.split(" "):
        offsets.append(i)
```

```
        toks.append(tok)
        i += len(tok) + 1
    chunks, start, cid = [], 0, 0
    stride = max(1, size - overlap)
    while start < len(toks):
        window = toks[start:start + size]
        cstart = offsets[start]
        cend = offsets[min(start + size, len(toks)) - 1] + len(toks[min(start + size,
    len(toks)) - 1])
        chunks.append(Chunk(cid, " ".join(window), cstart, cend))
        cid += 1
        start += stride
    return chunks


chunks = chunk_text(report_md)
print(f"{len(chunks)} chunks of <= {CHUNK_WORDS} words (overlap {OVERLAP_WORDS})")

# Show the structural damage: find the chunk holding the "Total" segment row and
# check whether the four segment rows that must sum to it are in the SAME chunk.
total_chunks = [c for c in chunks if "| Total | All |" in c.text]
seg_chunks = [c for c in chunks if "Cloud Platform" in c.text]
print("Total row in chunk(s):", [c.id for c in total_chunks])
print("Cloud Platform in chunk(s):", [c.id for c in seg_chunks])
print("segment table split across chunks:",
      {c.id for c in total_chunks} != {c.id for c in seg_chunks})
```

The document becomes `12 chunks`. The damage is immediate: the `Total` row lands in chunk 2 while `Cloud Platform` is scattered across chunks 1, 6, 9 and 10, so the split check prints `True`. The sum anchor that GEODE enforces during the build (Chapter 5) — that the four segment revenues equal the `Total` — cannot even be *stated* once the rows it relates live in different retrieval units. Structure that the graph preserves, the chunker destroys.

E.3 Step 2 — embed and index

The encoder is `all-MiniLM-L6-v2`, the textbook chunk-and-pray embedding model: 384-dimensional sentence vectors, roughly 80 MB, no fine-tuning. Every chunk is embedded once — the single heavy offline step — and L2-normalized so that cosine similarity reduces to a dot product. The corpus is small enough that a NumPy matrix with a scikit-learn cosine is exact; FAISS is the drop-in at scale and changes none of the conclusions here.

```
# Step 2 -- embed and index. all-MiniLM-L6-v2 is the textbook chunk-and-pray
# encoder: 384-dim sentence embeddings, ~80MB, no fine-tuning. We embed every
# chunk once (the one heavy offline step) and L2-normalize so cosine similarity
# is a dot product. No FAISS here -- the corpus is tiny, so a numpy matrix and
# scikit-learn cosine are exact; FAISS is the drop-in at scale.
import numpy as np
from sentence_transformers import SentenceTransformer

encoder = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")
chunk_emb = encoder.encode([c.text for c in chunks],
                           normalize_embeddings=True,
                           show_progress_bar=False)
```

```
chunk_emb = np.asarray(chunk_emb, dtype=np.float32)
print("chunk embedding matrix:", chunk_emb.shape)
```

The index is a `(12, 384)` matrix. This is the whole memory of a chunk-and-pray system: an opaque embedding matrix that supports nearest-neighbor lookup and little else (Appendix B).

E.4 Step 3 — top-k retrieval, and the exact-number trap

Retrieval is cosine similarity between the query embedding and every chunk embedding, returning the k nearest. Relevance *is* cosine distance; there is no notion of whether a chunk actually *holds* the asked-for fact. The listing below asks for total revenue and prints the numbers that co-occur in each retrieved chunk.

