Building Industrial RAG Systems from Daily Drilling Reports

A hands-on guide to retrieval-augmented engineering intelligence

Stephane Djimra

July 2026

Table of contents

Welcome

Every rig produces a paper trail. Every well produces a story — told in Daily Drilling Reports (DDRs), scattered across shared drives, scanned PDFs, and email attachments, written by dozens of different hands in a shorthand that only the field understands.

That story holds answers your team needs today: *Have we seen this stuck pipe signature before? What led to the packers failing to set before that fishing run? Have we had losses like this before?*

Somewhere in your archive, the answer already exists. Finding it is the problem.

This book builds, from scratch and in plain Python, a system that can read that archive and answer those questions — with evidence, not guesses. You will not start with theory. You will start with a single PDF and a script that reads it. By the end, you will have an Industrial DDR Intelligence Platform: a retrieval-augmented generation (RAG) system purpose-built for drilling and completions data, that you built chapter by chapter and understand end to end.

Every example in this book uses real Daily Drilling Reports — not invented ones. The archive is from **Utah FORGE**, a Department of Energy-funded geothermal research well (FORGE 16A(78)-32) whose reports are publicly available. No anonymisation, no synthetic stand-ins: you'll read genuine rig-floor language, including a real stuck-pipe event and a real fishing operation, from page one.

Before and after

Today, when someone needs to know if something has happened before, the options are limited: open PDFs one at a time and hope you pick the right one, ask whoever's been around the project longest and hope they remember, or rely on memory and risk repeating a mistake nobody wrote down clearly enough to find again.

By Chapter 5, you'll have built a working RAG system — retrieve, generate, cite — and run it against a real local model. By the end of the book, it answers cross-report questions like this one, pulling the cause from one report and the outcome from the next:

> **i** Where this is headed: a cross-report answer from the finished system

```
Question: What led to the fishing operation on report #50?

Answer: On report #49 (2020-12-07), the crew attempted multiple times
to set packers on BHA #32, but pressure readings showed the ball did
not seat and the packers failed to set. They tripped out with the
packer assembly and picked up a fishing BHA (#33) that same day. On
report #50 (2020-12-08), that fishing run milled up lost pieces of bit.
```

```
Evidence:
  FORGE-16A-78-32_Drilling_049_2020-12-07.pdf
  FORGE-16A-78-32_Drilling_050_2020-12-08.pdf
```

Both claims in that answer are directly quoted from the real report text — nothing here is invented. Getting there takes the whole book: Chapter 5's first system answers from a single best-matching report; stitching two reports together like this — the cause in #49, the outcome in #50 — needs the chunking (Chapter 7) and hybrid retrieval (Chapter 9) that Part II builds. The rest of this book is how you build that road, one working script at a time, starting with a single PDF.

What this becomes

The scripts in this book are the engine. The book's optional companion app (`book/app/`) puts the same flow on one screen — a question, a local-model answer, and the evidence it's traceable to — over the sample archive:

