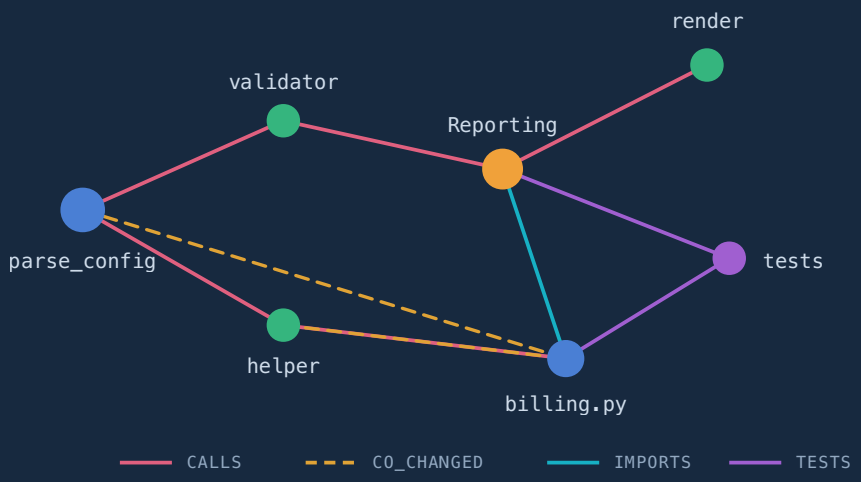
A FIELD GUIDE TO GRAPH-POWERED AI

The Impact Map

How one small repository teaches knowledge graphs, vector search, honest evaluation, and agents that have to earn their keep: a complete, code-level tour of **LabGo**, the Change Impact Analyst.



Written for engineers coming back to code and getting into AI · Built from the LabGo repository and its decision log · 2026

About This Book

This is a guided, end-to-end tour of a single real repository: **LabGo**, a small system that answers one question about a codebase. The question is *"if I change this file, what else breaks, which tests must run, and who should review it?"* By the final page you will understand every file in that repository, every measured number it reports, and every mistake its author caught and wrote down along the way. The mistakes are the valuable part.

Who this is for

You are an engineer. You went to school for this, or you did it professionally, and then life took you elsewhere for a while. You can read code, but your day-to-day fluency has faded. Meanwhile the industry sprouted a new vocabulary (RAG, embeddings, vector databases, knowledge graphs, agents, MCP) and you would like to genuinely understand it rather than nod along. You have heard of retrieval-augmented generation and have a rough sense of what it is. You want to understand how a *graph* fits into an AI system, and you would rather learn from a real, working, measured project than from a hello-world tutorial.

Nothing here assumes recent experience. Where the book leans on something you may have half-forgotten, such as abstract syntax trees, git internals, cosine similarity, or the difference between SQL and graph queries, there is a refresher box at the point of use. Where the book uses a term of art, the glossary in Appendix A has it.

Why this repository, of all repositories

Most AI tutorials teach you to assemble parts: load documents, embed them, stuff a prompt, call a model. What they skip is the part that separates a demo from an engineered system: *how do you know it works, and how do you know each added layer earned its place?* LabGo is built in the opposite order. It builds the measuring stick first, then the simplest possible predictor, and only then adds vectors, then an agent. And it publishes the result even when the fancy new layer *loses* to the simple one. That happens in this book, in Chapter 12, and it is the most instructive chapter in it.

The repository also keeps an append-only decision log (`DECISIONS.md`) with nineteen entries at the time of writing. Each entry records what was decided, what the alternatives were, what evidence drove it, and what later changed. Five of those entries document the author catching their own numbers being misleading and correcting them. Those five are the real curriculum, and this book is organized so you hit each one in context.

How the book is organized

Part I sets up the problem and the two big ideas: the difference between searching by meaning and following connections, and the trick of mining git history for free ground truth. **Part II** builds the map by parsing Python into a code graph and mining history into an exam. **Part III** loads the graph into Neo4j and establishes the deterministic baseline, including the project's hardest-won lessons about leakage and prediction budgets. **Part IV** adds vector search and measures it honestly against the graph. **Part V** builds the LangGraph agent, watches it lose to the baseline, diagnoses why, and then turns the same tools inside

out as an MCP server. **Part VI** covers observability, CI, and the interactive viewer. **Part VII** distills the recurring lessons and hands you the keys: how to run all of it against your own repository.

Conventions

Five kinds of boxes recur throughout:

> **IF YOU'RE RUSTY**
>
> A short refresher on a concept the main text is about to lean on. Safe to skip if you already know it.

> **THE TRAP**
>
> A real bug or misleading number from this project's history: what it looked like, why it was believable, and how it was caught.

> **FROM THE DECISION LOG**
>
> A summarized entry from `DECISIONS.md`, cited by number (D001 to D019). When the text says "D012," this is the file it means.

> **TRY IT**
>
> Commands you can run yourself, against the demo corpus or your own repository.

> **ASIDE**
>
> Context that enriches but is not required: history, alternatives, industry practice.

Code excerpts are taken from the repository verbatim, occasionally shortened with `...` where boilerplate would not teach anything. All measured numbers come from the project's own runs against the `httpx` corpus (an open-source Python HTTP client library of about sixty Python files and 1,500 commits), and are quoted exactly as recorded in its README and decision log.

Contents

PART I

The Question

A change that looked safe breaks something four steps away. Why the obvious tools can't warn you, why this is a question about *connections* rather than *content*, and where a measuring stick comes from when nobody has time to write one.

1

The Tuesday Problem

In which a nineteen-line function breaks billing from four directories away, and we meet the question this entire book is about.

Tuesday

You are asked to change one function. It is called `parse_config`, it is nineteen lines long, and the change is small. You make it, you run the tests, they pass. You ship it.

On Friday, billing breaks.

The chain takes a while to reconstruct. A reporting module called a helper, which called a validator, which called `parse_config` and depended on it *raising an exception* rather than returning `None`. Four steps away, in a directory you had never opened. Nobody flagged it, because nobody knew. The engineer who would have known left in March.

This is not a story about carelessness. You did the normal things and they were not enough. It is a story about a question every developer asks constantly and nobody can answer reliably:

> ***If I change this, what else is affected?***

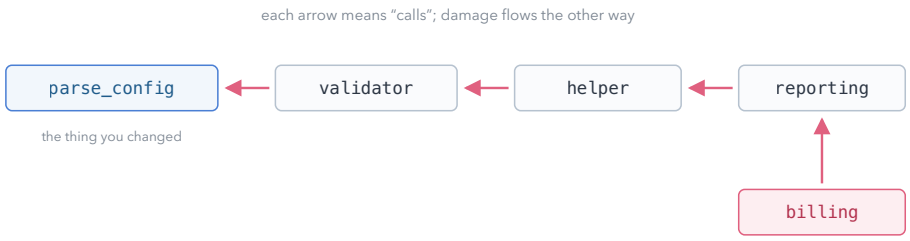


Figure 1.1 · The Friday chain. You changed the leftmost box; the crash surfaced in the bottom-right one. No single arrow is surprising. The *chain* is.

Why the question is hard

The obvious approaches all fail the same way: they see one step, and the damage is never one step away. If it were, you would have seen it yourself.

Searching (`grep parse_config`) shows you who calls the function directly. It does not show you who calls *them*. Each additional step multiplies the amount of code you would need to hold in your head. Three callers, each with three callers, each with three more, is twenty-seven places to check by hand at depth three, and you will not check twenty-seven places on a Tuesday.

Documentation describes the system as somebody understood it at the moment they wrote it down. Code moves; prose does not move with it.

Asking a colleague works right up until the person who knows has moved teams, or forgotten, or only ever knew their own corner. Institutional memory is real, and it walks out of the building a person at a time.

Asking an AI chatbot feels like it should work, and mostly does not. A chatbot reads a handful of files, notices some things that *look* related, and writes a confident paragraph. It has no mechanism for actually following the chain, so it produces something that reads like an answer and isn't one. That failure mode is worse than no answer, because it is persuasive. (What stops a language model from following the chain on its own, and what to give it so that it can, is the subject of the next chapter.)

So people ship changes half-blind, and find out on Friday.

The idea

Build a *map* of the codebase. Not a list of files but a map of *relationships*: which function calls which, which file imports which, which test exercises which code, which files have historically changed together. With that map, the question stops being a research project and becomes navigation. You start at the thing you are about to touch and follow the connections outward.

That map has a name in the AI industry, a **knowledge graph**, and the system this book dissects, **LabGo**, is one specific, complete, measured implementation of the idea. LabGo answers a three-part question about any Python repository you point it at:

> ***"If I change this file: what breaks, which tests must run, and who should review it?"***

What LabGo actually is

Concretely, LabGo is a Python package (`src/labgo/`) exposing one command-line tool, `labgo`, plus a small browser-based graph viewer. It was built in six deliberate stages, and the ordering of those stages is itself the first lesson of this book:

Stage	What gets built	What it proves
1	Parse code into a graph; mine git history into an exam	The map and the answer key exist before any "AI"
2	Load the graph into Neo4j; score a deterministic predictor	The honest floor: 22.4% recall with no LLM at all
3	Embed the code; measure vector search against the same exam	Vectors alone lose to the graph; the *union* nearly doubles it
4	A LangGraph agent that chooses among the tools	It loses to the deterministic baseline, measurably and instructively
5	An MCP server exposing the same tools to any AI client	Same capability, opposite direction of control
6	OpenTelemetry tracing + CI running the free half of the evals	You can see what every run cost and catch regressions

The six stages. Note what is *absent* until Stage 4: a language model making decisions.

Two constraints drove every design choice, and they are worth memorizing now because every chapter of this book returns to them:

- **The graph has to be load-bearing.** If the central question could be answered by a simpler tool, plain text search or the vector store behind every "chat with your docs" product, then using a graph database would be résumé decoration. LabGo is scoped to a question a vector store provably cannot answer, so that "why Neo4j?" has a real answer.
- **Everything gets measured.** Git history supplies free ground truth, so retrieval quality is a number, not a vibe. Every added layer must beat the number posted by the layer before it.

> FROM THE DECISION LOG · D001
>
> **Graph must be load-bearing, not decorative.** "Chat with your codebase" needs no graph; a vector store answers it. So the project is scoped to a question a vector store *provably cannot* answer: transitive impact. The rejected alternative, RAG over source files, was "cheap to build, no defensible reason for Neo4j." Consequence: the system must route between graph traversal, vector search, and plain deterministic code, and be able to justify each. *That routing decision is the deliverable.*

The demo corpus

Throughout the book, every measured number comes from running LabGo against `httpx`, a real, popular, well-maintained open-source Python HTTP client. It is a good exam subject precisely because it is unremarkable: about 60 Python files, roughly 1,300 functions and classes, and 1,523 commits of ordinary human history. Features, fixes, refactors, renames, mistakes. Everything LabGo claims, it claims about this corpus, with the corpus pinned to an exact commit so the claims stay checkable. Your own repository will produce different numbers; Chapter 17 shows you how to get them.

> ASIDE · THE NAME OF THE PROBLEM
>
> What this book calls "the Tuesday problem" the software-engineering literature calls *change impact analysis*, and it has been studied since at least the 1990s, long before LLMs. What is new is the surrounding tooling: graph databases that make multi-hop traversal a one-line query, embedding models that make "find me similar code" cheap, and language models that can orchestrate both. LabGo is change impact analysis rebuilt with 2026 parts, and measured with a discipline the demo economy usually skips.

Where we go from here

Chapter 2 builds the conceptual foundation: what retrieval-augmented generation actually is, what an embedding is, what a graph is, and why "what breaks if I change this?" belongs to the graph in a way no amount of clever searching can fix. Chapter 3 explains the project's best idea, which is where the answer key comes from. Then we start reading real code, and we do not stop until we have read all of it.

2

Two Ways to Find Things

The librarian and the subway map: semantic search, knowledge graphs, and the single sentence that justifies this whole architecture.

First, a fast refresher on the machine doing the talking

You cannot reason clearly about retrieval without a working model of the thing retrieval feeds. A **large language model** (LLM), the engine behind Claude, ChatGPT, and the rest, is a function that takes a sequence of tokens (word fragments) and predicts the next one, over and over, until it has produced a reply. Everything the model "knows" lives in two places:

- **Its weights:** billions of numbers fixed at training time. This is where general knowledge lives. Python syntax, English prose style, the concept of an HTTP client. Your repository, as it exists today, is not in there.
- **Its context window:** the text you hand it right now. This is the model's working memory. If a fact is in the context, the model can use it; if not, the model can only guess. Fluently, confidently, and sometimes wrongly. The polite industry term for a confident wrong guess is a **hallucination**.

That division of labor explains the past few years of AI system design in one sentence: since you cannot retrain the model on your private data every morning, *you fetch the relevant bits of your data at question time and place them into the context.* That pattern is called **retrieval-augmented generation**, RAG. The quality of a RAG system is dominated not by the model but by the fetching. Hand the model the right material and a mediocre model looks brilliant; hand it the wrong material and the best model in the world writes you a beautiful, useless paragraph.

So the serious question is never "should I use RAG?" It is: *how does the system decide what is relevant?* There are two genuinely different answers, and most people building AI systems today only reach for one of them.

The librarian: search by meaning

The first answer is **semantic search**, the machinery behind "chat with your documents" and most retrieval tools. The idea: convert every piece of text into an **embedding**, a list of numbers (a vector, typically a few hundred to a few thousand of them) with the remarkable property that texts with similar meanings get nearby vectors. "How do we retry failed requests?" and a function named `backoff_delays()` end up close together in that space even though they share no words. At question time you embed the question, find the nearest stored vectors, and hand those texts to the model.

Think of an excellent librarian. You say "something about handling flaky networks," and she returns with the retry documentation. She understood what you meant, not just what you said. Chapter 9 opens the machinery up properly: where the numbers come from, what "nearby" means mathematically, what a vector index does. For now the interface is what matters: text in, similar texts out.

The subway map: search by connection

The second answer stores *relationships* and lets you walk them. A **graph**, in this sense, is the mathematician's word: a set of **nodes** (things: files, functions, classes) connected by **edges** (typed relationships: *calls, imports, tests, changed-together-with*). A graph attaches no meaning to similarity; two functions can be word-for-word twins and sit in unconnected corners. What a graph is unbeatable at is **traversal**: start here, follow edges of this type, up to this many steps, and tell me everything you reach.

Think of a subway map. It is useless for "recommend me a nice station." It is unbeatable for "this station closes on Sunday, which journeys are disrupted?", because that is a question about following lines outward from a point.

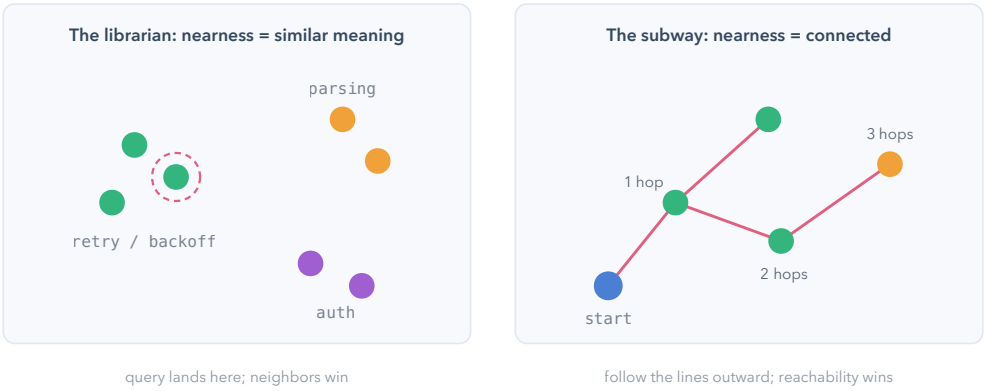


Figure 2.1 · Two meanings of "related." Left: an embedding space, where distance encodes similarity of meaning. Right: a graph, where an edge encodes a verified relationship and distance is counted in hops.

The whole idea in one sentence

> ***"What breaks if I change `parse_config`?" is a subway question, not a librarian question.***

Ask a semantic search engine and it returns other parsing functions. Code that *resembles* `parse_config`. Which is precisely, almost comically, the wrong answer. You did not want things that look alike; you wanted the chain of things that depend on it. Following a chain of connections is not something semantic search does badly. It is something semantic search cannot do *at all*, the same way a dictionary cannot tell you the fastest route across a city. Different tool, different question.

The formal name for what the Tuesday problem requires is **transitive closure** over the call graph: not just who touches X, but who touches anything that touches X, carried outward. "Transitive" is the property that damage propagates; "closure" is the complete set you reach when you keep going. Hold onto that phrase. When you meet Neo4j's query language in Chapter 7, you will see it expressed in a single line, and you will see why the equivalent in a conventional database or a vector store is somewhere between painful and impossible.

And yet the librarian keeps her job. Some questions about a codebase really are librarian questions: "where do we implement retry backoff?" has nothing to do with call chains, and a system with only a graph answers it badly. A dependency edge can also simply be *missing* from the map (Chapter 4 is honest about how often), and similar-looking code is sometimes genuinely coupled code. So the mature system needs both, and the real skill, the thing this repository exists to demonstrate, is knowing which kind of question you have been handed, and being able to prove your routing helped.

ASIDE · "GRAPH RAG"

The industry name for augmenting retrieval with a knowledge graph is **GraphRAG**, popularized around 2024. Most published variants use the graph to improve document question-answering: build a graph of entities from text, summarize communities of related entities, answer broad questions from the summaries. LabGo is a purer specimen. The domain, source code, *already is* a graph, no entity extraction required, and the target question is one that only traversal can answer. If you want to understand why graph-augmented systems work at all, code is the cleanest place to watch one do it.

Three signals, not two

LabGo's map actually carries a third signal, and it is the sleeper hit of the whole project. Alongside the structural edges parsed from source code (calls, imports, tests) it records a *behavioral* edge mined from git history: `CO_CHANGED`. These two files have repeatedly been modified in the same commit. No parser can see this relationship, because it often is not in the code at all: a client and its config format, a module and its documentation, two halves of a protocol that must move in lockstep. Files that change together are coupled, whatever the imports say.