```
# Step 3 -- top-k retrieval. Cosine similarity between the query embedding and
# every chunk embedding; return the k nearest. Relevance IS cosine distance --
# there is no notion of whether the chunk actually *holds* the asked-for fact.
from sklearn.metrics.pairwise import cosine_similarity

TOP_K = 4


def retrieve(query: str, k: int = TOP_K) -> list[tuple[Chunk, float]]:
    q = encoder.encode([query], normalize_embeddings=True)
    sims = cosine_similarity(np.asarray(q, dtype=np.float32), chunk_emb)[0]
    order = np.argsort(-sims)[:k]
    return [(chunks[i], float(sims[i])) for i in order]


# The exact-number trap: ask for total revenue and look at what surfaces. The top
# chunk is a passage of the report -- it contains 355.0, but also 320.0 (FY2024),
# 200.0, 95.0 ... Several candidate numbers co-occur; nothing marks which is THE
# answer. A chunk is not a cell.
for c, s in retrieve("What was total revenue?"):
    nums = sorted({t for t in c.text.replace("|", " ").split()
                   if t.replace(".", "", 1).isdigit()})
    print(f"  cos={s:.3f} {c.location}  numbers={nums}")
```

The top chunk (`cos=0.543`) carries the numbers `185.0, 200.0, 320.0, 355.0, 82.0, 95.0`. The correct answer 355.0 is present, but so is the prior-year figure 320.0 and four other values, and nothing in the retrieval marks which one is *the* total revenue. This is the exact-number trap: a chunk is not a cell, and similarity ranks the passage, not the figure. Exact Numerical Memory (Chapter 2) exists precisely to read 355.0 from one addressed cell instead of hoping the generator picks it out of six candidates.

E.5 Step 4 — stuff and generate, with no way to decline

The top-k chunks are concatenated into the prompt and the model is asked to answer from them. This is the "pray" step: the system trusts the model to pick the right number and not to invent one. The pipeline has *no abstention path* — `decision` is always `"accept"` — *no* Exact Numerical Memory and *no* verification against structured facts. A deterministic extractive reader stands in

for the model on CPU so the notebook is reproducible without a GPU; the real run uses Qwen and is shown next.

```
# Step 4 -- stuff and generate. Concatenate the top-k chunks into the prompt and
# ask the model to answer from them. This is the "pray" step: we trust the model
# to pick the right number out of the passages and not to invent one. There is
# NO abstention path -- the pipeline always returns an answer -- NO Exact
# Numerical Memory and NO verification of the answer against structured facts.
from dataclasses import dataclass, field


@dataclass
class BaselineAnswer:
    answer: str
    decision: str                       # always "accept" -- chunk-and-pray cannot
    abstain
    sources: list                       # the retrieved Chunks (chunk-level provenance)
    contexts: list = field(default_factory=list)


def make_prompt(question: str, ctx: list[Chunk]) -> str:
    passages = "\n\n".join(f"[{c.id}] {c.text}" for c in ctx)
    return (f"Answer the question using only the passages below.\n\n"
            f"{passages}\n\nQuestion: {question}\nAnswer:")


class BaselineRAG:
    """Chunk-and-pray: retrieve top-k, stuff, generate. No abstention, no ENM,
    no verification. `generate(prompt) -> str` is the only pluggable part."""

    def __init__(self, generate):
        self._generate = generate

    def query(self, question: str, k: int = TOP_K) -> BaselineAnswer:
        hits = retrieve(question, k)
        ctx = [c for c, _ in hits]
        text = self._generate(make_prompt(question, ctx))
        # The pipeline has no branch that returns "I don't know": decision is
        # ALWAYS accept. That is the architectural choice this appendix measures.
        return BaselineAnswer(answer=text.strip(), decision="accept",
                              sources=ctx, contexts=ctx)
```

The real generator is the same local Qwen the GEODE-RAG pipeline uses. Only the generator changes between the CPU stand-in and the run below; the retrieval and the prompt are identical, and the cohort numbers reported in this appendix come from this run.