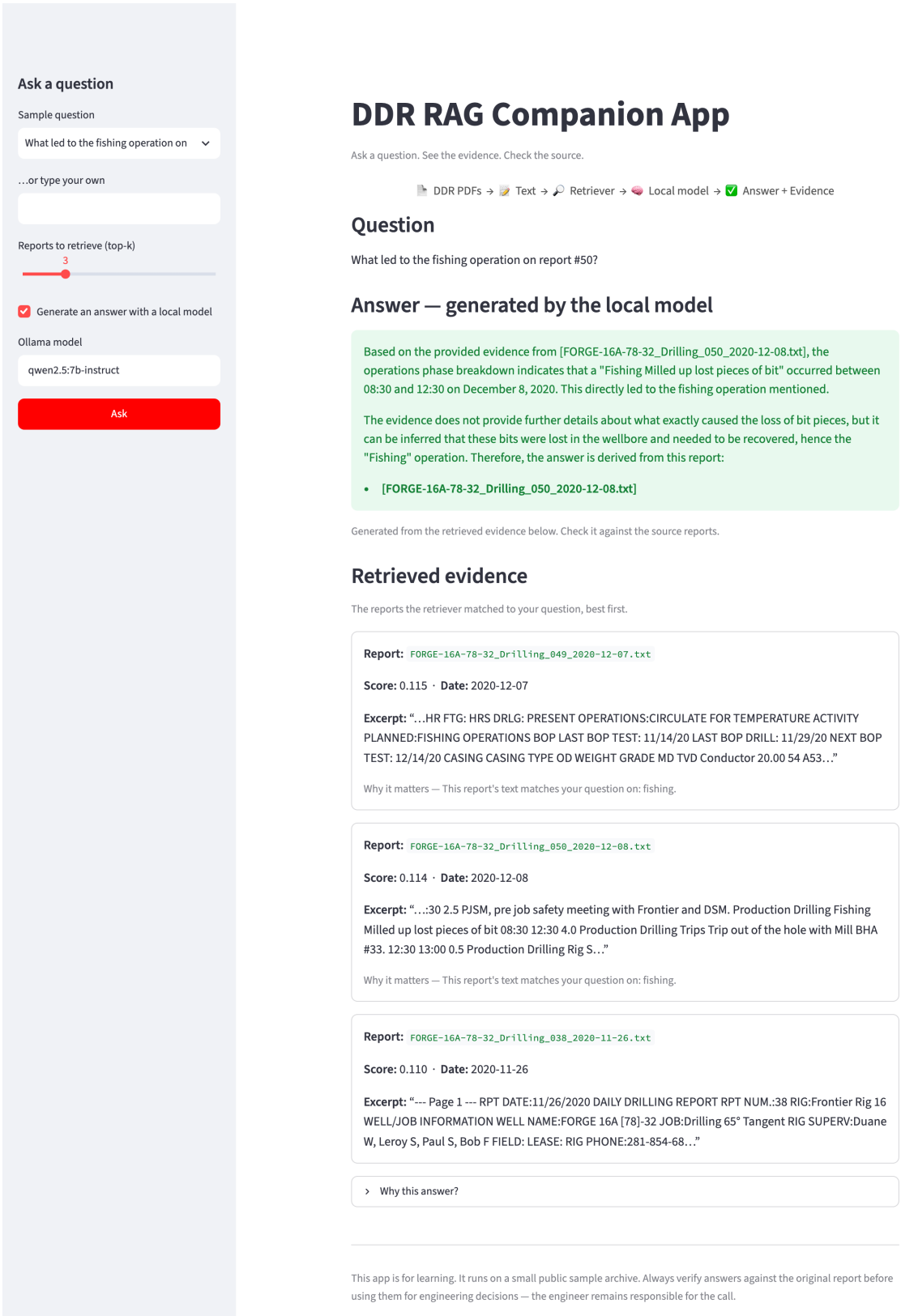


Figure 1.: The DDR RAG Companion App: a question, a local-model answer, and the source reports it's traceable to.

The companion pipeline scales the same idea to the whole 76-report archive. Appendix A shows how to set either one up and point it at your own reports.

Who this book is for

You should read this book if you are a drilling engineer, completions engineer, intervention engineer, production engineer, petroleum data scientist, or part of a digital transformation team in energy, and you:

- Have never written a line of Python in your life — or tried once and gave up.
- Have strong operational experience and are comfortable in Excel.
- Have never built a RAG system, used a vector database, or worked with embeddings.
- Are curious about AI and automation, and are willing to type code examples yourself rather than just read about them.
- Want working tools, not a machine learning course.

No prior programming or AI/ML background is assumed. Every concept — even one as basic as what a function is — is introduced only after you've hit the problem it solves, never before. The goal of this book isn't to turn you into a software engineer. It's to make you capable of building useful engineering tools.

What you will build

Chapter	You will be able to…
0	Set up a working Python environment, in whichever editor or notebook tool you prefer
1	Extract text from a DDR PDF
2	Expand oilfield abbreviations (BHA, WOB, MWD…) automatically
3	Keyword-search your entire DDR archive
4	Search DDRs by meaning, not just exact words
5	Ask a question and get a cited, evidence-backed answer
6–12	Scale the system to scanned reports, large archives, hybrid retrieval, reranking, and full traceability

Management view: why this matters

If you're deciding whether to fund or adopt work like this, here's the case in plain terms — and its limits.