Keep the three straight from the start, because the entire measurement story of this book is about refusing to blur them together before each has been scored on its own:

Signal	Source	Kind of evidence	Failure mode
`CALLS` graph	Parsing source (Stage 1)	Structural: A provably invokes B	Incomplete; Python hides most calls from static analysis
`CO_CHANGED`	Git history (Stage 1)	Behavioral: A and B moved together, repeatedly	Noisy; commits bundle unrelated work
Vectors	Embedding source (Stage 3)	Semantic: A resembles B	Resemblance is not dependency

The three retrieval signals. Each is measured alone before any blending is allowed, a discipline that catches three separate illusions later in the book.

IF YOU'RE RUSTY · WHY NOT JUST ONE DATABASE?

You may be wondering why one system carries three overlapping notions of "related." The honest answer: because each one is wrong *differently*. The call graph misses dynamic dispatch; git history bundles unrelated edits; embeddings confuse lookalikes with dependents. A signal that is wrong differently from your other signals is valuable precisely because their errors don't overlap. The measured proof arrives in Chapter 10, where the union of two mediocre signals nearly doubles either one alone.

What would convince a skeptic?

Suppose you build all this. Parser, graph database, embeddings, agent. It produces answers that look plausible, and a skeptical colleague asks the only question that matters: "how do you know it works?" Demos don't survive that question. Numbers do. But numbers need ground truth, and ground truth for "what else would this change affect?" sounds like it requires senior engineers hand-labeling hundreds of scenarios, which is slow, expensive, and famously inconsistent.

It doesn’t. The answer key has been sitting in the repository all along, and the next chapter is about the trick that reads it out.

3

The Answer Key in the Attic

Git history is full of correctly-answered copies of our question. How LabGo turns 1,482 commits into 610 exam questions, and how to read precision and recall like a professional.

Every commit is a labeled example

Here is the best idea in the project, and it costs nothing.

Every commit ever made is a developer asserting, under the best possible conditions: *when I changed this file, I also had to change these other files.* That is a question and a verified answer, recorded at the moment somebody actually did the work, by the person who knew why. A mature repository contains thousands of them.

So we set an exam. Take a real commit from history. Hide everything except one changed file, called the **seed**. Ask the system: what else needs to change? Compare its prediction against the files the developer actually changed, the **expected** set. Do that hundreds of times and average the scores, and you have a measurement nobody had to label.

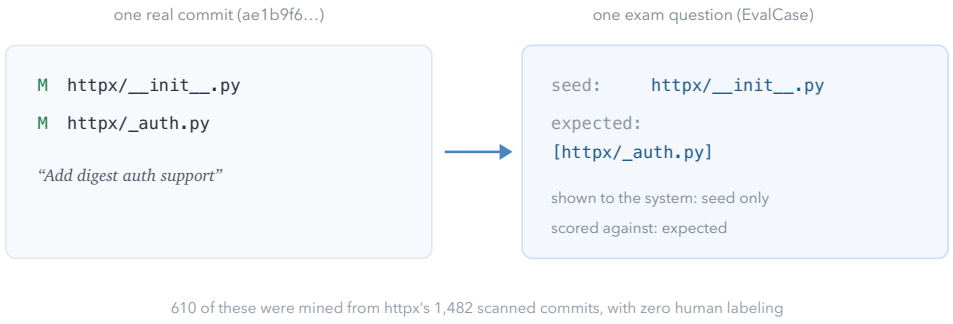


Figure 3.1 · From commit to exam question. This is a real case from the shipped benchmark: given `httpx/__init__.py` as the seed, the correct answer is `httpx/_auth.py`.

On `httpx`, scanning 1,482 commits produced 610 usable exam questions plus a record of which file pairs habitually change together (the `CO_CHANGED` evidence from Chapter 2; same mining pass, two outputs). The scan, the filters that keep the exam honest, and two delicious bugs encountered along the way are Chapter 5's subject. Here we care about what the exam lets us do.

> FROM THE DECISION LOG · D002
>
> **Ground truth comes from git history, not hand labels.** For each historical commit, feed the system one changed file and ask it to predict the rest. Recall against what actually shipped requires zero human labelling. Consequence: retrieval and generation can be evaluated separately. Known limitation: file renames split one logical file into two identities (httpx's `client.py` and its renamed successor `_client.py` carry the same coupling under two names), which understates coupling strength. Recorded, not hidden.

How to score a prediction

Suppose for some exam question the developer's commit changed four files besides the seed, and the system predicts five files, of which two are correct. Two numbers, both older than software engineering itself, capture everything worth knowing about that prediction:

- **Recall:** of the files that truly needed changing, what fraction did the prediction find? Here: 2 of 4, 50%. Recall measures *coverage*. Low recall means Friday surprises survive.
- **Precision:** of the files the system predicted, what fraction were right? Here: 2 of 5, 40%. Precision measures *trust*. Low precision means the tool cries wolf, and engineers stop reading its output.

The two live in tension, and the tension has a degenerate corner you must learn to spot on sight, because it is the villain of this entire book: *predict everything*, and recall hits 100% while the prediction becomes worthless. A system that answers "everything might break" is technically never wrong and practically useless. Its precision collapses, and so does the reader's attention. You will watch three separate, sophisticated-looking measurements in this project turn out to be this failure wearing a costume: an inflated vector score in Chapter 10, a 92% combined baseline in Chapter 8, and the agent's downfall in Chapter 12. The antidote, comparing methods only at a *matched prediction budget*, is introduced in Chapter 8 and reused for the rest of the book.

LabGo reports two further aggregates alongside the per-case means: **hit rate**, the fraction of exam questions where the system found at least one correct file (did it even show up?), and **mean predicted size**, the average number of files per answer. That last one is the budget number that unmasks the predict-everything costume.

> **IF YOU'RE RUSTY · READING A RECALL/PRECISION PAIR**
>
> A quick grammar for the tables ahead. *High recall, low precision*: a dragnet. Catches most of the truth plus a pile of bystanders. *Low recall, high precision*: a sniper. Rarely wrong, misses a lot. Which you want depends on the cost of each error. For change impact, a reviewer's time is the scarce resource, so a handful of high-quality warnings usually beats an exhaustive list nobody reads. And always ask for the *prediction size* next to any recall figure. Recall without size is an unfinished sentence.

The separation almost nobody bothers with

The exam measures **retrieval**: did the system find the right files? A complete impact analyst also involves **generation**: given the right files, does the explanation it writes about them make sense? LabGo's architecture keeps these two scores separate on purpose. When an answer is wrong, you immediately know which half failed. Without that separation all you learn is "it's wrong somewhere," which is the same as learning nothing. This is why the exam's ground truth is a plain set of file paths rather than prose: file sets can be scored mechanically, at scale, for free.

The answer key contains wrong answers, and that's fine

Honesty requires admitting what commit-mined labels are not. A commit's file list records what changed *together*, not what *had* to change together. Developers bundle: the drive-by typo fix, the "while I was in there" cleanup, the mechanical rename sweeping six files, two logical changes squashed into one commit. So the expected sets contain false positives. The answer key has wrong answers in it.

Three consequences, each of which shapes how every number in this book must be read:

- **A ceiling exists.** With meaningful label noise, no honest system scores near 100%. Chasing a high absolute number means overfitting to noise.
- **Relative comparison survives.** If the baseline scores 22% and a fancier method scores 40% against the same noisy key, the 18-point gap is real. The noise afflicts both sides equally and cancels. This is the core argument for building the boring baseline first, and it is why LabGo can make strong claims with a noisy exam.
- **Absolute claims do not survive.** Never "40% accurate at impact analysis," only "40% on this benchmark, which has known label noise." The noise is also *biased*, not random. Certain authors and eras bundle more, so it cannot be modeled away as uniform error.

FROM THE DECISION LOG · D009

Co-change labels are noisy, and the ceiling is not 100%. Mitigation (cheap, planned): also score against a high-confidence subset where the co-change is corroborated independently, either by an actual call-graph edge between the files or by a pairing recurring across many commits. Similar scores mean the noise isn't binding. Much higher scores on the clean subset mean a chunk of apparent failures are the benchmark's fault. This is the standing trade in mining version history: thousands of noisy labels or dozens of clean hand-written ones. The failure mode is not knowing which you have.

Why the boring version gets built first

With the exam in hand, the project's build order becomes a strategy rather than a preference. The next thing built is the stupid version, no AI anywhere near it: files within two steps on the call map, plus files that historically changed alongside this one. Score it. Suppose it gets 22% (it will; see Chapter 8).

That unremarkable number is then the most valuable artifact in the project, because when the agents arrive and score 30%, you can say exactly what they were worth. And if they score 23%, you have learned something more useful still: the expensive, slow, complicated layer bought you one point, and the problem was somewhere else all along. Skip the boring version and you end up with a system that scores 70% and no way to know whether that is excellent or whether the exam was easy. Nearly every AI portfolio project skips it, and it is the reason so many collapse under a single interview question:

> ***How do you know the agents helped?***

LabGo was built in this order specifically so that question has an answer. By Chapter 12 you will have watched the answer turn out to be "they didn't, yet, and here is precisely why." That is a far more hireable sentence than any demo.

TRY IT · SEE THE RAW MATERIALS

Everything in this Part exists as plain files you can read. In the LabGo repository: `WHY.md` is the prose argument this Part expanded; `DECISIONS.md` is the log the purple boxes quote; `benchmarks/httpx/cases.json` holds all 236 pinned exam questions (Figure 3.1 is the first one); and `benchmarks/httpx/manifest.json` records exactly how they were produced. No tooling needed. Open them in any editor.

PART II

Building the Map

Stage 1: two programs, neither of which uses AI, and that is deliberate. A parser that turns Python source into a graph of who-calls-whom, and a miner that turns git history into an exam. Plus the art of keeping a metric honest before you record it.

4

Reading Code with Code

Python ships a parser for itself in the standard library. LabGo uses it to turn sixty files into 1,301 nodes and 2,100 edges, and then tells you, to the decimal, how much of the truth that map is missing.

What an AST is, and why we want one

To build a map of who-calls-whom you must read code the way the language itself does. Text search will not do: `grep "helper("` matches calls, definitions, comments, and strings alike, and has no idea that `helper` on line 40 belongs to a different class than `helper` on line 90. What you want is the **abstract syntax tree** (AST): the tree structure the interpreter builds from your source before executing it, in which every construct is a typed node whose children are its parts. A class definition, a function, a call, an import: each is a node.

> **IF YOU'RE RUSTY · A TWO-LINE AST**
>
> The snippet `def f(): return g(1)` parses into a tree shaped like: Module → FunctionDef(name="f") → Return → Call(func=Name("g"), args=[Constant(1)]). Nothing runs; parsing is purely structural. Python exposes this via the standard-library `ast` module: `ast.parse(source)` returns the tree; you then walk it. One gift matters most for our purposes: every node carries its line span (`lineno` / `end_lineno`), which Stage 3 will quietly rely on to find each function's source text without storing a single line of code in the graph.

The schema: three node kinds, five edge kinds

Before parsing anything, LabGo pins down what the map may contain, in one small file, `src/labgo/ingest/models.py`, using plain frozen dataclasses. Deliberately plain: no database types, no ORM, so the extractor can run and be tested with no database at all. Persistence is somebody else's job (Chapter 7). The schema is small enough to hold in your head, and you should, because every later chapter speaks in these terms:

```python
class NodeKind(StrEnum):
    FILE = "File"
    FUNCTION = "Function"
    CLASS = "Class"


class EdgeKind(StrEnum):
    CONTAINS = "CONTAINS"       # File -> Function | Class
    CALLS = "CALLS"             # Function -> Function
    IMPORTS = "IMPORTS"         # File -> File
    TESTS = "TESTS"             # Function(test) -> Function
    CO_CHANGED = "CO_CHANGED"   # File -> File   (from git history)


@dataclass(frozen=True)
class Node:
    id: str          # stable unique key, e.g. "pkg/mod.py::MyClass.method"
    kind: NodeKind
    name: str
    path: str
    lineno: int | None = None
    end_lineno: int | None = None
    props: dict[str, Any] = field(default_factory=dict)
```

From `ingest/models.py`: the entire vocabulary of the map.

Two design choices here reward a close look. First, the **node id** is `path::qualified-name`. It is derivable from source alone, unique across the repository, and stable across runs. No auto-increment counters, which would shuffle on every re-parse and make two graphs incomparable. Second, `CO_CHANGED` sits in the same enum as the structural edges but *is* documented as the odd one out: it comes from git history rather than source, and it is the only edge carrying evidence about *behavior over time* rather than structure. The schema keeps them side by side. The *measurement* chapters keep them strictly apart, for reasons that will get dramatic in Chapter 8.

Walking the tree: the visitor

Parsing happens in `ingest/pyast.py`. Python's `ast.NodeVisitor` implements the classic visitor pattern: subclass it, define `visit_ClassDef`, `visit_FunctionDef`, `visit_Call` and so on, and the walker calls your method for each matching node. LabGo's `_FileVisitor` maintains three stacks while it walks (the qualified-name scope, the enclosing functions, and the enclosing classes) so that when it meets a call site deep inside `class Engine: def run(self): ...`, it knows the caller is `pkg/core.py::Engine.run` and not merely "somewhere in the file."

The subtle part is what the visitor does *not* do: it does not try to resolve calls while walking. It just records raw evidence. Who called what name, on what receiver, inside which class:

```
def visit_Call(self, node: ast.Call) -> None:
    if not self._func_stack:
        self.generic_visit(node)        # module-level call; no caller to attribute to
        return
    caller = self._func_stack[-1]
    cls = self._class_stack[-1] if self._class_stack else None
    func = node.func
    if isinstance(func, ast.Name):                    # helper(...)
        self.calls.append((caller, func.id, None, cls))
    elif isinstance(func, ast.Attribute):             # self.step(...), client.send(...)
        base = func.value.id if isinstance(func.value, ast.Name) else None
        self.calls.append((caller, func.attr, base, cls))
    self.generic_visit(node)
```

Call sites are *recorded*, not resolved. Resolution needs every file's definitions, which don't exist until all files are parsed.

Hence the two-pass shape of `extract_repo()`: pass one parses every file, collecting definitions, imports, and raw call sites, and builds a repository-wide symbol table (name to definitions, module to file). Pass two emits `IMPORTS` edges. Pass three walks the recorded call sites and tries to resolve each one against the now-complete symbol table. A file with a syntax error is counted (`files_failed`) and skipped, never fatal. Real repositories contain broken files, and a mapper that dies on the first one maps nothing.

The resolution ladder

Here is the uncomfortable truth the whole chapter turns on: **in Python, you cannot, in general, know statically which function a call invokes.** When the parser sees `response.read()`, what is `response`? Its type is decided at runtime; it could be anything, including something assembled from a config file at startup. `getattr(obj, name)()`, duck typing, decorators that swap implementations, monkey-patching: all unresolvable without executing the program. This is not sloppiness. For a dynamic language it is impossible *in principle*.

LabGo's response is not to pretend. Every call site is pushed down a ladder of four increasingly desperate strategies, and every edge that comes out the bottom is stamped with which rung produced it:

```python
def _resolve_one(rel, call, v, idx):
    _caller, name, base, cls_qual = call

    # (a) self.foo() / cls.foo() inside a class -> that class's method
    if base in ("self", "cls") and cls_qual:
        cand = idx.defs_by_id.get(f"{rel}::{cls_qual}.{name}")
        if cand:
            return cand.id, Confidence.SELF

    # (b) explicit import: `from mod import name`
    if name in v.imports:
        mod, orig = v.imports[name]
        tgt_file = idx.module_to_file.get(mod)
        if tgt_file and orig:
            cand_id = idx.local_index.get((tgt_file, orig))
            if cand_id:
                return cand_id, Confidence.EXACT

    # (c) defined in the same file
    cand_id = idx.local_index.get((rel, name))
    if cand_id:
        return cand_id, Confidence.LOCAL

    # (d) unique name anywhere in the repo
    cands = idx.by_name.get(name, [])
    if len(cands) == 1:
        return cands[0], Confidence.HEURISTIC

    return None, None
```

The ladder in `pyast.py`. A call that survives no rung is counted as unresolved. Counted, not hidden.

Read the rungs in order of trust. (a) `self.step()` inside `class Engine` resolves to `Engine.step` in the same file: near-certain. (b) a name arriving through an explicit `from mod import name` resolves to that module's definition, which is certain when the module is in this repository. (c) a bare name defined in the same file: probably right, shadowing aside. (d) a name defined exactly once anywhere in the repository: a bet, honestly labeled `heuristic`. If two files define `helper`, the ladder refuses to guess and the call goes unresolved. The docstring of `_resolve_one` notes that it was split into its own function because this is where a type-inference backend would plug in, if the evidence ever justifies one. A hook for a future that has deliberately not been built.

One bonus falls out for free: when the resolved caller is a test function (name starts with `test_`, in a test file), the resolver emits a second edge, `TESTS`, alongside `CALLS`. A test calling a function is evidence of coverage, and that is the entire mechanism behind "which tests must run," one third of LabGo's headline question.

The honest denominator

So: what fraction of calls does the ladder resolve? The first implementation measured 20.2%. That number was wrong, in the manner numbers are most dangerously wrong: not by lying, but by dividing by the wrong thing.

THE TRAP · A WRONG DENOMINATOR (D005)

The 20.2% counted *every* call site in the denominator, including `len()`, `isinstance()`, and calls into the standard library. Those were never repository-internal functions; no resolver, however clever, could ever point them at a node in the map. Counting them as "unresolved" made the resolution rate meaningless. Worse than meaningless, actually, because it would have been recorded as *the baseline*, and every later comparison would have silently inherited the error. The fix: classify builtin/stdlib/third-party calls as *external* and exclude them, then ask the honest question. *Of the calls that could plausibly have been internal, how many did we resolve?* Same graph, same resolver: **27.2%**. The lesson worth keeping verbatim: a wrong denominator is worse than no metric.