```
# CI ONLY -- the real chunk-and-pray run on local Qwen2.5-3B-Instruct (GPU).
# Same retrieval, same prompt; only the generator changes. This is the generator
# a real chunk-and-pray deployment ships. The lead runs this in CI; it is NOT
# executed during authoring. The cohort numbers reported in the appendix come
# from this run.
from knowlytix.knowledge.geode import QWEN_3B
from knowlytix.knowledge.llm_backend import LocalTransformersBackend

qwen = LocalTransformersBackend(QWEN_3B, device="cuda")
```

```
def qwen_generate(prompt: str) -> str:
    return qwen.call(system="You are a helpful financial-report assistant. "
                            "Answer concisely from the passages.",
                     user=prompt, max_tokens=64)


qwen_baseline = BaselineRAG(qwen_generate)
print(qwen_baseline.query("What was total revenue?").answer)
```

On total revenue Qwen answers *Total revenue for FY2025 was 355.0 million US dollars* — correct. Chunk-and-pray is not always wrong; the failures appear when the cohort is run in full.

E.6 The cohort, scored exactly as in Chapter 15

The grader is byte-identical to Chapter 15: accept rate, provenance rate, abstention precision and the confident-wrong count. Only the system under test differs. An accepted answer is counted confident-wrong when its expected value is absent, or when it is accepted on a question whose label says abstain — the prose blind spots.

```
# CI -- run the cohort through the REAL (Qwen) baseline and compute the SAME
# trust metrics as Chapter~\ref{ch:abstention-coverage}: provenance rate, abstention
    precision and the
# confident-wrong count, plus the accept rate. The grader is byte-identical to
# Chapter~\ref{ch:abstention-coverage}; only the system under test differs. (The
    extractive `baseline`
# above gives the same architectural verdict on CPU; see the self-check.)
def trust_metrics(rag, rows):
    confident_wrong = 0
    with_prov = answered = 0
    abst_correct = abst_total = 0
    per_case = []
    for r in rows:
        ans = rag.query(r["question"])
        accepted = ans.decision == "accept"
        exp = r.get("expected_answer")
        # case-insensitive substring match (cohort labels are lowercased).
        hit = bool(exp) and exp.lower() in ans.answer.lower()
        if accepted:
            answered += 1
            if any(c.location for c in ans.sources):
                with_prov += 1
        if ans.decision == "abstain":
            abst_total += 1
            if r.get("expect_decision") == "abstain":
                abst_correct += 1
        # confident-wrong: accepted with the expected value missing, OR accepted
        # on a question whose label says abstain (the prose blind spots).
        if accepted and exp is not None and not hit:
            confident_wrong += 1
        if accepted and r.get("expect_decision") == "abstain":
            confident_wrong += 1
```

```
        per_case.append((r["id"], r["type"], ans.decision, hit, ans.answer[:40]))
    n = len(rows)
    return {
        "accept_rate": answered / n,
        "abstain_count": abst_total,
        "provenance_rate": with_prov / answered if answered else 0.0,
        "abstention_precision": abst_correct / abst_total if abst_total else None,
        "confident_wrong": confident_wrong,
    }, per_case


metrics, per_case = trust_metrics(qwen_baseline, cohort)
print("Baseline (chunk-and-pray, Qwen) trust report:")
for k, v in metrics.items():
    print(f"  {k:22} {v}")
print()
for cid, typ, dec, hit, ans in per_case:
    print(f"  {cid:16} {typ:13} {dec:8} hit={hit!s:5} {ans!r}")
```

The report reads `accept_rate 1.0`, `abstain_count 0`, `provenance_rate 1.0`, `abstention_precision None` and `confident_wrong 6`. The baseline accepts every one of the twelve questions and is confidently wrong on half of them. The per-case breakdown locates the failures, and each is a textbook chunk-and-pray pathology:

- **`q-seg-rev` and `q-adversarial` (Cloud Platform revenue, expected** 120.0**).** Both miss. Top-k surfaces the Management Discussion prose — *the Cloud Platform segment continued to lead the company's topline* — which is lexically the closest passage to the question yet holds no figure, so Qwen answers about "sustained demand" and never reaches the table cell. Similarity to the words is not relevance to the fact.

- **`q-multihop-head` (head of the division containing Logistics, expected *Sam Reyes*).** The baseline answers *Dana Cole*. The hop — Logistics → Operations → Sam Reyes — crosses two table rows that the chunker placed in different chunks, so no single retrieved span carries the chain. The variable-environment triple chain of Chapter 8 is exactly what this lacks.

- **`q-retail-head` (Retail headcount, expected** 520.0**).** Qwen answers *520 people work in Retail* — right in substance, but the byte-exact string 520.0 that the grader (and Exact Numerical Memory) requires is absent. Without ENM a number is whatever the model chose to print, and a downstream exact-match consumer rejects it.

- **`q-outlook` and `q-risks` (prose, labeled abstain).** Sections 6–8 of the report carry no triples (coverage ratio 0.56; Chapter 13). GEODE-RAG abstains on them; chunk-and-pray has no abstention path, so it fluently answers both — two confident-wrong answers the architecture cannot avoid.

The six that succeed — total revenue, net income, shareholders equity, CEO, the one-hop region question and the paraphrase — are the cases where a single chunk happens to contain the answer and the model copies it. That is the regime where a vector-RAG framework is the right tool (Appendix B); it is not the regime a financial report lives in.

E.7 Provenance exists, but it is not verifiable

Every accepted answer carries a source, so `provenance_rate` is 1.0 — the same number GEODE-RAG reports. The figure is misleading on its own. The baseline's source is a 60-word window, not a cell, and nothing ties the answer's number to an authoritative figure within it.