- **Search time.** Finding a prior stuck-pipe or fishing precedent goes from opening reports one at a time to a single query — minutes to seconds on the archives used here.
- **Offset and lessons-learned reuse.** The same index that answers one question surfaces related events across every report, so a lesson written once can be found again instead of re-learned.
- **Onboarding.** An engineer new to a well can ask what happened and get cited reports back, rather than depending on catching whoever remembers.
- **Less reliance on tribal knowledge.** Answers come with the report and page they came from, so the knowledge lives in the archive, not only in one person's memory.
- **Auditability.** Every generated claim carries a citation an engineer can open and check (Chapter 10) — the system shows its work.
- **Build vs. buy.** The core is a few hundred lines of standard Python. The trade-off against a vendor product is control and inspectability versus vendor support and maintenance.
- **Private deployment.** It runs locally against a local model (Chapter 5), so confidential DDRs need never leave your environment.

This supports an engineer's judgment and a reviewer's audit; it doesn't replace either. It makes no claim to reduce operational risk on its own — a faster way to find the right report is only as good as the decision someone makes with it.

How this book works

Every chapter solves one operational problem and ends with something you can run. There is no chapter you read passively — you will type code, run it against real DDR text, and see it work. Theory appears only when it explains *why* the code in front of you behaves the way it does.

We follow one rule throughout: **engineering usefulness over AI novelty**. Retrieval matters more than generation. Traceability matters more than eloquent prose. Engineers trust evidence, not predictions — so this system is built to show its work, every time.

The visual language of this book

A few recurring elements appear in every chapter, explained once here so they never need repeating:

- **Progress bar** — `###.......... 3 / 12` at the top of every chapter, showing exactly how far you are through the book.
- **Estimated time** and **difficulty badge** (Beginner, Intermediate, Advanced) — next to the progress bar, so you know the commitment before you start.
- **CHECKPOINT** — a checklist near the end of each chapter, confirming the specific skills you just gained.
- **WHAT YOU BUILT** — a green box naming the concrete artifact (a script, an index, a pipeline) you now have, distinct from the conceptual "Key takeaways" earlier in the chapter.
- **Field notes** — a real operational vignette: something an experienced engineer would notice in the data that a tool alone would miss. Wherever this book's own bundled code and data can reproduce the result, it has been checked directly; a handful draw on the companion pipeline's own structured outputs (e.g. `ddr_facts.parquet`, its BM25 index) and are reported rather than independently re-derived here.

You'll also meet four recurring engineers, each asking the question that opens a chapter's **Operational Problem**:

Name	Role
Oumy	Drilling Engineer
Mike	Completions Engineer
Sarah	Intervention Engineer
Sean	Production Engineer

They're a framing device, not a source of data: whichever question Oumy, Mike, Sarah, or Sean asks, every report, number, and quote used to answer it is the real, unaltered Utah FORGE archive described above — never invented on their behalf.

Companion repository

All code, sample datasets, and notebooks referenced in this book live in the companion repository. Every chapter lists exactly which files you need under **Repository files**.

If you've never installed Python or don't know which editor to use, start with **Part 0 — Preparing Your Python Workshop**: it walks through everything from scratch, works with any of five common setups (Jupyter Notebook, VS Code, PyCharm Community, Positron, or a terminal alone), and ends with your first script actually running. If you already have Python and a working setup, jump straight to Chapter 1 — Appendix A has the rest of the reference material (the sample dataset, rendering the book, the companion pipeline) if you need it later.

Let's read our first DDR.

Preface

Why this book exists

I wrote this book because I kept seeing the same gap on drilling and completions teams: the field generates enormous amounts of written operational knowledge — Daily Drilling Reports, end-of-well reports, morning reports, NPT logs — and almost none of it is *usable* by the next engineer who needs it (Dawson and Verkuil 2014). It sits in inconsistently named PDFs — scanned or digital — scattered across SharePoint folders and email threads. When the assembly loses tool face mid-slide and gets stuck downhole, the fastest way to find out if this has happened before is often to ask the engineer who's been on the project the longest, hoping they remember a similar incident.

This book's own archive has exactly that event — a real stuck-pipe day, verified against the original report. But dropping the archive's PDFs into a folder and searching them doesn't solve it either: searching `"stuck pipe"` finds the report where the pipe actually got stuck — but misses the very next day's report entirely, because that one talks about `"tight hole,"` `"high torque,"` and a `"decision to pull out of hole"` instead. Same event, continuing, one day later — invisible to anything that only matches on exact words. That gap, between what a report *says* and what it *means*, is exactly what retrieval-augmented generation (RAG) closes.