The classification lives in `_is_external()`, and its docstring records a second-order subtlety: the function is *conservative on purpose*. A name that also exists as a repository definition is never classified external, even if it shadows a builtin (a repo that defines its own `list()` gets a `CALLS` edge, not an exclusion). Why? Over-classifying calls as external would shrink the denominator and *inflate* the resolution rate, which is the exact failure mode the function exists to correct. There is a test pinning this behavior (`test_shadowed_builtin_is_not_external`), and the stats dataclass makes the arithmetic self-describing:

```python
@property
def in_scope_calls(self) -> int:
    """Call sites that could plausibly target a repo-internal function."""
    return self.total_calls - self.external_calls


@property
def resolution_rate(self) -> float:
    """Resolved fraction of in-scope calls. Zero when there is nothing to resolve."""
    if self.in_scope_calls == 0:
        return 0.0
    return self.resolved / self.in_scope_calls
```

From `models.py`: the denominator, encoded so it cannot silently regress.

What the map is missing, measured

Run against `httpx`, the extractor reports:

in-scope call sites (excludes builtin/stdlib/third-party)	count
total in scope	2,845
resolved: exact (import)	4
resolved: self/cls	99
resolved: local (same file)	327
resolved: heuristic (unique name)	343
unresolved	2,072
resolution rate	**27.2%**

Call resolution on httpx (D004). The residual is diagnosed, not mysterious: almost all misses are method calls on local variables whose type is unknowable statically.

Nearly three-quarters of the in-scope call graph is invisible to static analysis of this kind. Sit with that number, because it recalibrates everything downstream. The call graph is the project's most trustworthy signal and it is still mostly blind. This is why Chapter 2 insisted on three signals: the co-change evidence exists to catch coupling the parser cannot see, and on httpx there turns out to be a lot of it.

> FROM THE DECISION LOG · D004
>
> **Static call resolution: name-based, with honest confidence tiers.** Tools exist (pyright, jedi) that would lift 27.2% materially by inferring types, at a real cost in complexity and runtime. Decision: *do not adopt type inference until the eval set shows call-graph recall is the binding constraint on end-to-end performance.* It may not be. The co-change signal may carry most of the weight, and Chapter 8 tests exactly this. "That restraint," as `WHY.md` puts it, "is most of what architecture is."

> TRY IT
>
> `uv sync`, then `uv run labgo ingest ../labgo-corpora/httpx`. No database, no API key, no network. It prints the resolution table for whatever repository you point it at and writes the whole map to `data/graph.json`. Open that file and find yourself in it: every node and edge in this chapter is a plain JSON object.

5

Mining History

One git log invocation, two output streams (co-change evidence and exam questions), and two of the sneakiest bugs in the whole project, both hiding in the space between a format string and a newline.

What git actually records

A git repository is, at heart, an append-only ledger of snapshots. Each commit records an author, a timestamp, a message, and the complete state of the tree. From that, git can derive, for display, the list of files that changed relative to the parent commit. That last list is the raw material of this chapter. LabGo asks git for it with one command:

```
git log --no-merges --max-count=2000 --format=@@C@@%H%x1f%at%x1f%an --name-only
```

which emits, for each commit, a header line (hash, timestamp, author) followed by the changed file paths, one per line. Parsing this stream sounds like the easy part. It produced two of the best bugs in the decision log.

> **THE TRAP, TWICE · BYTES IN ARGV AND THE SPLITLINES AMBUSH (D006)**
>
> **Trap one.** The natural instinct for an unambiguous field separator is a null byte: `--format=...\x00...`. Python refuses: `ValueError: embedded null byte`. The strings a process passes as arguments (argv) are C strings, null-terminated, so a null can never appear inside one. The fix is git's own escape code `%x1f`: the argument carries the literal four characters "%x1f", and git itself emits the byte on output.
>
> **Trap two** is nastier. Switching the record marker to the ASCII "record separator" byte `\x1e`, the character built for exactly this job, made the parser return zero commits, *silently*. Why? Python's `str.splitlines()` treats `\x1c`, `\x1d`, and `\x1e` as line boundaries. The marker was being consumed as a newline, so no line ever started with it. No exception, no warning. Just a plausible-looking empty result. (`\x1f`, the unit separator, is *not* a splitlines boundary, which is why the field separator survived.) The shipped format therefore uses a literal text marker, `@@C@@`, for records and `%x1f` for fields, and the module comment explains both scars.
>
> The lesson generalizes: the second bug failed silently with a plausible zero. Any pipeline stage that can legitimately return empty needs an assertion or a test pinning non-emptiness. A happy-path test proves nothing here, because the happy path is exactly what a silent zero imitates. The test suite contains `test_git_log_parser_rejects_silent_emptiness` for precisely this.

The parser itself, `parse_commits()`, is a small generator, split from the subprocess call so it can be tested against literal strings with no repository at all. That separation exists because of the silent-zero incident:

```python
def parse_commits(raw: str) -> Iterator[Commit]:
    sha = ts = author = None
    files: list[str] = []
    for line in raw.splitlines():
        if line.startswith(_REC):                       # "@@C@@"
            if sha is not None:
                yield Commit(sha, int(ts or 0), author or "", files)
            parts = line[len(_REC):].split(_SEP)        # "\x1f"
            sha, ts, author = [*parts, "", "", ""][:3]
            files = []
        elif line.strip():
            files.append(line.strip())
    if sha is not None:
        yield Commit(sha, int(ts or 0), author or "", files)
```

From `ingest/gitlog.py`. Boring by design. The interesting decisions are in what gets filtered, next.

The filters are the intellectual content

The module docstring says it plainly: “the filtering decisions here matter more than the code.” Raw commit history is full of events that would poison both outputs. Each filter is a precision/recall trade made explicitly:

- **Merge commits are excluded.** A merge’s file list is the union of unrelated branches; treating it as evidence of coupling manufactures edges that don’t exist.
- **Only source files count.** Lockfiles (`uv.lock`, `package-lock.json`) and changelogs co-change with everything; they would inflate apparent coupling while carrying none.
- **Single-file commits are unusable.** No pair, no prediction task, skipped without suspicion.
- **Large commits are excluded**, and this one deserves its own arithmetic.

A commit touching N files asserts a co-change relationship for every pair of them: $N(N-1)/2$ pairs. Five files make 10 pairs; twenty make 190; forty-seven make 1,081. Pair production grows with the *square* of commit size, which means the largest, least meaningful commits (the lint sweep, the great renaming) dominate the evidence unless you stop them. Measured on httpx before capping:

files/commit	commits	pairs	% of all pairs
2-3	328	546	3.9%
4-5	131	982	7.0%
6-10	106	2,737	19.4%
11-20	45	4,377	31.0%
21-50	10	5,483	38.8%

Pair production by commit size on httpx (D010). Ten commits, 1.6% of the usable history, would contribute 38.8% of all co-change evidence. The five largest alone contribute 30%.

One cap becomes two

The original design used a single `max_files` cap for “too large to count.” Then a design flaw surfaced, subtle enough to be worth pausing on: that one parameter governed both *which commits contribute co-*

change evidence and *which commits become exam questions*. Those two uses want different values. The exam wants small, focused commits (labels can't be de-noised after the fact), while the evidence tolerates looser input because a downstream threshold filters it anyway. Worse, coupling them means tuning the threshold to improve your evidence silently rewrites the exam you are scoring against. That is a mild but real form of leakage, of tuning the test to fit the answer. So the parameter was split:

```
def extract_history(
    repo: Path,
    max_commits: int = 2000,
    evidence_max_files: int = 20,   # above this: no co-change pairs
    eval_max_files: int = 10,       # above this: pairs kept, but no exam case
    min_files: int = 2,
) -> History:
    ...
    for commit in parse_commits(_run_git_log(repo, max_commits)):
        files = sorted({f for f in commit.files if _is_source(f)})
        ...
        for i, a in enumerate(files):
            for b in files[i + 1:]:
                hist.pairs[(a, b)] += 1        # co-change evidence

        if len(files) > eval_max_files:
            st.too_large_for_eval += 1
            continue                           # evidence yes, exam no

        hist.cases.append(EvalCase(sha=commit.sha, seed=files[0], expected=files[1:]))
```

The two caps in `extract_history()`. One commit stream, two outputs, different admission standards.

What the seed is not

Notice `seed=files[0]`: the seed is the alphabetically first source file in the commit. The `EvalCase` docstring spells out two things the seed is *not*, and the care taken here is a small masterclass in dataset documentation. It is not "the file the developer touched first." Git records no edit order; that information does not exist in history, so pretending otherwise would be fiction. And it is not a clean causal label. `expected` is what changed together, not what *had* to change (Chapter 3's noise argument, D009). Alphabetical choice is arbitrary but deterministic: the same history always yields the same exam.

The evidence stream gets its cheapest quality filter on the way out: `co_change_edges(hist, min_count=2)` keeps only pairs seen together at least twice. A pair seen once is a coincidence; twice is the beginning of a habit. That parameter, `min_count`, looks like a throwaway here. Remember it. In Chapter 8 it turns out to be the single most consequential knob in the project.

The audit trail

Every filtering decision increments a counter in `HistoryStats` (commits scanned, merges skipped, too large for evidence, too large for exam, no source files, used), and `labgo history` prints them all. On httpx: 1,482 commits scanned, 610 exam cases out the other end, 1,745 co-change edges at `min_count=2`. The strongest coupling it prints is a preview of Chapter 8's drama: `httpx/_client.py` ↔ `httpx/_models.py`, changed together 39 times. No single number here is impressive, and that is the point. The value is that none of the drops happened silently.

ASIDE · THIS TRICK HAS A NAME

Mining co-change from version history is a respectable research lineage. The software-engineering literature calls it *evolutionary coupling* or *logical coupling*, studied since the early 2000s, and it powers commercial code-health tools (Adam Tornhill's *Your Code as a Crime Scene* is the popular treatment). The insight is always the same: version history knows things the compiler cannot, because it records what people actually did, repeatedly, under pressure. LabGo's contribution is not the idea. It is the discipline of measuring what the idea is worth on this corpus before believing it.

TRY IT

`uv run labgo history ../labgo-corpora/httpx`. Still no database, no key, no network. It prints the audit table and the strongest couplings, then writes `data/cochange.json` and `data/evalset.json`. Note the closing warning it prints: the eval set is *raw*, not yet usable for scoring. Why not is the next chapter.

6

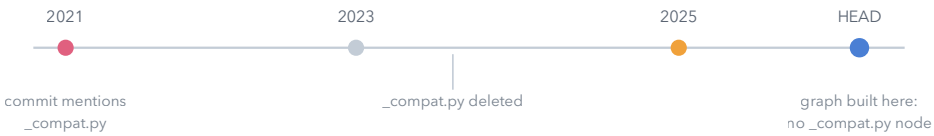
Pinning the Exam

In which 45% of the exam turns out to be unanswerable by construction, and the fix is a benchmark that carries its own birth certificate.

The question that found the flaw

Between mining the eval set and scoring anything against it, the author asked a simple due-diligence question: *does the eval set mean anything without the corpus?* The answer dismantled the naive plan.

Exam questions are mined from commits across all of history, some years old. But the map (Chapter 4) is built from the corpus as it exists *today*, at HEAD. In the years between, files get deleted and renamed. A 2021 commit whose seed is `httpx/_compat.py` poses a question about a file that no longer exists. The system cannot look it up in the graph, cannot traverse from it, cannot even name it. The case isn't hard; it is unanswerable by construction. This is **temporal drift**: the exam and the world it examines have drifted apart in time.



A question mined on the left cannot be answered by a map built on the right.

Figure 6.1 · Temporal drift (D008). The exam spans history; the map is a snapshot. Every case referencing a since-deleted file is unanswerable no matter how good the system is.

Measured on httpx, the damage was not a rounding error:

	cases
cases whose seed no longer exists	278 / 610
cases with ≥1 expected file gone	305 / 610
cases where nothing survives	101 / 610

D008's audit. Any score computed against the raw set would be silently depressed by ~45% impossible questions, and misread as the system underperforming.

Three fixes, one chosen, one recorded

The decision log weighs three responses like an engineer, not an advocate:

- **Filter** to cases where every referenced file still exists at the pinned commit. Cheap. Introduces *survivorship bias*: you only examine code that lasted, and code that survives refactors is systematically different from code that didn't.

- **Restrict the window** to recent history where file identity is stable. Fewer cases, same bias in softer form.
- **Time-travel:** for each case, rebuild the graph from that commit's parent, so the system sees exactly the world the developer saw. Methodologically ideal, and N times the cost, rebuilding the graph per case.

Chosen: filter now, with the bias documented in the artifact itself, and time-travel recorded as the rigorous version to build if a result ever hinges on it. On httpx the filter kept 236 of 565 raw cases, dropping 58.2%. Losing most of the exam is the right trade. 236 answerable questions beat 565 of which two in five are impossible, because the second set produces a number that *looks* like underperformance and is actually arithmetic.

What "pinned" has to mean

The episode produced a corollary the project treats as law: **a score is only meaningful if four things are fixed together.** (1) The corpus, at an exact commit. (2) The eval set. (3) The extraction parameters that produced it; change `eval_max_files` and you have a different exam. (4) Temporal consistency between (1) and (2). Committing the eval set alone, the original plan, was half a fix. So `labgo benchmark` writes a benchmark directory that fixes all four, and commits it to version control:

```
benchmarks/httpx/
  manifest.json    corpus URL + exact SHA, extraction params, filter rule,
                   the survivorship bias it introduces, creation time
  cases.json       the 236 answerable EvalCases
  evidence.json    the RAW co-change pair counts (all of them, unfiltered)
```

Two details in the writer deserve attention. First, the manifest carries its own warning label. The JSON literally contains: "Committed to version control on purpose. Regenerating this changes the exam; do it as a logged decision, never as a side effect of updating the corpus." Second, `evidence.json` stores the co-change pairs *raw and unfiltered*: every count, including the singletons the `min_count=2` filter would discard. At this point in the book that looks like wasteful hoarding. It is actually load-bearing foresight. Chapter 8 will need to *subtract* one commit's contribution from the evidence, and you cannot correctly subtract from a total that was filtered first. The reason is spelled out in D012, where this file becomes the hinge of the project's biggest correction.

Refusing to produce a wrong number

The last piece is enforcement. A benchmark pinned to commit `b5addb64` is meaningless if you score it while the corpus sits at some other commit. So the scorer checks, and refuses:

```python
def verify_corpus(manifest: dict[str, Any], repo: Path) -> None:
    """Fail loudly if the corpus is not at the pinned commit.

    Scoring against a drifted corpus produces a number that looks fine and means
    nothing. A wrong answer is recoverable; a plausible wrong answer is not.
    """
    expected = manifest["corpus"]["sha"]
    actual = corpus_sha(repo)
    if actual != expected:
        raise CorpusMismatchError(
            f"corpus is at {actual[:12]}, benchmark '{manifest['name']}' was built "
            f"against {expected[:12]}.\n"
            f"Scores would not be comparable. Fix with:\n"
            f"    git -C {repo} checkout {expected}"
        )
```

From `benchmark.py`. Note the error message includes the exact command that fixes the problem, a courtesy this codebase extends everywhere and defends in its lint config.

Read that docstring line again, *"a wrong answer is recoverable; a plausible wrong answer is not,"* because it is the thesis of the whole Part. Chapter 4 refused a plausible resolution rate with a wrong denominator. Chapter 5 refused a plausible zero from a silent parser. This chapter refuses a plausible score against a drifted world. Stage 1 ends with no AI, no database, and no impressive capability. Just a map, an exam, and a set of guarantees that every number from here forward will mean what it appears to mean. That foundation is about to pay for itself three times over.

TRY IT

`uv run labgo benchmark ../labgo-corpora/httpx --name httpx` builds the pinned directory; `uv run labgo verify benchmarks/httpx ../labgo-corpora/httpx` checks it. Then check out any other commit of httpx and run `verify` again to watch it refuse, politely, with the fix in the error message.

PART III

The Graph

Stage 2: the map goes into Neo4j, one Cypher query answers the Tuesday question with no AI at all, and the project catches its own benchmark handing out partial credit for having seen the answers. The most important Part in the book.

7

A Database Made of Arrows

Property graphs, the Cypher query language, and why "follow the arrows as far as they go" is one line in a graph database and a term paper in SQL.

The property graph model

A graph database stores what Chapter 4 produced: nodes and edges, both carrying properties. Neo4j, the one LabGo uses and the category's incumbent, calls this the **property graph model**:

- **Nodes** carry labels (like `:File`, `:Function`) and key-value properties (`id`, `path`, `lineno`).
- **Relationships are first-class citizens**: directed, typed (`:CALLS`, `:CO_CHANGED`), and property-carrying (`confidence`, `count`). This is the defining difference from a relational database, where a relationship is an implicit thing you reconstruct with a join every time you want it.

Queries are written in **Cypher**, and the reason engineers love it is that the query is a *picture* of the pattern you want. Nodes go in parentheses, relationships in square-bracketed arrows:

```
MATCH (caller:Function)-[:CALLS]->(fn:Function {name: "parse_config"})
RETURN caller.id
```

reads exactly as it looks: find every Function node with an arrow of type CALLS pointing at a function named `parse_config`. The superpower, the one this entire architecture was chosen for, is the **variable-length pattern**:

```
MATCH (caller:Function)-[:CALLS*1..3]->(fn)
```

The `*1..3` means "follow one, two, or three CALLS arrows in a row." That is Chapter 2's transitive closure, the whole Tuesday problem, as four characters in a pattern.

> **IF YOU'RE RUSTY · WHAT THIS COSTS IN SQL**
>
> In a relational database you would store edges as a table `calls(src, dst)` and answer depth-1 with a join. Depth-2 is a join of the join. Arbitrary depth requires a recursive common table expression, `WITH RECURSIVE`, with explicit cycle handling and de-duplication, and the query planner treats each depth as fresh joins against ever-growing intermediate results. It can be done; people do it; nobody enjoys it. Performance also degrades with depth in a way graph-native storage is specifically engineered to avoid (Neo4j chases pointers from node to node rather than re-scanning join tables). The deeper point is D001's: if your central question is multi-hop, a store where hops are native is load-bearing, not fashionable. A vector store cannot express the question at all. It has no edges.