```
# The chunk-level provenance is real but not verifiable. Every accepted answer
# carries a source (provenance_rate = 1.0), yet the source is a 60-word window,
# not a cell, and nothing ties the answer's number to an authoritative figure.
# Compare the two provenance shapes side by side.
demo = qwen_baseline.query("What is Cloud Platform revenue?")
print("baseline source :", demo.sources[0].location,
      "  (a", CHUNK_WORDS, "-word window spanning many cells)")
print("GEODE-RAG source: data/annual_report.md:15:553-558",
      "  (the exact cell 120.0 resolves to)")

# The two prose blind spots (Outlook, Risk Factors) carry no triples; GEODE-RAG
# abstains on them (Chapter~\ref{ch:abstention-coverage}, abstention_precision = 1.0).
    Chunk-and-pray has
# no abstention path, so it answers anyway -- a confident-wrong on each.
prose = [r for r in cohort if r["type"] == "unanswerable"]
for r in prose:
    a = qwen_baseline.query(r["question"])
    print(f"{r['id']}: decision={a.decision!r} (GEODE-RAG abstains) answer={a.answer
    [:48]!r}")
```

The baseline cites `chunk#6:1524-1988` — a span covering many cells — while GEODE-RAG cites `data/annual_report.md:15:553-558`, the exact characters 120.0 resolves to (Chapter 3). A `provenance_rate` of 1.0 means "every answer points somewhere"; only the cell span means "every answer points at the figure it asserts." The two prose questions both return `decision='accept'` with a fluent paragraph, where GEODE-RAG abstains.

Metric (same grader, same cohort)	**Chunk-and-pray (Qwen)**	**GEODE-RAG (Ch. 15)**
Accept rate	1.0 (answers all 12)	operating-point dependent
Abstentions	0	2 (the prose blind spots)
Abstention precision	undefined (never abstains)	1.0
Provenance rate	1.0 (chunk-level)	1.0 (cell-level span)
Provenance granularity	60-word window	`file:line:char` cell
Exact numbers	parsed from prose	ENM, byte-exact
Confident-wrong	**6** of 12	**0**

Table E.1: The same twelve-case cohort and the same trust grader, two retrieval contracts. The release gate is the confident-wrong count.

E.8 Self-check

The appendix's claim is *architectural*: it holds for any generator, so the self-check runs on CPU with the deterministic extractive reader and asserts the three properties that no choice of model can repair. Chunk-and-pray never abstains; its provenance is a chunk offset, never a `file:line:char` cell span; and it is confidently wrong on at least the two prose blind spots GEODE-RAG correctly abstains on.