Large language models are genuinely useful here — not because they are new or exciting, but because they solve a real, boring, expensive problem: turning an unstructured archive into something you can query. This book teaches you to build that system yourself, understand every part of it, and trust its output, rather than adopting a black-box product you cannot inspect.

What this protects

The gap this book closes isn't really about PDFs. It's about what happens when the person who remembers a similar incident retires, changes companies, or is simply on a different rig this week. A DDR archive is, in a sense, tribal knowledge that hasn't been asked to prove itself yet — findable only if you already know which report to open, or who to call.

A system like the one you'll build here doesn't replace that engineer's judgment. It replaces the search. Deciding what a stuck-pipe signature means, or whether a packer failure changes tomorrow's plan, is still yours to make — the system's job is only to put the right evidence in front of you fast enough that the decision doesn't wait on someone's memory, or on however long it takes to open dozens of PDFs one at a time. That matters most exactly where it's easiest to overlook: getting an engineer new to a well up to speed on events they weren't there for, and investigations where the fastest path to an answer today is still "ask the person who's been here longest" — which stops working the day that person isn't.

Why "industrial"

A demo that answers one question about one PDF is easy to build and easy to find online. An industrial system is different. It has to:

- Handle scanned reports as well as digital ones.
- Scale to thousands of DDRs across dozens of wells.
- Never silently invent an answer — every claim must trace back to a specific report and page.
- Be auditable by someone who was not in the room when it was built.

Every chapter in Part II exists because a toy RAG system fails at one of these requirements. We fix them one at a time, in the order you would actually hit them in the field.

How to read this book

This book borrows the practical-first, project-based spirit of *Python Crash Course* (Matthes 2023) — small wins early, theory only when the code in front of you needs it, a working artifact at the end of every chapter — but assumes considerably less going in. You do not need to have written Python before. You do not need to know what a function or a loop is before Chapter 1; each concept is explained in plain language, translated into terms an engineer already knows, exactly when you need it to solve the problem in front of you.

If you've never installed Python or set up a project folder before, start with Part 0 — it gets you to a working setup in whichever editor or notebook tool you're most comfortable with, with no assumption that you've done this before.

Read it in order the first time. Each chapter assumes the code from the previous one exists and works — this is a build, not a reference manual. After your first pass, use it as a reference: jump to the chapter that matches the problem you currently have (Chapter 6 if your PDFs are scanned, Chapter 9 if your retrieval quality is poor, Chapter 10 if your answers aren't trustworthy yet).

Type the code yourself. Do not copy-paste from the companion repository until you have typed it once and watched it run. This is slower and it is worth it — and if you finish this book able to say "I can build this," rather than "I understand the theory," it did its job.

A note on data

Most Daily Drilling Reports are operationally sensitive and confidential — which is exactly why this book doesn't use one from a commercial operator. Every example here comes from **Utah FORGE**, a Department of Energy-funded enhanced geothermal system research well (FORGE 16A(78)-32) whose reports are public: real well name, real dates, real personnel, real events, exactly as filed. No anonymisation was applied, because none was needed — this is what "publicly available" means in practice, and it lets this book show you genuine field language instead of an approximation of it.

That said, the *technique* generalises directly to a confidential archive: everything from Chapter 6 onward — OCR gating, traceability, corpus-completeness checks — exists precisely because production archives are messier and more sensitive than this one. If you point this book's code at

your own organisation's DDRs, do not commit them to a public repository, and see Appendix A for a data-handling checklist.

Acknowledgements

This book exists because daily drilling reports capture far more than operational summaries. Behind short entries such as "POOH to change BHA due to erratic torque" sit decisions, experience and operational context that are obvious to the engineers involved in the operation but often invisible to everyone else.

Thanks are due to the open-source communities behind the tools this book relies on throughout: Python, Quarto, Jupyter, and the broader Python data and NLP ecosystem. None of the engineering in this book would be reproducible without them.

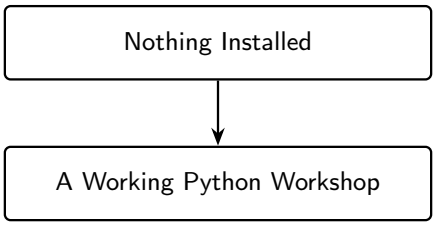
Finally, thanks to every reader working through this material on a real archive of real reports. If you find an error, an oilfield abbreviation that's wrong for your basin, or a chapter that assumes something it shouldn't, please open an issue in the companion repository — corrections from practitioners are what keep a book like this honest.