Loading the map

`src/labgo/graph.py` is the bridge: it reads Stage 1's `graph.json` (plus `cochange.json` if present) back into dataclasses and pushes them into Neo4j. It is sixty lines of workmanlike loader wrapped around four decisions worth understanding.

One generic label plus one specific label per node. Every node is created as `:Node:File`, `:Node:Function`, or `:Node:Class`. The generic `:Node` label lets a single uniqueness constraint cover all three kinds. In Chapter 9 it will also let a single vector index cover everything embeddable, while the specific labels keep queries expressive.

`MERGE`, not `CREATE`. Cypher's `MERGE` is find-or-create: run the loader twice and you get the same graph, not a duplicated one. Idempotency turns "did the load succeed?" from a forensic question into a shrug. Just run it again.

```
for batch in _chunks(rows, batch_size):                # 500 rows per round-trip
    session.run(
        f"UNWIND $rows AS row MERGE (n:Node:{kind.value} {{id: row.id}}) SET n += row",
        rows=batch,
    )
```

The node loader. `UNWIND` turns one network round-trip into 500 rows of work: the difference between a 3-second load and a 20-minute one.

Batching. Sending 1,301 nodes as 1,301 queries means 1,301 network round-trips. To a cloud database, that dominates everything. `UNWIND` passes a list as a parameter and iterates server-side.

Direction as storage, not as meaning. `CO_CHANGED` edges are stored directed (an edge must point somewhere) but git history carries no inherent direction between two files that changed together. So every query that touches them uses the undirected pattern `-[:CO_CHANGED]-`, with no arrowhead. A comment in the loader records this so no future query author infers meaning from an arrow that has none.

Where the database lives, and why that changed

The original plan ran Neo4j locally in Docker; the repo still ships the `docker-compose.yml`. The shipped default is the opposite: a free cloud instance (Neo4j AuraDB Free). The decision log entry is a small study in how to change your mind honestly.

> **FROM THE DECISION LOG · D013 (AND D011'S PRECEDENT)**
>
> The project began with a local-only stance. D011 broke it first, for embeddings (Chapter 9): the development laptop is resource-constrained, and running models locally competes with everything else on the machine. D013 then applied the same reasoning to the graph: *"once one cloud dependency is accepted for laptop-resource reasons, a second free one for the same reason is not a new trade-off. It's the same one, again."* Due diligence recorded in the entry: AuraDB Free's capacity (either quoted bound comfortably holds httpx's 1,301 nodes), and the real binding constraint, which is that a free instance auto-pauses after 72 idle hours and is deleted 30 days later. An odd fit for a learning project touched in bursts. Mitigations: `data/graph.json` stays the source of truth (the database is regenerable), and the Docker file is kept as the offline fallback. Notice also what the entry does: it corrects the project's own earlier overstatement, noting that D001 never actually required local hosting. The docker-compose comment had claimed more than the decision said.

One operational consequence is flagged rather than buried: with a single cloud instance and no local/prod split, `labgo load --clear`, which wipes the database first, is now a destructive action against the only copy. Recoverable (re-ingest, re-load), but the kind of thing you want written down before 2am, not after.

The `connect()` helper completes the picture: URI, user, and password come from `.env`, never hard-coded, and a missing password raises immediately with a message that says exactly which file to fix.

> **TRY IT**
>
> Create a free instance at Neo4j Aura (or `docker compose up -d`), copy `.env.example` to `.env`, fill in `NEO4J_URI` and `NEO4J_PASSWORD`, then `uv run labgo load --clear`. It prints the node and edge counts as they land. Then open the Aura console's query pane and try the chapter's first Cypher query yourself.

8

The Number to Beat

One deterministic query earns 22.4% and becomes the project's honest floor. Then a 94.9% appears, and dies twice: once of leakage, once of an unbounded budget. If you read only one chapter carefully, make it this one.

The predictor with no opinions

Stage 2's deliverable is `baseline.py`: impact prediction as pure graph traversal. No LLM, no ranking, no cleverness. Given a seed file, find every function it contains, walk the `CALLS` arrows *backwards* up to `hops` steps (who calls into this file, and who calls them), and return the files those callers live in:

```python
def predict_impact_calls(driver: Driver, seed: str, *, hops: int = 2) -> set[str]:
    predicted: set[str] = set()
    with driver.session() as session:
        result = session.run(
            f"MATCH (:File {{id: $seed}})-[:CONTAINS]->(fn:Function) "
            f"MATCH (caller:Function)-[:CALLS*1..{hops}]->(fn) "
            f"MATCH (callerFile:File)-[:CONTAINS]->(caller) "
            f"RETURN DISTINCT callerFile.id AS file",
            seed=seed,
        )
        predicted.update(r["file"] for r in result)
    predicted.discard(seed)
    return predicted
```

The Stage 2 predictor, whole. Three `MATCH` clauses: the seed's functions; their transitive callers; the callers' files.

Read the Cypher as a sentence: *start at the File node whose id is the seed; step to the Functions it CONTAINS; find every Function reaching them through one to `hops` CALLS arrows; report the Files containing those callers.* The seed itself is discarded from its own prediction, since predicting "the file you are editing will change" earns no points. One engineering footnote: `hops` is interpolated into the query text rather than passed as a parameter, because Neo4j does not allow parameterizing a variable-length range. Interpolating values into query strings is how injection vulnerabilities happen, so the docstring justifies it: `hops` is an internal integer, never user-supplied text. The right reflex is not "never interpolate" but "never interpolate without writing down why it's safe."

Scoring (from Chapter 3) is a list comprehension away: for each of the 236 pinned cases, compute the prediction, intersect with `expected`, average recall and precision per case. The result, on httpx, at `hops=2`:

metric	value
mean recall	22.4%
mean precision	12.2%
hit rate (found ≥1 correct file)	28.8%
empty predictions (said nothing at all)	94 / 236
mean prediction size	4.8 files

The call-graph-only baseline. Unimpressive, believable, and clean: `CALLS` edges come from source, not history, so nothing about the exam can leak into them.

Is 22.4% good? It is *plausible*, and plausibility is what matters for a floor. The graph is missing ~73% of its edges (Chapter 4), the labels are noisy (Chapter 3), so a 2-hop walk over a mostly-blind map recovering a fifth of a noisy answer key is the kind of number that is probably true. Too-good numbers are the ones that need fear. One is coming.

The 94.9% that wasn't

The baseline's second mode adds the other Stage 1 signal: also predict every file connected to the seed by a `CO_CHANGED` edge. First measurement:

	recall	precision	hit rate
call-graph only	22.4%	12.2%	28.8%
call-graph + `CO_CHANGED`	94.9%	7.2%	97.5%

The first combined measurement. A 4× jump in recall from adding one signal. Numbers like this are not gifts; they are alarms.

A 4× jump from one extra edge type is not a capability difference. It is a leak. Walk the logic slowly, because spotting this pattern is a transferable professional skill. The exam's answer keys (Chapter 5) are mined from git commits. The `CO_CHANGED` evidence is mined from *the same git commits*. And the containment is airtight, by construction: an exam case exists only if its commit touched ≤10 files (`eval_max_files`), and any commit that small also clears the ≤20-file evidence cap (`evidence_max_files`). So every exam question's own commit is *guaranteed* to have contributed to the evidence used to answer it. When the scorer asks "what changes with `_client.py`?", the co-change edges already contain a vote cast by the very commit being predicted. The predictor is grading itself with the answer sheet in its lap.

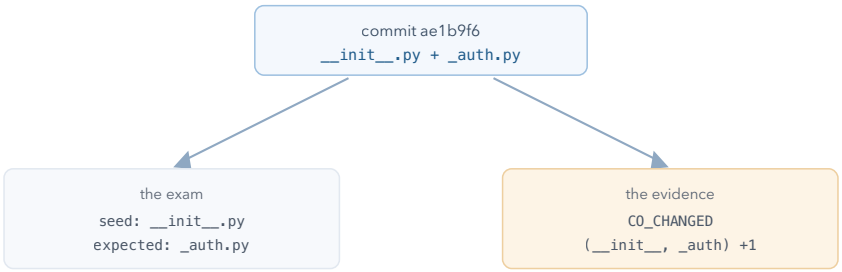


Figure 8.1 · The leak (D012). Every exam case's own commit also contributed co-change evidence, so the combined predictor received partial credit for having seen each answer.

> **IF YOU'RE RUSTY · LEAKAGE, THE ML TERM OF ART**
>
> In machine learning, **leakage** is any path by which information about the test answers reaches the system being tested. Training on your test set is the classic case. It is the most common way published results turn out to be wrong, and the professional reflexes are: (1) be suspicious of any number that jumps too much, (2) trace the *provenance* of every input. Where did this data come from, and does it share an origin with the labels? Here both signals came from git history. That shared parentage was the whole problem.

The fix: leave-one-out

The principled response is not to discard `CO_CHANGED`. Historical coupling that does not depend on any single test commit is real evidence; that was the whole argument of Chapter 3. The response is **leave-one-out scoring**: when scoring the case mined from commit C, first subtract C's own contribution from the evidence, then predict. Each case faces evidence assembled from every commit except its own.

This is where Chapter 6's hoarding pays off. The benchmark pinned the *raw* pair counts, every pair including count-1 singletons, and the subtraction must run against that raw aggregate *before* the `min_count` threshold is applied. Consider a pair seen exactly twice, once by the excluded commit. Filter first and the pair shows count 2 ("passes"); subtract first and it shows 1 ("correctly fails"). Order of operations is the entire fix:

```python
def leave_one_out_neighbors(
    pairs: Counter[tuple[str, str]], seed: str, exclude_files: set[str],
    *, min_count: int = 2,
) -> set[str]:
    neighbors: set[str] = set()
    for (a, b), count in pairs.items():
        if seed not in (a, b):
            continue
        other = b if a == seed else a
        adjusted = count - 1 if other in exclude_files else count
        if adjusted >= min_count:            # threshold applied AFTER subtraction
            neighbors.add(other)
    return neighbors
```

From `gitlog.py`. A commit contributes at most 1 to any pair (files in a commit form a set), so subtracting 1 is exact, never approximate.

And the API is designed so honesty is not optional: `score_baseline(..., use_cochange=True)` *requires* the raw evidence and raises if it is missing, rather than silently falling back to the leaky global edges. The old global-edge path (`predict_impact`) still exists for live, ad-hoc queries and the viewer, where there is

no exam to leak into, and its docstring says, in bold, *do not use this for benchmark scoring*. Same capability, two entrances, the dangerous one labeled.

The fix barely moved the number, and that was the real clue

Re-measured with leave-one-out:

	recall	precision	hit rate
call-graph only (unaffected)	22.4%	12.2%	28.8%
+ `CO_CHANGED`, leaky	94.9%	7.2%	97.5%
+ `CO_CHANGED`, leave-one-out	92.1%	5.9%	97.0%

D012's re-measurement. The leak was real, yet removing it moved recall by only 2.8 points. When a fix barely moves a number, the number was mostly something else.

A drop of 2.8 points is not the collapse a "was leaking" diagnosis implies, and the project's response to that anomaly is the most professional move in the whole log: *it checked why before trusting either number.* Directly measured: of the 510 (seed, expected-file) pairs in the eval set with any co-change evidence, 86.5% are supported by commits other than the one being scored. httpx's coupling is concentrated and recurring (that `_client.py` ↔ `_models.py` pair appears 39 times), so removing one commit's vote rarely changes the verdict. Only 69 single-commit flukes died with the fix. Same-commit leakage was real, but never the dominant driver.

So why is 92.1% still absurd? The answer came from the discipline this book flagged in Chapter 3: *always ask for the prediction size next to any recall figure.* Mean prediction size of the leak-free combined predictor: **40.5 files per case**, against a ~60-file corpus. Two-thirds of the repository, every time. (Max observed: 74, which is more files than httpx has Python files, since the prediction set never bothered restricting itself to `.py`.) A predictor that returns most of the codebase scores high recall no matter what it "found," for the same reason a broken clock is right twice a day.

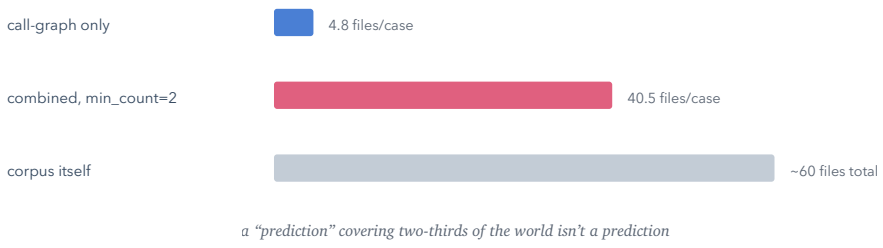


Figure 8.2 · The budget problem behind the 92.1%. Recall was earned by volume, not by finding anything specific.

The right knob, found by sweeping the wrong one

Which parameter reins the budget in? The original suspicion (recorded in D010 when the question was first raised) pointed at `evidence_max_files`: cap the big commits harder and the noise should fall. The sweep said otherwise. The real lever is `min_count`, the "how many times must a pair recur before it counts" threshold from Chapter 5:

min_count	recall	precision	mean predicted size
2 (shipped default)	92.1%	5.9%	40.5
5	74.3%	11.4%	21.9
10	56.7%	20.2%	11.1
20	32.6%	19.6%	6.0
24-30 (plateau)	26.9%	15.7%	5.1

D010's `min_count` sweep (hops=2, leave-one-out, 236 cases). The curve plateaus at about 24 because httpx simply runs out of pairs recurring more often (max observed count: 39).

Now the comparison can finally be made fair. The call-graph baseline predicts 4.8 files per case; at `min_count` around 25 the combined predictor predicts 5.1, the same budget. And at that matched budget it scores 26.9% recall / 15.7% precision against the floor's 22.4% / 12.2%. Better on both axes: a genuine, modest, undisputable win for adding co-change. This is the **matched-budget comparison**, the methodological centerpiece of the entire project. Two predictors may only be compared at similar prediction sizes, because recall can always be bought with volume.

The sweep closed two loops with one more table (crossing both parameters): once `min_count` is tuned, all four tested values of `evidence_max_files` converge to nearly identical results. The originally-suspected knob barely matters in the regime worth quoting from. And a bug in an early draft of the sweep is confessed in the log: it averaged prediction size over unique seeds instead of per case, reporting 1.6 instead of 4.8. That is the same class of weighting error the recall means were already carefully avoiding. Even the meta-measurement got measured.

> **ASIDE · WHY THE SHIPPED DEFAULTS DIDN'T CHANGE**
>
> Given all this, you might expect `min_count`'s default to become 25. It stayed 2, deliberately. The default serves `co_change_edges()`'s original customer, the impact viewer, whose job is showing a human everything plausibly coupled. Silently raising it would hide real coupling from a person to flatter a metric they aren't computing. The scoring path takes `--min-count 25` explicitly. One knob, two purposes, two values, and the distinction written down so nobody "fixes" it later.

Where the floor stands

predictor (all deterministic, no LLM)	recall	precision	files/case
call-graph only, hops=2: *the honest floor*	22.4%	12.2%	4.8
+ co-change, min_count=2: *not citable (budget)*	92.1%	5.9%	40.5
+ co-change, min_count=25: *matched-budget result*	**26.9%**	**15.7%**	**5.1**

Stage 2's ledger. Everything later in the book must beat this table or explain itself.

Notice what this chapter never needed: a language model. Stage 2 is pure Cypher, a `Counter`, and arithmetic, and it has already survived two assassination attempts on its own credibility. When people ask what "evaluation-driven development" means in AI engineering, this chapter is the answer. The evaluation harness did not merely score the system; it found two design flaws (a leak and an unbounded

budget) that code review had missed, because those flaws lived in the *relationship between data sources*, not in any single function.

> **TRY IT**
>
> Reproduce every number in this chapter:
> `uv run labgo baseline benchmarks/httpx --hops 2 --no-cochange` gives the floor.
> `uv run labgo baseline benchmarks/httpx --hops 2 --cochange` gives the seductive 92.1% (watch the empty-predictions count drop to ~1, suspicious in itself).
> `uv run labgo baseline benchmarks/httpx --hops 2 --cochange --min-count 25` gives the fair 26.9%.

PART IV

The Librarian

Stage 3: every function becomes a point in a 1,024-dimensional space, the “vectors can’t do impact” claim finally gets a number instead of an assertion, and an inflated result gets caught by the same trick that caught the last one.

9

Turning Code into Coordinates

What an embedding actually is, why LabGo embeds source code instead of docstrings, and how 1,229 functions became searchable-by-meaning for 162,884 tokens.

Embeddings, from zero

Chapter 2 asked you to accept a miracle on credit: texts go in, vectors come out, and similar meanings land near each other. Time to pay off the credit.

An **embedding model** is a neural network trained on an enormous amount of text to produce, for any input, a fixed-length list of numbers: a vector. LabGo's chosen model emits 1,024 numbers per input, so every function in httpx becomes a point in a 1,024-dimensional space. The numbers individually mean nothing a human can read; what was trained into the model is the *geometry*. Inputs that occur in similar contexts, or say similar things, are pushed toward nearby points. "Nearby" is measured by **cosine similarity**: the cosine of the angle between two vectors, $+1$ for pointing the same way, 0 for unrelated, -1 for opposite. Two functions can share no identifiers and still sit close, because the model has seen ten million retry loops and knows one when it reads one.

> **IF YOU'RE RUSTY · COSINE, IN ONE BREATH**
>
> For vectors a and b, cosine similarity is their dot product divided by the product of their lengths: $(\Sigma a_i b_i) \;/\; (|a||b|)$. Dividing by the lengths makes it a pure *direction* comparison, so a long-winded function and a terse one can still point the same way. That is the entire formula behind the word "semantic" in semantic search. Everything else is plumbing to compute it fast over thousands of stored vectors, which is what a *vector index* is for.

One asymmetry matters in practice: the model is told whether it is embedding a *document* (something to be found) or a *query* (something searching). LabGo passes `input_type="document"` when indexing functions and `input_type="query"` at search time. The model shapes the geometry slightly differently for each role, and mixing them up quietly degrades retrieval.