```
# Self-check (CPU, deterministic): the appendix's claim is architectural, so it
# holds for ANY generator. (1) chunk-and-pray never abstains; (2) provenance is
# chunk-level, never a file:line:char cell span; (3) it is confidently wrong on
# every prose blind spot the corpus never triplified -- at least the two
# unanswerable cases GEODE-RAG correctly abstains on.
metrics, _ = trust_metrics(baseline, cohort)

# (1) no abstention path: every case is accepted.
assert metrics["accept_rate"] == 1.0, metrics
assert metrics["abstain_count"] == 0, metrics

# (2) provenance exists but is chunk-level, not a cell span.
loc = baseline.query("What was total revenue?").sources[0].location
assert loc.startswith("chunk#") and ":" in loc
assert "annual_report.md:" not in loc       # never a file:line:char cell span

# (3) confident-wrong on the prose blind spots GEODE-RAG abstains on.
n_unanswerable = sum(1 for r in cohort if r["type"] == "unanswerable")
assert metrics["confident_wrong"] >= n_unanswerable, metrics

print("App E self-check passed: chunk-and-pray never abstains, gives only "
      "chunk-level provenance, and is confidently wrong on prose blind spots.")
print("confident_wrong =", metrics["confident_wrong"],
      " accept_rate =", metrics["accept_rate"])
```

GMS in practice — the same cohort, two verdicts

Run on one corpus with one grader, chunk-and-pray accepts all twelve questions and is confidently wrong on six; GEODE-RAG (Chapter 15) abstains on the two it cannot ground and reports zero confident-wrong. The difference is the retrieval contract, not the generator: both run the same local Qwen. Chunk-and-pray ranks passages by cosine similarity and trusts the model to read the answer off the text; GEODE-RAG answers through bound triples, reads exact numbers from ENM, cites the cell span and declines when nothing binds. Table E.1 is the empirical form of the argument Chapter 1 makes in prose.

Anti-patterns

- **Reading the headline accept rate as a win.** Chunk-and-pray's 1.0 accept rate means it never declines, not that it is always right; six of those accepts are confident-wrong. An accept rate without a confident-wrong count flatters the system that guesses most.
- **Treating `provenance_rate = 1.0` as verifiability.** Both systems score 1.0, but only

the cell span ties the answer to the figure it asserts. A chunk-level citation answers "which passage," not "which cell," and cannot be replayed as proof.

- **Grading on retrieval recall instead of trust.** Had this appendix scored "did the right chunk appear in the top-k," chunk-and-pray would look strong — the right passage is usually retrieved. The cohort grades grounding, exactness and abstention, where the architecture fails (Appendix B).

Exercise — move a failure into the win column — and find the next one

Take `q-seg-rev` (Cloud Platform revenue), which the baseline misses because the MD&A prose out-ranks the segment table. (1) Re-chunk the document by markdown section instead of by fixed window, re-embed and confirm the segment table now out-ranks the prose for that question. (2) Observe that the multi-hop case `q-multihop-head` still fails — section chunking keeps a table together but does not create the Logistics → Operations → Sam Reyes chain — and that the two prose questions still cannot abstain. The deliverable is the recognition that better chunking moves individual cases without changing the contract: confident-wrong stays nonzero. There is no notebook solution; the point is the ceiling, not the tweak.

Limits and cross-references. This is one baseline on one corpus, not a tuned vector pipeline on a benchmark; a reranker, query rewriting or section-aware chunking would move individual cases, and Appendix B discusses what "with effort" can and cannot achieve. None of those changes adds an abstention path, exact-number memory or claim-level verification, so none drives confident-wrong to zero — that requires the triple-mediated contract, not a better chunker. The trust metrics are defined in Chapter 15; the abstention and coverage behavior the baseline lacks is Chapter 13; the verifier that keeps GEODE-RAG's confident-wrong at zero is Chapter 12; and the distrusted dense fallback — the one place GEODE-RAG admits a dense index, quarantined and flagged — is Chapter 16.

About the Authors

Agus Sudjianto

Agus Sudjianto is the Chief Scientist at KnowlytiX. He has spent more than two decades building, governing and validating quantitative models inside major financial institutions. He was Executive Vice President and Head of Model Risk at Wells Fargo, where he served on the Management Committee and led enterprise model risk management. Earlier in his career he held senior quantitative risk roles at Lloyds Banking Group and Bank of America. Since leaving corporate industry, he has continued this work as an advisor, builder and researcher across banking, fintech and AI.