Choosing the model: the ceiling argument

The embedding model is `voyage-code-3`: hosted, commercial, and code-specialized. The original plan was a small local model via Ollama, keeping the project free and offline. D011 reverses that, and the striking part is the reasoning. It is not just "the laptop is resource-constrained" but a point about *experimental design*. Stage 3 exists to measure what vector search's ceiling looks like against the graph baseline. Run that experiment with a weak embedding model and a poor result is ambiguous. Is vector search the wrong tool, or was the tool just cheap? Using a strong, code-specialized model makes the comparison mean something: if vectors lose at their best, the loss is informative. When you design a comparison, handicapping the side you expect to lose is how you fool yourself politely.

What to embed: the 19.8% audit

The pre-Stage-3 plan was to embed docstrings, the natural-language descriptions attached to functions. Before building it, the author ran one cheap query against the corpus: how many functions actually have one?

> **THE TRAP THAT DIDN'T HAPPEN · D014**
>
> On httpx: 225 of 1,134 functions, **19.8%**, carry a non-empty docstring. A docstring-only index would leave four out of five Function nodes with no vector at all. Invisible to search, not because vector search failed on them but because there was never anything to embed. Any later "vector recall" number would have been, in the decision log's phrase, *"a documentation-coverage measurement wearing a retrieval-quality costume,"* which is the D005 wrong-denominator disease in new clothes. Cost of the audit: one query and five minutes, spent *before* the wrong index was built instead of three days after. Auditing the input distribution before building on it is one of the highest-leverage habits in applied AI.

So LabGo embeds each Function and Class node's *source code*, which every node has, and which a code-specialized model was built for anyway. The mechanism is a quiet callback to Chapter 4: the AST recorded each node's `lineno` / `end_lineno`, so the embedder just reads those lines from the corpus on disk:

```python
def read_source(repo: Path, path: str, lineno: int | None, end_lineno: int | None) -> str | None:
    if lineno is None or end_lineno is None:
        return None
    try:
        lines = (repo / path).read_text(encoding="utf-8", errors="replace").splitlines()
    except OSError:
        return None
    snippet = "\n".join(lines[lineno - 1 : end_lineno]).strip()
    return snippet or None
```

From `embed.py`. The graph stores line ranges, never source text. The corpus remains the single copy of the code, and the graph stays small.

The indexing run

`embed_nodes()` queries Neo4j for every Function/Class node's metadata (after `labgo load`, the database is the canonical copy, so there is no second file to keep in sync), reads each node's source, and ships batches of 100 texts to the Voyage API. The batch size is an engineering-with-a-comment choice: the API caps a call at 1,000 texts or 120K tokens; single functions rarely exceed a few hundred tokens, so 100 per call stays comfortably clear without token-counting machinery. Returned vectors are written straight onto the nodes as an `embedding` property, and the index that makes them searchable is one idempotent statement:

```python
session.run(
    f"CREATE VECTOR INDEX {INDEX_NAME} IF NOT EXISTS "
    f"FOR (n:Node) ON (n.embedding) "
    f"OPTIONS {{indexConfig: {{"
    f"`vector.dimensions`: {EMBED_DIM}, "                  # 1024
    f"`vector.similarity_function`: 'cosine'}}}}"
)
```

One native Neo4j vector index, on the generic `:Node` label from Chapter 7, covering Functions and Classes without an index per label. Nodes with no `embedding` property (Files) are simply absent from it. Confirmed, not an error.

The run on httpx: 1,229 candidates, 1,229 embedded, 0 skipped, 162,884 tokens billed, comfortably inside the model's 200-million-token free tier. The vectors and the graph now live in the same database, which is about to matter architecturally: queries can mix traversal and similarity in a single Cypher statement.

Searching is the same trip in reverse. Embed the query (`input_type="query"`), ask the index for the `k` nearest nodes:

```
vector = client.embed([query], model=EMBED_MODEL, input_type="query").embeddings[0]
rows = session.run(
    "CALL db.index.vector.queryNodes($index, $k, $vector) "
    "YIELD node, score "
    "RETURN node.id AS id, node.path AS path, score",
    index=INDEX_NAME, k=k, vector=vector,
)
```

`semantic_search()`: the librarian as four lines. No graph traversal, no LLM. (Neo4j logs a deprecation notice for this procedure; D014 records leaving it until the replacement is actually needed. Swapping syntax ahead of need is churn, not progress.)

ASIDE · THE WEIGHT OF ONE DEPENDENCY

Installing `voyageai` transitively drags in `langchain-core`, `tokenizers`, `huggingface-hub`, `numpy`, `pillow`, and more. A startling haul for "one embeddings API call." D014's response: quarantine it in an optional install (`uv sync --extra vectors`), and import it lazily inside the two CLI commands that need it, so every other command works on a bare install. The project applies the same pattern to `langgraph` / `anthropic` (Stage 4), `mcp` (Stage 5), and `opentelemetry-sdk` (Stage 6). Four optional extras, one discipline: the core pipeline must never grow a hard dependency because a later stage wanted a convenience. Check where the lint suppression comment sits in `cli.py`: even the exception is documented.

TRY IT

`uv sync --extra vectors`, put a `VOYAGE_API_KEY` in `.env`, then
`uv run labgo embed ../labgo-corpora/httpx` and
`uv run labgo search "retry a request with exponential backoff"`.
The results are ranked function ids with cosine scores. Notice it finds retry logic whether or not the word "retry" appears, and notice equally that nothing about the results tells you what *depends on* any of them.

10

The Librarian, Measured

D001's founding claim finally faces the exam, but only after a 75.5% recall is exposed as the budget illusion in a new costume. Then a naive union of two mediocre signals nearly doubles the floor, and earns its keep by catching different truths.

A module with a pointed absence

Stage 3b lives in `hybrid.py`, and its first design decision is visible in what it refuses to import: no `voyageai`, anywhere. Prediction reuses the vectors already stored on the nodes via `db.index.vector.queryNodes`. The seed's functions already have embeddings, so finding their neighbors needs Neo4j alone. Scoring the benchmark 236 times therefore costs zero API calls, and the module even duplicates the `INDEX_NAME` constant rather than importing it from `embed.py`, specifically so importing `hybrid` never pulls in the heavyweight dependency transitively. A one-line duplication, a comment explaining it, and a boundary that holds.

The wrong 75.5%, and the fix that defines k

The vector predictor's job: given a seed file, return files whose code resembles the seed's code. The first implementation did the obvious thing. For each function in the seed file, fetch its top-`k` nearest neighbors, and union all of them into the prediction. It scored 75.5% recall.

> **THE TRAP, THIRD APPEARANCE · D015**
>
> 75.5% would have been the headline result *disproving* D001. Vectors triple the graph baseline! Except the discipline installed in Chapter 8 fired first: check the prediction size. A seed file like `_client.py` contains dozens of functions; each contributed its own top-`k` list; the union averaged 17.4 files per case (max 36), nearly 30% of the 60-file corpus, versus the call-graph baseline's 4.1. The broken clock again, wearing vectors this time. Same disease as D012, different mechanism: there it was leakage; here, an unbounded prediction size. The log's phrasing is worth keeping. The measurement was *"wrong in the informative direction."* It looked like a triumph, and the routine of checking the budget converted it into a diagnosis within an hour instead of a retraction within a month.

The fix makes `k` mean something disciplined: rank candidate *files* by their single best-matching function score, and cap the whole prediction at `k` files, however many functions the seed contains. That makes `k` the vector predictor's analogue of `hops`: a bounded retrieval budget, not an accident of file size:

```
result = session.run(
    "MATCH (:File {id: $seed})-[:CONTAINS]->(fn) "
    "WHERE fn.embedding IS NOT NULL "
    "CALL db.index.vector.queryNodes($index, $k, fn.embedding) YIELD node, score "
    "WHERE node <> fn "
    "MATCH (f:File)-[:CONTAINS]->(node) "
    "WHERE f.id <> $seed "
    "WITH f, max(score) AS best "
    "ORDER BY best DESC LIMIT $k "
    "RETURN f.id AS file",
    seed=seed, index=INDEX_NAME, k=k,
)
```

`predict_impact_vector()`: graph pattern and vector search cooperating in one statement. Hop from the seed to its functions, query the index from each function's own stored vector, hop from neighbors to their files, keep each file's best score, cut at `k`.

Two guards in the `WHERE` clauses carry the intellectual honesty: a node is never its own neighbor, and same-file matches don't count. In the docstring's words, resembling your own sibling isn't evidence of anything.

The claim, checked

Sweeping the budget on the fixed predictor:

k	mean predicted size	recall	precision	hit rate
3	~6.5	17.1%	13.1%	24.6%
4	~7.5	19.3%	13.2%	28.0%
5	~8.3	22.3%	12.4%	31.8%
10	~8.5	45.5%	9.6%	61.0%
20	~9.6	60.7%	7.0%	73.3%

Vector-only prediction vs. budget (D015). Recall can be bought (60.7% at `k=20`) but the price is paid in precision, now on purpose and disclosed instead of by accident.

The fair reading happens at matched budget: vectors at `k=4` (~7.5 files) score 19.3% / 13.2% against the call-graph's 22.4% / 12.2% (~4.8 files). At comparable budgets, the graph holds. D001's founding scope decision, that transitive impact is a question vector search can't answer, spent three stages as well-reasoned assertion. It is now a measurement. Not a landslide, and it shouldn't be: resemblance correlates with coupling often enough to score points. But correlation is what it is, and the graph, blind to 73% of its own edges, still finds dependency better than similarity does.

The union that earned its keep

Then the interesting question: are the two signals finding the *same* fifth of the truth, or *different* fifths? `predict_impact_hybrid()` answers with the crudest possible instrument, a set union. No weighting, no ranking, deliberately naive: *"if this doesn't beat the call-graph floor, a smarter blend isn't worth building yet; if it does, how much it helps is the number that justifies building one."*

method (hops=2, no cochange)	k	recall	precision	hit rate
calls only	n/a	22.4%	12.2%	28.8%
vectors only	4	19.3%	13.2%	28.0%
hybrid (union)	**4**	**40.1%**	**14.8%**	**53.8%**
hybrid (union)	10	54.0%	9.4%	69.1%

D015's headline. At k=4 the union nearly doubles recall over either signal alone, with precision *improving*. Union size (~8.1 files) was checked before celebrating; this one is real.

Stop and appreciate how strange 40.1% is. Two predictors scoring 22% and 19% union into 40%, which is possible only if their correct answers barely overlap. Each signal catches true positives the other structurally cannot: the call graph finds the dependent that shares no vocabulary with you; the vectors find the parallel implementation the parser couldn't link. And precision *rising* in the union (14.8% against 12.2% and 13.2%) means their false positives overlap even less. This is Chapter 2's "wrong differently" argument graduating from rhetoric to arithmetic. It is the first number in the project where combining signals helps *without* an asterisk, and it was verified by checking the union's budget before believing it.

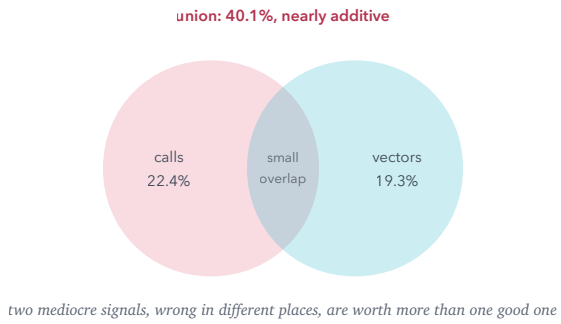


Figure 10.1 · Why the union nearly doubles: the two signals' true positives barely overlap, so their recalls almost add.

Adding the third signal, now that it's safe

One signal is still on the bench: co-change. It was excluded from D015's original hybrid because at the time (before D012's leave-one-out fix landed) there was no honest way to score it. Once there was, the follow-up measurement asked whether it adds real lift on top of the union, at the matched `min_count=25`:

method (hops=2, k=4)	recall	precision	mean size
calls ∪ vectors	40.1%	14.8%	8.1
calls ∪ vectors ∪ cochange (loo, min_count=25)	**43.3%**	**15.4%**	**8.3**

The full three-signal hybrid, the project's best citable retrieval number. +3.2 recall points for +0.2 files of budget, precision improving: real, modest, honestly measured.

The fine print is characteristically candid: the lift holds at `k=3`, and by `k=6` the two rows converge *exactly*. A wide-enough vector net catches the same handful of files that thin, high-confidence co-change evidence contributes. Signals overlap more as budgets grow; marginal value is a function of budget, not a constant.

The scorer that refused to move

A quiet detail from this stage deserves loud praise. Adding `--method vectors|hybrid` to `labgo baseline` meant touching the code path that produced Chapter 8's numbers. So the default path was kept byte-for-byte identical, and then re-run: 22.4% recall, 28.8% hit rate, digit-for-digit. When you refactor the instrument, you re-measure the standard before trusting the new readings. It costs one command and it is the difference between a benchmark and a memory of one.

And the deferral, one more time, now a pattern with a name: the union is unweighted. A file predicted by all three signals counts no more than a file predicted by one. Building a ranker (vote-counting, cosine tie-breaks) is real engineering with no measurement yet showing the naive union is the bottleneck. Every number so far says retrieval *recall* is. Same posture as D004's type inference: named, scoped, and parked until evidence calls for it. The decision log calls this out explicitly rather than leaving it silent, which is the difference between a backlog and an alibi.

> **TRY IT**
>
> `uv run labgo baseline benchmarks/httpx --method vectors --k 4`
>
> `uv run labgo baseline benchmarks/httpx --method hybrid --hops 2 --no-cochange --k 4`
>
> `uv run labgo baseline benchmarks/httpx --method hybrid --hops 2 --cochange --min-count 25 --k 4`
>
> Then sweep `--k` yourself and watch the recall/precision trade move just as the tables promise. No API key needed; the vectors are already in the graph.

PART V

The Apprentice

Stages 4 and 5: the first LLM enters the system, as a router choosing among the tools every earlier stage already measured. It loses to the deterministic baseline, the loss is diagnosed to a single missing constraint, and then the same five tools get turned inside out for the rest of the world to call.

11

What an Agent Actually Is

Strip the mystique: an agent is a loop in which a language model chooses tools, sees their output, and decides what to do next. Here is that loop, built visibly, in 449 lines you can read.

Tool use, mechanically

Everything hinges on one API capability. When you call a modern LLM, you may attach a list of **tools**: each a name, a natural-language description, and a JSON schema for its parameters. The model cannot execute anything. It emits a structured message that says, in effect, "I choose to call `co_change_neighbors` with `{"file": "httpx/_client.py", "min_count": 20}`." Your code runs the real function, appends the result to the conversation, and calls the model again. The model reads the result and either requests another tool or writes its final answer.

That loop (model proposes, harness executes, result returns, repeat) is the entire trick behind the word "agent." Everything else is engineering around its failure modes. It can loop forever, so you bound the turns. It can ask for nonsense arguments, so you catch errors and feed them back as text. It can produce an answer in the wrong shape, so you parse defensively. LabGo's Stage 4 confronts all three, visibly.

> **IF YOU'RE RUSTY · WHY DESCRIPTIONS MATTER MORE THAN SCHEMAS**
>
> The model chooses tools by reading their *descriptions*, the way you choose a function from library docs. Vague description, bad routing. LabGo's tool descriptions are miniature literature reviews. The co-change tool's literally says: *"catches real coupling the call graph misses … but is noisy. Raise min_count for a stronger, cleaner signal (D010 found min_count~25 matches th project's call-graph budget)."* The measurement campaign of Parts II through IV is being handed to the model as operating instructions. Stages 1 through 3 didn't just produce numbers; they produced the prompt.

Five tools: three measured, two new

The agent chooses among five tools. Three wrap the same signals this book has already measured one by one: `call_graph_traverse` (Chapter 8), `co_change_neighbors` (Chapters 5 and 8), `semantic_search` (Chapters 9 and 10). Two are new, and they complete the project's headline question, covering the "which tests" and "who reviews" parts nothing before Stage 4 answered:

- `test_coverage`: a Cypher hop across the `TESTS` edges Chapter 4 quietly emitted. Which test functions exercise functions in this file?
- `likely_reviewer`: not a graph query at all. `git log` on the file, count authors, return the top three committers. The docstring is candid, "not code-aware, just git log." A heuristic wearing its limits on its sleeve.

Each tool function takes the Neo4j driver (or the repo path) plus a plain dict of arguments and returns a JSON string. Note the shape: these are ordinary functions with boring signatures. That design choice pays off twice, first for testing, and again in Chapter 13 when a second, completely different caller wires them up unchanged.

LangGraph, thinly

The loop is orchestrated with **LangGraph**, a library for expressing agent workflows as explicit state machines: named nodes (functions that update a shared state), edges (which node runs next), and conditional edges (a function inspects state and picks the next node). You could write this particular loop with a `while` statement, and the README's stack table is upfront that LangChain is "used thinly." But the state-machine form makes the control flow inspectable and gives the routing logic a place to live as data rather than as scattered `if`s.

What LabGo pointedly does *not* use: `create_react_agent`, the prebuilt LangGraph agent loop, or `langchain_anthropic`'s model wrapper. The model call is the raw `anthropic` SDK, and state carries Anthropic's own message JSON unchanged. No translation layer, nothing serialized twice. The reasoning traces straight to D001: *the routing decision is the deliverable.* A prebuilt agent hides the very behavior this stage exists to inspect. When the point of your project is to examine how a model routes, importing a black box that routes for you is self-defeating. The general lesson: frameworks are for the parts you don't need to understand, and which parts those are depends on why you are building.

```python
class AgentState(TypedDict):
    messages: Annotated[list[dict[str, Any]], operator.add]   # append, don't replace
    turns: int
    max_turns: int
    stop_reason: str | None
    input_tokens: int          # running cost, accumulated per call
    output_tokens: int
    tool_calls: list[str]      # audit trail of every tool the model chose


def _route_after_agent(state: AgentState) -> str:
    if state["stop_reason"] == "tool_use":
        return "finalize" if state["turns"] >= state["max_turns"] else "tools"
    return END
```

The whole nervous system: state is a dict; routing is a three-line function. `Annotated[..., operator.add]` tells LangGraph that when a node returns new messages, they are appended to history rather than replacing it.

Three nodes, wired `agent → (tools | finalize | END)`, with `tools → agent` closing the loop:

- **agent** calls the model with the system prompt, tool schemas, and conversation so far; appends the reply; accumulates token counts.
- **tools** executes every tool request in the last reply. Exceptions are caught and returned to the model as JSON error text, not raised. A model that passes a bad argument should get the chance to read the error and correct itself, not crash the run.
- **finalize** is the escape hatch. When the model is still requesting tools at the turn budget (`MAX_TURNS = 6`, with a comment worth framing: "an unbounded tool-calling loop is a real failure mode, not a hypothetical one"), this node sends one last request *with no tools attached*. The model physically cannot ask for another call and must answer. Termination is guaranteed by construction, not hoped for.

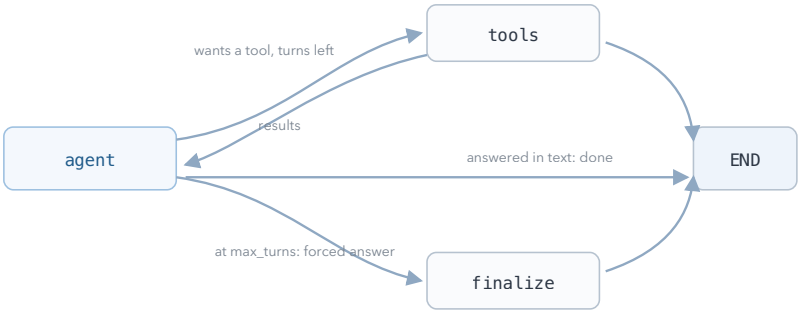


Figure 11.1 · The compiled state machine in `agent.py`. The `finalize` path makes non-termination impossible: the final call carries no tool schemas, so a text answer is the only legal move.

Grounding, and refusing to score spelling

One more design decision completes the picture, and it is easy to miss how much rides on it. The system prompt begins by listing every file id in the corpus (60 files, small enough to afford). Without that grounding, the model would have to produce path strings from memory, and the benchmark would silently measure *its spelling of paths* rather than its reasoning about impact. Correspondingly, the output parser refuses to meet the model halfway: `parse_impacted_files()` keeps only mentions that exactly match a real file id, drops the rest into an `unrecognized_mentions` field, visible in the result, excluded from scoring, never fuzzy-matched. If the model wrote something almost-right, that fact is preserved as data instead of being silently repaired into a point. (The docstring also names the scaling limit honestly: a large corpus couldn't list every file in the prompt and would need a `list_files` / `search_files` tool. Not built, same restraint-until-binding rule as D004.)

The required output format is plain text, not JSON: four labeled lines (`IMPACTED_FILES:`, `TESTS_TO_RUN:`, `LIKELY_REVIEWER:`, `SUMMARY:`) pulled out with regular expressions. For a reply this flat, labeled lines are easier for a small model to emit reliably than nested JSON, and the failure mode (a stray character) degrades one line rather than un-parsing the whole reply. That choice is about to be stress-tested in the next chapter. Watch for the stray `**`.

> **TRY IT**
>
> `uv sync --extra agents`, set `ANTHROPIC_API_KEY`, then ask one question:
> `uv run labgo agent "httpx/_client.py" ../labgo-corpora/httpx`
> The CLI prints the model's report plus the run stats: which tools it called, in what order, how many turns, and what it cost in tokens. Run it twice and note the routing isn't deterministic. You are watching a model make choices.

12

The Agent Doesn't Win

30.3% recall, 5.8% precision: the LLM loses to a Cypher query and a Counter. Why publishing that number, with its diagnosis, is worth more than any demo that hides it.

The measurement

Stage 4 ends the way every stage ends: against the benchmark. With one honest wrinkle. Running 236 cases through a multi-turn LLM loop costs real money and real time, so `labgo agent-eval` scores a *sample*: 40 cases, selected with a fixed random seed (`--sample-seed 42`) so the sample is reproducible. A disclosed, deterministic sampling choice, in the same spirit as every filter in Chapter 5. Visible in the method line, not buried. Results, on Haiku 4.5 (a fast, inexpensive model tier), `max_turns=6`:

predictor	recall	precision	hit rate	files/case
call-graph floor (Ch. 8)	22.4%	12.2%	28.8%	4.8
hybrid + cochange, matched budget (Ch. 10)	43.3%	15.4%	57.2%	8.3
Stage 4 agent (n=40)	**30.3%**	**5.8%**	**45.0%**	**13.05**

D016. The agent's recall lands between floor and hybrid; its precision is worse than the floor's. It is not close.

Read the table the way a reviewer would. Recall: middling. The agent finds more than raw traversal, less than the tuned union. Precision: 5.8%, worse than the free, zero-intelligence floor's 12.2%. Nineteen of every twenty files the agent names are wrong. By D001's own standard (does the added complexity earn its place?) the decision log answers in its own words: *"the honest answer right now is no."*

The diagnosis: an uncalibrated budget, not a dumb model

Here the training of the last four chapters pays off instantly. You already know the first question to ask: *what was the prediction size?* 13.05 files per case, 57% above the hybrid's 8.3, nearly triple the floor's 4.8. Now look at the shape of the whole ledger. Every deterministic method carries an explicit, tunable size cap: `hops` bounds the walk, `k` bounds the vector list, `min_count` bounds the evidence. The agent has no such knob anywhere. The system prompt explains the tools' trade-offs beautifully and never once asks for a bounded answer. So the model, diligent and unpriced, gathers evidence across turns (the transcript shows `co_change_neighbors` called 51 times over 40 cases, often re-called with different `min_count` values to explore) and then lists essentially everything it gathered.

The decision log's verdict deserves quoting because it refuses the lazy headline: *"this is not a reasoning failure, it's an uncalibrated budget,"* structurally the same mistake as D015's 75.5% and D012's 40.5-file predictor, *"an otherwise-plausible number produced by predicting too much, not by finding more."* Three appearances, three costumes, one disease. And note which claims the evidence licenses. "The agent doesn't beat the baseline, because it isn't budget-constrained" is supported. "LLMs are bad at this" is not. Those are different sentences, and only the first has data behind it.

What it cost

Because the loop counts tokens per run (and Chapter 14 will wrap every call in spans), the price of the experiment is a number, not a shrug: 429,839 input tokens, 38,534 output tokens for 40 cases. About 11,700 tokens per question, a mean of 3.05 turns each. On Haiku pricing that is pennies; the sting is not the bill, it is what the bill bought: a worse answer than a free Cypher query. That sentence, cost attached to quality delta, is what "evaluating an AI feature" means in production engineering, and almost nobody can produce it for their demo.

A small secondary finding rounds out the eval: in 3 of 40 cases the model leaked markdown bold markers (`**`) into the `IMPACTED_FILES:` line despite the plain-text instruction. The strict parser did its job. The mangled entries landed in `unrecognized_mentions`, costing the agent a little recall on those cases but never corrupting a score. Defensive parsing turned a formatting quirk into a measured footnote instead of a silent bias.

The fix that is named but not built

The obvious repair is written down with a scope and a success criterion: apply D015's matched-budget standard to the agent itself. Instruct it to return its best N files, ranked, and score it against the deterministic baselines at the same N. Until that measurement exists, the log insists, the citable claim stays as-is. It would be a twenty-minute change; leaving it unbuilt in the repository is the pedagogy. The difference between the measured claim and the plausible one is the entire lesson of the project, demonstrated one last time on the most expensive component.

> **ASIDE · WHY A LOSING NUMBER IS A STRONG PORTFOLIO**
>
> The project's stated purpose is evidence for an AI-architect interview. Imagine the two candidates. One says: "I built a multi-agent RAG system with LangGraph and a knowledge graph." The other: "My agent scored 30.3/5.8 against my deterministic baseline's 43.3/15.4. Here's the leakage I caught in the benchmark first, here's the budget analysis showing why the agent over-predicts, and here's the bounded-answer experiment that would settle it." The second candidate has demonstrated the rarer skill: knowing how to *find out* whether the thing works. Anyone can assemble parts; the market shortage is people who can be trusted to measure them.

> **TRY IT**
>
> `uv run labgo agent-eval benchmarks/httpx ../labgo-corpora/httpx --n 40 --sample-seed 42` reproduces the table (real tokens, real pennies). Pass `--out transcript.json` to save every case's full conversation. Reading two or three transcripts, watching the model reason about which tool to trust, is the fastest way to build intuition for what these systems actually do.

13

The Same Tools, Turned Inside Out

Stage 5 writes almost no new logic and changes something fundamental anyway: who is driving. MCP, the USB-C of AI tools, and LabGo's graph plugged into Claude Code.

The direction of control

Look at Stage 4 as a diagram: LabGo's own loop calls LabGo's own model, which chooses among LabGo's tools. Everything inside one process, one owner, one API bill. Stage 5 asks: what if the model isn't ours? Claude Code in a terminal, Claude Desktop on a laptop. Each has its own model and its own agent loop, and each would benefit from asking LabGo's graph the questions this book has been asking. What is needed is a standard way for any AI application to discover and call someone else's tools.

That standard is the **Model Context Protocol** (MCP), introduced by Anthropic in 2024 and since adopted broadly across the industry. The analogy that stuck is USB-C: one connector, any device. An MCP *server* exposes **tools** (functions the client's model may call), **resources** (data the client may read), and prompts. An MCP *client*, the AI app, connects, asks "what do you offer?", and lets its model call what it finds. The wire format is JSON-RPC; for a local server like LabGo's the transport is simply stdin/stdout of a subprocess the client launches.

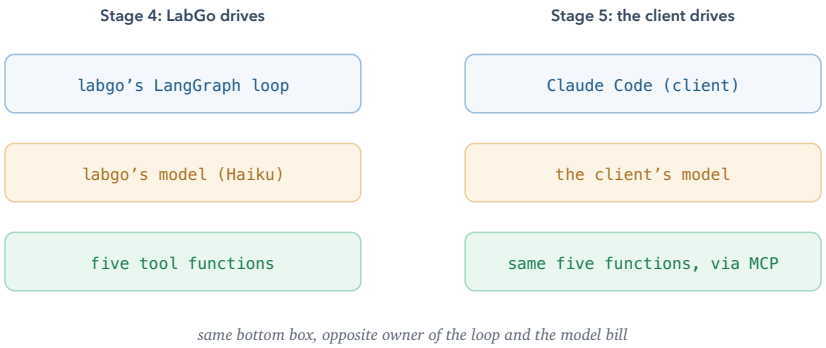


Figure 13.1 · The inversion. Stage 4: an LLM this project drives calls these tools. Stage 5: this project's tools are offered to any client's own model.

A server with no logic of its own

`mcp_server.py` is 99 lines, and its docstring leads with a boast most modules would hide: *it owns no business logic.* Stage 4's tool functions were deliberately given public names and dumb signatures; Stage 5 wraps each in a decorated function whose typed signature becomes, automatically, the JSON schema advertised to clients:

```
server = MCPServer(
    name="labgo",
    description="Change-impact analysis over a code graph: call graph, co-change "
    "history, semantic search, test coverage, and likely reviewers.",
)


@server.tool()
def call_graph_traverse(file: str, hops: int = 2) -> str:
    """Files reachable via the static CALLS graph within `hops` of a seed file."""
    with tracing.span("mcp.tool_call", tool="call_graph_traverse"):
        return tool_call_graph(driver, {"file": file, "hops": hops})
```

One of five wrappers in `build_server()`. The driver is captured by closure, opened once at process start, because an MCP server is a long-lived process, unlike the CLI's one-shot commands.

Alongside the five tools, the server exposes one *resource*: `labgo://files`, the list of every file id in the corpus. The distinction is worth internalizing, because designing good MCP surfaces is becoming a real skill. A tool is an *action* the model invokes repeatedly with arguments; a resource is *context* the client reads once on its own terms. The file list is the same grounding Stage 4 baked into its system prompt (Chapter 11). Same information, redelivered in the protocol's native shape for context rather than action.

Verified over the real wire

The test suite's Stage 5 coverage makes a distinction every integration author should steal: it does not merely call the five wrapped functions directly (which would test Stage 4's logic again and prove nothing about the protocol). The README records verification against a real client over the actual stdio JSON-RPC protocol: handshake, tool listing, tool calls, resource reads. The wiring is thin, but thin wiring still has two ends, and only an end-to-end exchange proves they meet. To plug it into Claude Code yourself, register the command and start asking impact questions in plain English:

```
claude mcp add labgo -- uv run labgo mcp ../labgo-corpora/httpx
```

From that moment, a Claude Code session can call `call_graph_traverse` or `likely_reviewer` mid-conversation. The client's model reads the same tool descriptions Chapter 11 dissected and routes with them. The five tools this book spent four Parts measuring are now a capability any MCP-speaking assistant picks up for free. Which is, quietly, the strongest argument for the "boring functions with dumb signatures" design: logic written once, measured once, served to two entirely different masters unchanged.

> **ASIDE · MCP IN THE WIDER WORLD**
>
> The same pattern powering this 99-line file powers production integrations everywhere: filesystem and database servers, browser automation, issue trackers, design tools. The economics are the point. Before MCP, every AI app integrated every service pairwise (N×M adapters); with a protocol, each side implements it once. If you build one thing after this book, a small MCP server over data you care about is the highest-leverage exercise. Everything you need, you have now seen.

PART VI

Proof and Polish

Stage 6 and the viewer: making the system watchable. Real OpenTelemetry traces with cost and latency on every span, a CI pipeline that knows exactly what it refuses to pay for, and an interactive map you can fly around in a browser.

14

Watching It Run

An agent you cannot see inside is a bill you cannot explain. Stage 6 wraps every model and tool call in real OpenTelemetry spans, and catches one last bug: a "trace" that was actually a pile of strangers.

Observability, the sixty-second version

In modern operations vocabulary, **observability** is the ability to ask new questions of a running system without shipping new code. Its unit of account for request-shaped work is the **trace**: a tree of **spans**, where each span records one operation. A name, start and end times, and arbitrary key-value attributes. All spans in one logical operation share a `trace_id`; each records its parent, forming the tree. The industry standard for producing them is **OpenTelemetry** (OTel): vendor-neutral APIs and SDKs. You emit spans once, and point them at whatever backend you like later.

For LLM systems, tracing is not an ops luxury; it is the only way to answer the questions that actually get asked. Why was this answer slow? Which tool ate the time? What did this run cost in tokens? Did the agent loop, and how many times? Chapter 12 could quote 11.7K tokens per case because this chapter's instrumentation was recording it.

One function, honestly optional

The whole of Stage 6's machinery is `tracing.py`: 59 lines exposing a single context manager. It opens a span, sets your attributes up front, yields the span for later additions, and stamps `latency_ms` on exit no matter how the block ends:

```python
@contextmanager
def span(name: str, **attributes: AttributeValue) -> Iterator[object | None]:
    if not TRACING_AVAILABLE:
        yield None
        return
    start = time.monotonic()
    with _tracer.start_as_current_span(name) as s:
        for key, value in attributes.items():
            s.set_attribute(key, value)
        try:
            yield s
        finally:
            s.set_attribute("latency_ms", round((time.monotonic() - start) * 1000, 1))
```

All of `tracing.py` that matters. `TRACING_AVAILABLE` is set by a `try/except ImportError` at module load, the fourth appearance of the optional-extra pattern (voyageai, mcp, langgraph before it).

If `opentelemetry-sdk` isn't installed (it lives in the `observability` extra), `span()` yields `None` and every caller runs identically, just untraced. Callers must tolerate the `None`; you saw the `if s: s.set_attribute(...)` guards in Chapter 11's code. Cheap ceremony, real payoff: observability that can be absent without a single code path changing is observability that never becomes a load-bearing dependency by accident.

The pile-of-strangers bug

> **THE TRAP, FINAL APPEARANCE · D017**
>
> The first version wrapped each LLM call and each tool call in its own `start_as_current_span()` and called it done. Checked directly (the discipline by now is reflexive), every span had its own `trace_id`. Technically spans; collectively not a trace. A backend would have displayed one agent run as eight unrelated one-span "traces," with no way to reassemble which belonged together. The fix: wrap the whole of `run_agent()` in one root `agent.run` span, so every `agent.llm_call` and `agent.tool_call` opened inside it nests as a child, sharing one `trace_id` with correct parent chaining. Verified against a live run before the entry was marked done. The general form of the lesson: instrumentation is code, and it can be wrong in ways that look exactly like working. The spans printed fine; only their relationships were nonsense. Verify what the telemetry *means*, not just that it appears.

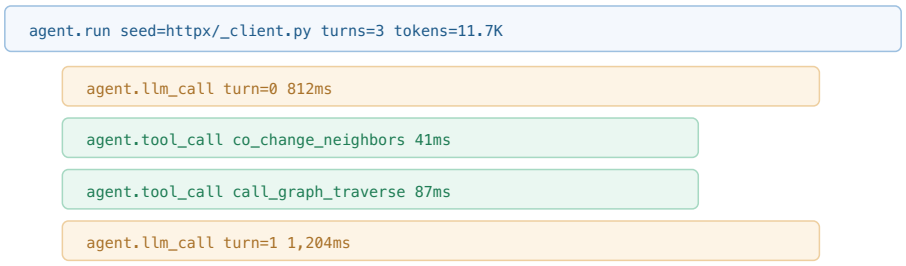


Figure 14.1 · The corrected shape: one root span per agent invocation, every child sharing its `trace_id`. LLM spans carry token counts; tool spans carry latency, so a slow tool is distinguishable from an expensive model call at a glance.

The attribute scheme completes the story. `agent.llm_call` spans carry `input_tokens` / `output_tokens` / `latency_ms`; `agent.tool_call` spans carry latency per tool. `mcp_server.py`'s wrappers (Chapter 13) get the same latency spans but no token attributes, because the model calling those tools belongs to the client, so there are no tokens of LabGo's own to report. Even the telemetry respects the direction of control.