Agus's work sits at the intersection of machine learning, model risk and governed AI systems. He created PiML and MoDeVa, toolkits for interpretable model development and validation, and his more recent work extends that same discipline into agentic AI, graph-grounded retrieval and geometric memory. Across these projects, the through-line is consistent: high-stakes AI should be built with the same rigor expected of high-stakes statistical models.

He is also co-author of *Design and Modeling for Computer Experiments*, holds several U.S. patents and has long worked across engineering, quantitative finance and applied machine learning. His current research centers on learning as geometry discovery in both predictive machine learning and generative AI.

In this series, Agus brings the perspective of someone who has spent a career asking not only whether a model works, but whether it can be governed, defended and trusted in practice.

Wing Yan Lau

Wing Yan Lau is the Chief Technology Officer at KnowlytiX. Her work centers on the systems layer that makes GMS usable in practice: document ingestion, knowledge-store construction, query infrastructure, verification pathways and the interfaces that connect governed AI to real enterprise data. She is a co-author of KnowlytiX's research on graph-verified evaluation and structured financial-document retrieval, including work reflected in FinStructBench and in the company's broader knowledge and testing stack.

Wing brings more than two decades of database and data-platform engineering experience to that work. She has contributed to core systems at IBM, SAP and Workday, with technical work spanning query optimization, storage systems and execution infrastructure. That background is visible throughout the KnowlytiX platform, where the challenge is not only to generate answers, but to connect models to structured knowledge in ways that remain exact, inspectable and operationally reliable.

In this series, Wing brings implementation discipline to every layer of the system: how documents become structured stores, how numeric facts remain exact, how graph-backed retrieval is made usable and how governed workflows are turned into code rather than left as intentions in prose. Her contribution is what turns the ideas in the architecture into systems an engineer can actually build, test and run.

About KnowlytiX

KnowlytiX builds governed infrastructure for agentic AI in regulated industries. Its focus is not on making models sound safer in a demo. Its focus is on making AI systems inspectable, reproducible and defensible when the output must stand up to internal review, model validation, compliance testing or regulatory scrutiny.

The company's premise is straightforward: in high-stakes settings, AI compliance cannot rest on probabilities alone. Retrieval-augmented generation can help a model find relevant source material. Guardrails can filter unsafe or disallowed content. But neither one can show that an answer actually complies with domain-specific rules, that a tool call should have been allowed or that a chain of reasoning remained consistent from start to finish. KnowlytiX builds the missing layer around the model: typed workflows, governed tool access, structured verification, benchmark-driven testing and audit trails that turn an AI run into something a validator or regulator can actually examine.

At the center of that work is the Geometric Memory System, or GMS. GMS is not a traditional knowledge graph, and it is not just a standard KG embedding model with new terminology. A traditional knowledge graph stores symbolic triples and supports logical or database-style queries. A classical KG embedding model learns scores for ranking or link prediction. GMS is built for something more demanding: verification. It treats geometry as an operational substrate for checking whether a claim is plausible, whether two claims are contradictory, whether a multi-step reasoning path is compositionally consistent and whether a numeric fact is exactly correct.

In GMS, entities are represented in geometric spaces that separate semantic similarity from logical polarity. Relations act as learned geometric operators on the unit hypersphere. The system adds capabilities that ordinary knowledge graphs do not provide in a unified way: contradiction detection through tension energy, path-consistency checks through holonomy, geometric handling of numeric constraints and Exact Numerical Memory for lossless values with integrity checks. That is what makes GMS more than a storage layer. It becomes a memory, retrieval and verification substrate that can support governed agents, structured document understanding and benchmark construction from the same underlying system.