Where the spans go, and the two-line Simple/Batch decision

The spans are exported… to the console. Deliberately. The stack table had listed "Langfuse / OpenTelemetry" as an either/or; D017 chose plain OTel with a console exporter because a Langfuse account is one more API key that can be unavailable at 2am, the exact failure mode the project had already paid for twice. The spans are still real: pointing them at a hosted backend later is one added line (`add_span_processor(...)`), additive rather than a rewrite. And a two-line detail shows the difference between copying an example and understanding one: the exporter is wired through `SimpleSpanProcessor`, not the usually-recommended `BatchSpanProcessor`, because LabGo's commands are short-lived CLI processes, and a batching processor can still be holding spans in memory when the process exits, silently losing the tail of every run. Defaults encode assumptions; this process's lifespan violated one.

CI: the free half, on purpose

Stage 6's other half is a GitHub Actions workflow of studied modesty: `ruff check` (the linter, in its strictest all-rules mode; the pyproject config enables everything and subtracts each ignored rule with a written

reason) plus `pytest`, on every push and pull request. What makes the workflow instructive is the pair of boundaries around it, both verified rather than assumed:

- **Everything in it is free and fast.** Before wiring CI, the author grepped the test suite for live-service constructors (`connect(`, `GraphDatabase`, `anthropic.Anthropic(`, `voyageai.Client(`) to confirm all 46 tests run with zero credentials and zero network. The one hit is a test that deliberately unsets the password and asserts `connect()` raises *before* touching the network. It tests the guard, not the service.
- **Everything not in it is excluded with a reason.** CI does not run `labgo baseline` or `agent-eval`: those need real Neo4j/Voyage/Anthropic credentials (which don't belong in a public repo's secrets for a personal project), and the agent eval spends real tokens per push. The line is drawn where the money starts, and drawn in writing.

The deeper pattern generalizes to every AI project: split your evaluation suite into the free half and the paid half, automate the free half mercilessly, and run the paid half as a logged, deliberate act. A regression in parsing, scoring, or graph logic is caught on every push by the 46 tests. A regression in retrieval quality is caught by re-running the benchmark, which the pinning discipline of Chapter 6 makes reproducible whenever you choose to pay for it.

> **TRY IT**
>
> `uv sync --extra observability`, then run any agent command. Spans print to the console as JSON as the run proceeds. Watch the root span arrive *last* (it closes last) carrying the run's totals. Then `uv pip uninstall opentelemetry-sdk` and run again: identical behavior, silent. That is the no-op guarantee, demonstrated.

15

Seeing the Graph

A force-directed map of your codebase in a browser, with an impact mode that answers the Tuesday question visually, and a serving trick that keeps a committed build honest.

What the viewer is

`labgo view` starts a tiny local web server and opens an interactive visualization of whatever `labgo ingest` last produced. The frontend (`viewer/`) is a React application built around `react-force-graph-2d`, which renders the graph with a force simulation: every node repels every other, every edge pulls its endpoints together, and the layout settles into clusters that end up mirroring the architecture, with no layout algorithm being told anything about the code, because heavily-connected modules literally pull each other close. The color language is the one this book's figures have used all along: blue files, green functions, amber classes; rose `CALLS`, teal `IMPORTS`, violet `TESTS`, amber dashed `CO_CHANGED`.

Sensible defaults carry the pedagogy even here. Of the five edge kinds, only `CALLS` renders initially. `CONTAINS` alone is 1,200+ edges of File→Function scaffolding that would drown the picture (the code comment says exactly this), so structure is opt-in via checkboxes rather than default noise.

Impact mode is the thesis, drawn

The viewer's second mode operationalizes the whole book. Pick any file or function; the viewer highlights (a) everything reachable backwards along `CALLS` edges within an adjustable hop depth, which is Chapter 8's predictor, live, and (b) files historically co-changed with the selection above an adjustable `min_count`, which is Chapter 5's evidence, live. The two signals render separately: different colors, separate sidebar sections, never merged into one undifferentiated blob-of-risk. That is the same separation discipline the measurement chapters enforced, applied to pixels. A human deciding whether to trust a warning deserves to know which kind of evidence is behind it, structural or historical, because Chapter 4 showed the structural signal is only 27% complete and Chapter 5 showed the historical one is noisy. The sliders are, in effect, the `hops` and `min_count` knobs from the CLI. Same trade-offs, manipulable by drag.

That symmetry is newer than it sounds. The hop-depth slider was there from the start; the co-change side had no equivalent, and rendered every edge in `cochange.json` unconditionally, at whatever that file's own baked-in floor of `count ≥ 2` happened to admit. In other words, the viewer shipped for a while showing exactly the unbudgeted picture Chapter 8 spent ten pages refusing to quote. A second range slider, "Co-change ≥", closes the gap: minimum pinned at 2, because that is the data's floor and pretending otherwise would be a lie about the file, and maximum computed from the highest count actually present in whatever was loaded rather than guessed. Moving it filters three things at once, the sidebar list, the node highlighting, and the edge colour and width, because all three read from one memoized selector; filter the selector and the picture stays internally consistent by construction rather than by three matching edits. A hint underneath cites the finding by number, *try ≥25 for the cleanest signal, matched to the call-graph's own budget (D010)*, and the demonstration is the point: select `httpx/_client.py` at the default threshold and 44 co-changed files light up; drag to 25 and exactly one

survives. That is D010's table, converted from a number you have to trust into a thing you can watch happen.

> **THE TRAP · THE ENDPOINT CHECK THAT WENT MISSING**
>
> Wiring the threshold into the edge colour and width functions introduced a bug on the way. The original conditions asked one question, *does this edge touch the selected file?* The first filtered version replaced it with a different one, *is either endpoint in the filtered set?*, and the two are not the same question. An unrelated `CO_CHANGED` edge between two other files could satisfy the second while failing the first, and light up as though it were evidence about your selection. It would have looked entirely plausible on screen, since an amber dashed line is an amber dashed line. The fix keeps both clauses: the edge must touch the selection *and* its other endpoint must survive the threshold. The general shape is familiar by now, from D005's denominator onward. When you add a filter to an existing condition, check whether you added a term or quietly replaced one.

The committed build, and the trick that keeps it honest

A deliberate deployment choice: the compiled frontend (`viewer/dist/`) is committed to the repository, so running `labgo view` requires no Node, no npm, no build step. The Python CLI serves prebuilt static files. Committing build artifacts is usually frowned upon; here it buys a zero-dependency experience for a viewer that changes rarely, and the README documents the regeneration path for anyone changing the frontend. The trade is made consciously and labeled, which is the pattern this book has now seen seventeen times.

But a committed build raises a subtle integrity problem, and the solution is the chapter's best code. The build was produced on the maintainer's machine. If the graph data were bundled into it, every user would forever see whatever corpus the maintainer last ingested, not their own repository. So the server intercepts exactly two paths and routes them to the live data directory, letting everything else fall through to the static bundle:

```
class ViewHandler(http.server.SimpleHTTPRequestHandler):
    def __init__(self, *args: object, **kwargs: object) -> None:
        super().__init__(*args, directory=str(dist_dir), **kwargs)

    def translate_path(self, path: str) -> str:
        """Route the two data endpoints to the live data dir, not dist/."""
        if path == "/graph.json":
            return str(graph_path)
        if path == "/cochange.json" and cochange_path.exists():
            return str(cochange_path)
        return super().translate_path(path)
```

From `cli.py`. The frontend always fetches `/graph.json`; the server decides what that means. The committed bundle stays generic, the data stays yours.

The frontend meets the data's optionality halfway. `graph.json` missing is an error (there is nothing to show), but `cochange.json` missing just means impact mode shows call-graph reach without history. A degraded mode, not a crash, mirroring how the CLI treats the same file. Small touches elsewhere reward the curious reader: the camera clamps its auto-fit zoom (one isolated node would otherwise fill the screen), and the dark/light theme follows the OS preference live.

TRY IT · ON YOUR OWN REPOSITORY THIS TIME

`uv run labgo ingest /path/to/your/repo && uv run labgo history /path/to/your/repo && uv run labgo view`

Three commands, no services, and your own codebase is on screen. Switch to Impact mode and click the file you're most afraid of. The amber dashed edges, the files that keep changing together with no structural link, are usually where the surprises live: hidden coupling the imports never admitted to.

PART VII

What This Project Teaches

The recurring illusions, named and catalogued; the disciplines that caught them; and the full command sequence for pointing all of this at your own repository. Plus what to build next, and what to keep refusing to build.

16

The Ways We Fool Ourselves

Every misleading number in this project was plausible, flattering, and caught by a repeatable check. Here is the field guide: seven illusions, four disciplines, one habit.

The catalogue of illusions

Reading the decision log end to end, the striking thing is that no number was ever wrong because of a typo or a crash. Every wrong number was produced by correct code computing the wrong thing, and every one would have survived code review, because the flaw lived in the *relationship between data sources*, not in any function. Collected in one place:

illusion	appearance here	the tell	the check that caught it
Wrong denominator	20.2% resolution counting builtins as failures (D005)	a rate whose population wasn't chosen on purpose	ask "resolved of what?"; audit the excluded
Silent zero	parser returns no commits, no error (D006)	an empty result that looks like a quiet day	non-emptiness assertions; tests that pin "must find something"
Temporal drift	45% of exam questions reference deleted files (D008)	eval set and corpus born at different times	pin them together; refuse to score on mismatch
Label noise ignored	commit file-lists treated as causal truth (D009)	absolute accuracy claims off a mined answer key	claim only relative gaps; plan a corroborated subset
Leakage	94.9% recall; each case's commit voted in its own evidence (D012)	a 4× jump from adding one signal	trace provenance of every input; leave-one-out
Unbounded budget	40.5-file predictions (D012); 17.4 (D015); 13.05 (D016)	great recall, unexamined prediction size	report size beside recall; compare at matched budget
Wrong aggregation	4.8 files/case briefly reported as 1.6 (D010)	a mean over the wrong population	state the weighting; per-case unless argued otherwise

Seven ways a plausible number lies. The unbounded budget appears three times; catching it became a reflex, and the reflex became the project's methodology.

If one row deserves to be tattooed somewhere, it is the budget row. Three separate subsystems (a co-change counter, a vector index, a language model) each independently discovered the same exploit: predict more, score higher. No component was malicious; the metric simply paid for volume, so volume is what emerged. Any metric you optimize without constraining will be gamed by the system under test, even when the system is your own arithmetic. Goodhart's law does not require anyone to be cheating. The project's antidote generalizes far beyond retrieval: **no recall without prediction size; no comparison except at matched budget.** In your domain the "budget" might be tokens, latency, alerts fired, or candidates surfaced, but the shape is identical.

The four disciplines

Against those illusions, the project wields four repeatable practices:

- **Baseline first.** The deterministic floor was scored before any LLM existed in the system, so every later layer faced the question "did you beat it?" One layer (the agent) publicly hasn't yet. Without Stage 2, Stage 4's 30.3% would have looked like success.
- **Isolate before blending.** Calls, co-change, and vectors were each measured alone before any union was allowed. When the union then nearly doubled the floor (Chapter 10), that number was interpretable. And when a blend jumped too much (Chapter 8), the isolated numbers made the leak findable.
- **Pin the exam.** Corpus SHA, cases, parameters, and evidence are committed together; the scorer refuses a drifted world. Scores from March and November are comparable, or the tool won't produce them.
- **Defer with a trigger.** Type inference (D004), a ranked union (D015), a budget-capped agent prompt (D016), the file-listing tool for large corpora (Chapter 11): each unbuilt thing is named, scoped, and attached to the measurement that would justify building it. Restraint is recorded, so it reads as judgment rather than omission.

The habit underneath all of it

Strip the graph database, the embeddings, and the agent away, and the load-bearing artifact of this repository is a Markdown file: an append-only log where every entry records what was decided, what the alternatives were, what evidence drove it, and what later changed it. Its own header states the reason with unusual candor. The question that separates senior candidates, "what went wrong and what architectural decisions did you change?", is only answerable if you wrote it down while it was still annoying.

Notice what the log did mechanically throughout this book: D005 preserved the wrong 20.2% next to the right 27.2%; D012 preserved the leaky 94.9% through two corrections; D010 recorded that its own original hypothesis named the wrong knob; D016 recorded the agent losing. A decision log is cheap, a heading and four bullet points at the moment of decision, and it compounds. It is the difference between a repository that contains its lessons and one that merely survived them. If you adopt a single practice from this book, adopt this one; it requires no infrastructure and no permission.

17

Running It Yourself, and What to Build Next

The complete command sequence from bare clone to measured agent, what each step needs and costs, and the roadmap of named next steps, each with the evidence that would trigger it.

The short version: two commands

The sequence below is the honest, tier-by-tier account, and it is worth reading in full because it is also the map of what costs money. But the repository now ships a fast path through it, for the reasonable reader who would rather see their own codebase on screen before studying the plumbing:

```
uv sync
uv run labgo analyze https://github.com/your/repo    # or a local path
```

`labgo analyze` is Tier 1 collapsed into one invocation. Hand it a git URL and it clones the target (`--filter=blob:none`, but with full history, because `labgo history` needs the commits) into `~/.labgo/corpora/<name>`, which satisfies D007's outside-the-tree rule automatically instead of asking you to remember it. Hand it a local path and it uses that path as-is, untouched. Then it runs ingest and history (skipping history with a message, not a stack trace, when the target isn't a git repository), prints a census of the languages it found, and serves the viewer. Zero credentials, zero configuration, no database.

Its companion answers the question the tiers below exist to answer, without making you read them first:

```
uv run labgo doctor            # per-stage readiness: core / neo4j / vectors / agent / extras
uv run labgo doctor --probe    # also test the Neo4j connection live
```

`doctor` prints one row per stage and, for every row that is not ready, the exact fix: the `uv sync --extra ...` to run, the `.env` line to add, the frontend build to rerun. `--probe` goes further and actually opens a connection to Neo4j rather than trusting that a URI in a file means a database at the other end. It is a report, not a gate: it always exits 0, and it says so in as many words, because nothing in it blocks `analyze`. D019 records both commands together for that reason. The point was never to hide the tiers; it was to replace tribal knowledge about which tier you are standing on with a command that tells you.

> FROM THE DECISION LOG · D019
>
> **Plug-and-play surface; the core stays zero-credential.** The old quickstart required knowing D007, running three commands in order, and knowing which stage needed which key. Two alternatives were weighed and rejected, both for the same reason. Auto-starting a local Neo4j via Docker from `labgo load`: rejected as hidden orchestration, since a command that silently starts containers is the opposite of legible. Managing corpora as a CLI cache with eviction: rejected because a directory you can `rm -rf` is simpler than cache bookkeeping. Consequence: the quickstart is one command against any repository, and the cloud stages remain exactly as opt-in as D013 and D014 designed them.

The full sequence

Everything below is cumulative; stop at any point and you have a complete, useful system. Costs are per the httpx-scale corpus; a larger repository scales roughly linearly.

Tier 1: no database, no keys, no network

```
uv sync
mkdir -p ../labgo-corpora
git clone --filter=blob:none https://github.com/encode/httpx.git ../labgo-corpora/httpx

uv run labgo ingest  ../labgo-corpora/httpx    # AST -> data/graph.json
uv run labgo history ../labgo-corpora/httpx    # git -> cochange.json, evalset.json
uv run labgo view                              # interactive map at 127.0.0.1:4173
```

(The corpus lives outside the project tree for a recorded reason, D007: `uv` auto-discovers nested `pyproject.toml` files and once silently rebuilt the virtualenv around the cloned corpus instead of LabGo. The workspace-exclude guard in `pyproject.toml` remembers the incident.) Substitute your own repository's path anywhere `httpx` appears, in any language. Python still travels the original path, `pyast.py`, untouched, which is why the 27.2% resolution rate and every httpx number in this book remain exactly reproducible. Ten more languages, JavaScript, TypeScript (with TSX), Go, Java, Rust, C, C++, Ruby, C#, and PHP, travel a generic tree-sitter engine that generalizes D004's confidence-tier ladder per language: self, exact, local, heuristic, and an honest count of the unresolved. Any other recognized extension gets a file-level fallback, a File node with its language, line count, and test flag, and no call graph. That third tier is thinner but not empty: co-change evidence is language-agnostic, and D010 and D012 established that it carries most of the retrieval signal on this corpus regardless of what any parser could see. So the map, the exam, and impact mode all work for a repository in a language nothing here can parse. What does not exist yet is a measured multi-language benchmark; every score in this book is still an httpx-and-Python score, and D018 is careful to say so.

> FROM THE DECISION LOG · D018
>
> **Multi-language ingestion: a registry, tree-sitter for ten languages, file-level fallback for the rest.** The project was Python-only in exactly two places, `pyast.py` and `gitlog.py`'s `SOURCE_SUFFIXES`, while everything downstream of `graph.json` was already language-agnostic. So the fix is an ingestion problem, not a rewrite: one registry (`ingest/languages.py`) as the single source of truth for extensions, grammars, test conventions, and vendored directories, and one dispatcher (`ingest/extract.py`) routing each file to one of the three tiers. Rejected: running Python through tree-sitter too, which would discard the measured narrative for zero measured gain. The honesty mechanism is per-language statistics, because external-call classification quality varies by language, so a weakly-classified language shows its own depressed rate instead of silently polluting the global one. That is the D005 lesson, enforced structurally. Known gaps are named rather than hidden.

Tier 2: add the graph database (free)

```
cp .env.example .env      # NEO4J_URI + NEO4J_PASSWORD from Aura (or docker compose up -d)
uv run labgo load --clear
uv run labgo benchmark ../labgo-corpora/httpx --name httpx      # pin the exam
uv run labgo baseline benchmarks/httpx --hops 2 --no-cochange   # the floor
uv run labgo baseline benchmarks/httpx --hops 2 --cochange --min-count 25
```

Tier 3: add vectors (free tier covers it; ~163K tokens for httpx)

```
uv sync --extra vectors   # + VOYAGE_API_KEY in .env
uv run labgo embed ../labgo-corpora/httpx
uv run labgo search "retry a request with exponential backoff"
uv run labgo baseline benchmarks/httpx --method hybrid --hops 2 \
      --cochange --min-count 25 --k 4                          # the 43.3% result
```

Tier 4: agent, MCP, tracing (agent-eval costs real tokens)

```
uv sync --extra agents --extra mcp --extra observability      # + ANTHROPIC_API_KEY
uv run labgo agent "httpx/_client.py" ../labgo-corpora/httpx
uv run labgo agent-eval benchmarks/httpx ../labgo-corpora/httpx --n 40 --sample-seed 42
uv run labgo mcp ../labgo-corpora/httpx           # or register it in Claude Code
```

The named roadmap

What distinguishes this backlog from a wish list is that every item carries its *trigger*, the measurement that would justify the work, because each was deferred by decision, not by fatigue:

next step	what it would do	build it when…
Budget-capped agent prompt	ask the agent for its best N files, ranked; score at matched N	immediately; D016 names it as the open experiment, and it settles whether the agent's loss was really the budget
Rename detection (`-M`)	unify pre/post-rename identities (client.py ↔ _client.py) in history mining	coupling understatement (D002) shows up as missed predictions on renamed files
Type inference (pyright/jedi)	lift the 27.2% call resolution	the eval shows call-graph recall, not co-change or vectors, is the binding constraint (D004's condition, still unmet)
Time-travel evaluation	rebuild the graph at each case's parent commit; no survivorship bias	a result materially hinges on the filtered exam's bias (D008 records it as "the rigorous version")
Ranked/weighted union	score files by how many signals agree	a measurement shows the naive union, not recall, is the bottleneck (D015's condition)
Corroborated-subset scoring	re-score on cases where co-change is backed by a call edge	an absolute (not relative) quality claim is ever needed (D009's mitigation)
`list_files` / `search_files` tools	replace the in-prompt file list	the corpus outgrows the context window (Chapter 11's noted limit)

next step	what it would do	build it when…
TypeScript path-alias resolution	give TypeScript the EXACT tier it currently lacks, by reading `tsconfig paths` and `baseUrl`	a measured non-Python corpus shows a named descope like this one, rather than the tree-sitter engine itself, is the binding constraint on EXACT-tier recall (D018)
Corpus pinning under `~/.labgo`	give corpora cloned by `labgo analyze` the pinning and verification `labgo benchmark`'s manifest already gives committed ones	corpora accumulated this way need provenance semantics beyond "a directory you can `rm -rf`" (D019)

The roadmap, with triggers. "Build it when" is the column that makes this architecture rather than procrastination.

The generic playbook

Abstract the six stages away from code and you have a template for building any measured AI system. Support triage, document routing, incident analysis, anything:

- **1.** Find the question that requires *structure*, not just similarity, and scope to it, so every architectural choice has a defense.
- **2.** Find ground truth your domain already records for free (histories, logs, tickets, resolutions) and mine an exam from it. Audit it for drift, noise, and provenance before quoting scores.
- **3.** Build the deterministic baseline and score it. This number is the project's spine.
- **4.** Add one retrieval signal at a time; measure each alone at a matched budget, then measure blends.
- **5.** Only now add an LLM, as a router over measured tools, and hold it to the baseline's standard. Publish the result either way.
- **6.** Expose the tools over MCP; trace every call with cost and latency; automate the free half of your evals in CI.

The order is the content. Every stage creates the measuring stick the next stage is held to. That is why the losing agent of Chapter 12 is not an embarrassment but the system working exactly as designed: the layers that couldn't prove their worth were caught, priced, and sent back with a named experiment. That is what *engineering* an AI system, as opposed to assembling one, actually looks like.

You now know every file in this repository, every number it publishes, and every mistake it survived. The map is drawn; the exam is pinned; the floor is set. Go point it at something you care about. When your own first flattering number arrives, you will know exactly which question to ask it.

Appendix A

Glossary

Every term of art used in this book, in plain words, with the chapter that treats it fully.

Agent

A loop in which an LLM chooses tools, sees their results, and decides what to do next; the harness executes, the model proposes. (Ch. 11)

AST (abstract syntax tree)

The tree structure a language parser builds from source code: every function, call, and import as a typed node. (Ch. 4)

Baseline

The simplest scoreable version of a system, measured first so every later addition can prove its worth against it. (Ch. 3, 8)

Benchmark (pinned)

An exam frozen together with everything that gives its scores meaning: corpus commit, cases, extraction parameters, evidence. (Ch. 6)

CALLS / IMPORTS / TESTS / CONTAINS / CO_CHANGED

LabGo's five edge kinds: function-calls-function, file-imports-file, test-exercises-function, file-contains-definition, and files-changed-together-in-git. (Ch. 4, 5)

Confidence tier

The label on every CALLS edge recording how it was resolved: exact (import), self/cls, local (same file), or heuristic (unique name). (Ch. 4)

Context window

The text an LLM can see for the current request. Its working memory, as opposed to its trained weights. (Ch. 2)

Cosine similarity

Angle-based closeness of two vectors; the mathematical meaning of "semantically similar." (Ch. 9)

Cypher

Neo4j's query language, in which graph patterns are drawn as ASCII pictures: `(a)-[:CALLS]->(b)`. (Ch. 7)

Embedding

A learned vector representation of text (here: source code) in which distance encodes similarity of meaning. (Ch. 9)

Evolutionary / logical coupling

The research-literature name for "these files keep changing together in version history." (Ch. 5)

Ground truth

Verified correct answers to score predictions against; here, mined from commits instead of hand-labeled. (Ch. 3)

Hallucination

A fluent, confident, wrong statement from an LLM, typically produced when the needed fact wasn't in its context. (Ch. 2)

Hit rate

Fraction of exam cases where a predictor found at least one correct file. (Ch. 3)

Hop

One edge-step in a graph traversal; `hops=2` means callers-of-callers. (Ch. 7, 8)

Idempotent

Safe to run twice; produces the same state. Why the loader uses MERGE, not CREATE. (Ch. 7)

Knowledge graph

A database of entities and typed relationships, queried by traversal; LabGo's map of files, functions, and their couplings. (Ch. 2, 7)

LangGraph

A library for writing agent workflows as explicit state machines: nodes, edges, conditional routing over shared state. (Ch. 11)

Leakage

Any path by which information about test answers reaches the system being tested; here, one commit feeding both exam and evidence. (Ch. 8)

Leave-one-out

Scoring each case against evidence with that case's own contribution subtracted first. (Ch. 8)

LLM (large language model)

A neural network that predicts the next token; the reasoning engine of Ch. 11's agent and nothing before it. (Ch. 2)

Matched budget

The rule that predictors may only be compared at similar prediction sizes, because recall can be bought with volume. (Ch. 8, 10, 12)

MCP (Model Context Protocol)

The open standard by which AI applications discover and call external tools and resources; JSON-RPC over stdio locally. (Ch. 13)

Precision

Of what was predicted, the fraction that was correct. The trust metric. (Ch. 3)

Prediction budget

How many files a predictor returns per case; the number that must accompany any recall claim. (Ch. 3, 8)

Property graph

Neo4j's data model: labeled nodes and typed, directed, property-carrying relationships. (Ch. 7)

RAG (retrieval-augmented generation)

Fetching relevant private data at question time and placing it in the model's context. (Ch. 2)

Recall

Of the true answer set, the fraction the prediction found. The coverage metric. (Ch. 3)

Resource (MCP)

Data an MCP server offers for a client to read once, as opposed to a tool the model invokes repeatedly. (Ch. 13)

Seed / expected

The one file shown to the system in an exam case, and the hidden set of files that actually changed with it. (Ch. 3, 5)

Semantic search

Retrieval by embedding similarity. Finds what resembles the query, not what depends on it. (Ch. 2, 9)

Span / trace

One timed, attributed operation, and the tree of spans sharing a `trace_id` that reconstructs a whole request. (Ch. 14)

Survivorship bias

Distortion from only examining what lasted; the disclosed cost of filtering the exam to still-existing files. (Ch. 6)

Temporal drift

Exam and corpus born at different times, making questions unanswerable by construction. (Ch. 6)

Tool use

The LLM API capability where the model emits a structured request to call a named function with arguments, and the harness executes it. (Ch. 11)

Transitive closure

Everything reachable by repeatedly following edges from a start point. The formal shape of "what breaks." (Ch. 2, 7)

Vector index

A data structure for fast nearest-neighbor search over stored embeddings; here, native inside Neo4j. (Ch. 9)

Appendix B

The Decision Log at a Glance

All nineteen entries of `DECISIONS.md`, one line each, with the chapter where this book treats them.

#	decision, in one line	ch.
D001	Scope to transitive impact, a question vector search provably cannot answer; the graph must be load-bearing	1, 2
D002	Ground truth mined from git commits: seed one file, predict the rest	3
D003	Exclude merges and oversized commits from history mining	5
D004	Name-based call resolution with confidence tiers; defer type inference until the eval demands it	4
D005	Fix the denominator: exclude builtin/stdlib calls; 20.2% was really 27.2%	4
D006	Two git-log parsing traps: no null bytes in argv; splitlines() eats \x1e	5
D007	Corpora live outside the tree; uv once rebuilt the venv around a cloned repo	17
D008	45% of raw eval cases were temporally unanswerable; pin benchmarks, filter, disclose the bias	6
D009	Co-change labels are noisy: relative comparisons survive, absolute claims don't	3
D010	The budget lever is min_count, not max_files; matched-budget combined: 26.9%/15.7%	8
D011	Voyage (cloud, code-specialized) over local Ollama, to measure vector search at its ceiling	9
D012	CO_CHANGED leaked into its own exam (94.9%); leave-one-out fix; the deeper flaw was the 40.5-file budget	8
D013	Neo4j AuraDB Free over local Docker; auto-pause is the real constraint; compose kept as fallback	7
D014	Embed source code, not docstrings (only 19.8% of functions have one); voyageai quarantined in an extra	9
D015	Vector-only measured (19.3% at matched budget) after catching an inflated 75.5%; naive hybrid hits 40.1%	10
D016	The agent loses (30.3%/5.8%, 13-file budget); diagnosis: uncalibrated budget, not bad reasoning	12
D017	Console-exported OTel over Langfuse; root span fixes trace identity; CI runs only the free eval half	14
D018	Multi-language ingestion behind one registry: Python keeps its untouched extractor, ten languages get tree-sitter with D004's ladder, everything else recognized gets a file-level node	17
D019	`labgo analyze` clones and runs the whole zero-credential pipeline in one command; `labgo doctor` reports per-stage readiness with the exact fix, and gates nothing	17

Appendix C

Command and File Reference

The repository's layout and every CLI command, annotated.

Source layout

```
src/labgo/
  ingest/
    models.py      graph schema: nodes, edges, confidence tiers, extraction stats
    languages.py   the language registry: extensions, grammars, test conventions (D018)
    extract.py     dispatcher: routes each file to pyast / tsast / file-level fallback
    pyast.py       Python AST -> call/import/test graph (two-pass, resolution ladder)
    tsast.py       tree-sitter engine + one TSSpec per language, ten of them (D018)
    gitlog.py      git history -> co-change pairs + eval cases (+ leave-one-out)
  benchmark.py     pinned benchmarks: manifest, cases, raw evidence, corpus verification
  graph.py         Neo4j loader: connect / read_graph_json / load_graph (MERGE, batched)
  baseline.py      deterministic impact prediction + scoring (Stage 2)
  embed.py         Voyage embeddings over Function/Class source + vector index (Stage 3a)
  hybrid.py        vector-only + hybrid prediction, same scoring (Stage 3b)
  agent.py         LangGraph tool-calling agent, five tools, budgetless (Stage 4)
  mcp_server.py    MCP server: same five tools + labgo://files resource (Stage 5)
  tracing.py       optional OpenTelemetry spans, console-exported (Stage 6)
  cli.py           all commands below
tests/              46 tests, zero live services (verified before CI was wired)
viewer/             React force-graph viewer; dist/ committed, served by `labgo view`
benchmarks/         pinned exams (committed on purpose)
data/               graph.json / cochange.json / evalset.json outputs
docs/file-map.html  the repo's own visual file-by-file explainer
DECISIONS.md        the append-only decision log (Appendix B)
WHY.md              the plain-English argument; README.md assumes you read it
```

Commands

command	needs	does
`labgo analyze <path-or-url>`	nothing	clone a URL to `~/.labgo/corpora/` (or use a local path), then ingest + history + language census + viewer, in one command
`labgo doctor [--probe]`	nothing	per-stage readiness table (core / neo4j / vectors / agent / extras) with the exact fix on every failing row; `--probe` tests Neo4j live; always exits 0
`labgo ingest <repo>`	nothing	dispatch each file to Python AST, tree-sitter, or file-level fallback → `data/graph.json` + per-language resolution stats
`labgo history <repo>`	nothing	git → `cochange.json`, `evalset.json` (raw)
`labgo view`	`graph.json`	serve the interactive viewer (no Node)
`labgo benchmark <repo> --name N`	nothing	pin an answerable, manifested exam under `benchmarks/N`

command	needs	does
`labgo verify <bench> <repo>`	nothing	refuse or confirm corpus-at-pinned-SHA
`labgo load [--clear]`	Neo4j	`graph.json` (+cochange) → database, batched MERGE
`labgo baseline <bench> [--method calls|vectors|hybrid]`	Neo4j	score a deterministic predictor; the book's tables
`labgo embed <repo>`	Neo4j + Voyage key	embed Function/Class source; build vector index
`labgo search "query"`	Neo4j + Voyage key	semantic search, no traversal, no LLM
`labgo agent <seed> <repo>`	Neo4j + Anthropic key	one agent run, report + tool/token stats
`labgo agent-eval <bench> <repo> --n 40`	Neo4j + Anthropic key	score the agent on a seeded sample (real tokens)
`labgo mcp <repo>`	Neo4j	serve the five tools over MCP stdio

Optional extras

`uv sync` alone runs Stages 1 and 2. `--extra vectors` adds `voyageai` (Stage 3); `--extra agents` adds `langgraph` + `anthropic` (Stage 4); `--extra mcp` (Stage 5); `--extra observability` (Stage 6); `--extra dev` adds `pytest` + `ruff`. Every extra is genuinely optional: absent, its stage's commands fail with a clear message and everything else runs identically.

Appendix D

All the Numbers on One Page

Every measured result quoted in this book, as recorded in the repository, on the pinned httpx benchmark (236 cases unless noted).

The corpus and the map (Stage 1)

quantity	value
corpus	httpx @ b5addb64 (~60 Python files)
graph	1,301 nodes · 2,100 edges
edge breakdown	567 CALLS · 35 IMPORTS · 257 TESTS
call resolution	27.2% of 2,845 in-scope sites
resolution tiers	4 exact · 99 self · 327 local · 343 heuristic · 2,072 unresolved
language coverage (D018)	Python AST · tree-sitter for JS, TS/TSX, Go, Java, Rust, C, C++, Ruby, C#, PHP · file-level fallback for the rest
history	1,482 commits scanned → 610 raw cases · 1,745 co-change edges
benchmark filter (D008)	565 raw → 236 answerable (58.2% dropped)
functions with docstrings (D014)	225 / 1,134 = 19.8%
embedded nodes / tokens (D014)	1,229 nodes · 162,884 tokens

Retrieval (Stages 2 and 3)

predictor	recall	precision	files/case	status
call-graph only (hops=2)	22.4%	12.2%	4.8	the honest floor
+cochange, min_count=2, leaky	94.9%	7.2%	n/a	retracted (D012 leak)
+cochange, min_count=2, leave-one-out	92.1%	5.9%	40.5	not citable (budget)
+cochange, min_count=25 (matched)	26.9%	15.7%	5.1	fair combined result
vectors only, k=4	19.3%	13.2%	~7.5	D001 checked
vectors only, first version	75.5%	n/a	17.4	retracted (D015 budget)
hybrid calls∪vectors, k=4	40.1%	14.8%	8.1	blending justified
hybrid + cochange (loo, mc=25), k=4	**43.3%**	**15.4%**	**8.3**	**best citable retrieval**

The agent (Stage 4, n=40 sample, Haiku 4.5)

quantity	value
recall / precision / hit rate	30.3% / 5.8% / 45.0%
mean prediction size	13.05 files (no cap; the diagnosis)
cost	429,839 in / 38,534 out tokens ≈ 11.7K/case
mean turns	3.05
format failures	3/40 stray `**`, caught by strict parsing
verdict (D016)	does not beat the deterministic baseline

Appendix E

Further Reading

Where to go deeper on each thread, roughly in the order the book pulled them.

- **Retrieval-augmented generation:** Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" (2020), the paper that named the pattern. For the graph-flavored variant, Microsoft Research's GraphRAG work (2024) is the reference point Chapter 2's aside contrasts with.
- **Mining software repositories:** the research field behind Chapter 5. Adam Tornhill's *Your Code as a Crime Scene* (Pragmatic Bookshelf) is the accessible treatment of change coupling and hotspot analysis; the MSR conference series is the academic vein.
- **Graph databases and Cypher:** Neo4j's free online documentation and the openCypher specification. The "variable-length pattern matching" chapter is the one this book's Chapter 7 compresses.
- **Embeddings:** Voyage AI's model documentation for the practical surface (input types, dimensions, batching caps); Jurafsky and Martin's *Speech and Language Processing* (free online) for vector semantics from first principles.
- **Evaluation discipline:** the classic treatment of leakage and test-set hygiene in any ML text; for the LLM era, Anthropic's and OpenAI's published evals guides. Chapter 16's budget obsession is Goodhart's law applied to retrieval metrics.
- **LangGraph:** the official documentation's "concepts" section (state, nodes, reducers, conditional edges) covers everything Chapter 11 used. Note how much of the prebuilt surface LabGo deliberately declined.
- **Model Context Protocol:** the specification and quickstarts at modelcontextprotocol.io. Building a toy server over data you own is the natural follow-on exercise to Chapter 13.
- **OpenTelemetry:** the official traces documentation. The span/trace/exporter model of Chapter 14 transfers unchanged to any language and backend.
- **The repository itself:** `WHY.md` for the argument, `DECISIONS.md` for the nineteen entries this book kept quoting, `docs/file-map.html` for the project's own interactive tour, and the 46 tests, which are the most honest documentation any codebase carries.

· · ·

Set in Charter and Avenir Next · diagrams in the viewer's own palette · every number quoted verbatim from the LabGo repository