The ideas, engineering examples, educational approach and technical validation presented in this book are the author's. AI tools, including Claude, were used throughout the project to assist with code generation, editing, documentation and iterative review. All engineering decisions, implementation choices and final content remain the responsibility of the author.

Part I.

Part 0 — Preparing Your Python Workshop

Part 0: Preparing Your Python Workshop



Progress: **Part 0 — Environment Setup** (before Chapter 1) · **Estimated time:** 20–30 min · **Difficulty:** Beginner

Before you can read a single DDR with code, you need somewhere to do the work. Think of it this way:

> The IDE is just the workshop. Python is the engine. The code is the tool. The DDR archive is the job.

You wouldn't judge a rig by which brand of hard hat the crew wears, and you don't need to fight over which code editor is "correct" before you start. Any of the environments in this section will run every example in this book. Pick the one that feels least intimidating and move on — you can always switch later once you know what you're doing.

This section assumes you have never installed Python, never opened a terminal, and never used Git. If you've done some of this before, skim ahead to whichever step you actually need.

0.1 What you need

Five things, all free, all one-time setup:

- **Python** — the language every example in this book is written in.
- **A code editor or notebook environment** — where you'll read and edit the book's code. Section 0.2 helps you choose one.
- **A terminal** — a text-based window where you type commands for the computer to run. Section 0.4 explains this in plain English if the word is new to you.
- **Git** — a tool for downloading (and later updating) this book's companion code and sample DDR files in one step. If you'd rather not install it, Section 0.6 shows a no-Git alternative.
- **The companion repository** — this book's actual code and sample DDR archive, which you'll download once in Section 0.6.

None of these cost money, and none of them lock you into a particular way of working. Let's go through them one at a time.

0.2 Choosing your environment

You do not need to use any particular editor to follow this book. Every example runs the same way — as a plain Python file, or a plain code cell — no matter which of these you pick:

Environment	Best for	Notes
Jupyter Notebook	Learning and exploration	Good for step-by-step code, runs in a web browser
VS Code	General coding	Free, very common in companies, lots of online help available
PyCharm Community	Python projects	Strong project structure, a bit heavier to install
Positron	Data science workflows	Good for notebooks and scripts side by side
Terminal only	Minimal setup	Works everywhere, nothing extra to install

If you genuinely don't know which to pick: install **VS Code**. It's free, widely used outside this book too, and Appendix A2 walks through every click. If you'd rather not install anything beyond Python itself, the **Terminal only** route (Appendix A5) works identically — you'll just type commands instead of clicking buttons.

Whichever you choose, Appendices A1–A5 give you a short, dedicated guide for that specific environment once you've finished this section.

0.3 Installing Python

> Engineering Translation: Python
>
> **Python** is the engine everything in this book runs on — the same code runs whether it's read by a notebook, an editor, or the plain terminal. Installing Python once gives every tool in Section 0.2 something to run your code with.

This book was written and tested against Python 3.11 or later. Version matters because some of the libraries used later in the book (starting with Chapter 4) rely on features only present in newer Python releases — an older version can fail in confusing ways that have nothing to do with your code.

Check whether Python is already installed. Open a terminal (Section 0.4 shows you how) and type:

```
python --version
```

Expected output: something like `Python 3.11.9` or higher. If you see that, skip ahead to Section 0.4.

If it fails — an error like `command not found` or a version starting with `2.` — try:

```
python3 --version
```

Many Mac and Linux systems only recognise `python3`, not the shorter `python`. If `python3 --version` shows 3.11 or later, use `python3` in place of `python` for every command in this book.

If neither command works, Python isn't installed yet. Download it from python.org/downloads and run the installer for your operating system — on Windows, make sure to check the box labelled "Add python.exe to PATH" during installation, since that's what lets the `python` command work from a terminal at all.

0.4 Opening a terminal

> Engineering Translation: Terminal
>
> A **terminal** (sometimes called a command window, console, or shell) is a plain text window where you type instructions for the computer to run, one line at a time, instead of clicking buttons. It looks intimidating the first time and is genuinely just a way of talking to your computer directly.

How to open one:

- **Mac:** open Spotlight (Cmd+Space), type `Terminal`, press Enter.
- **Windows:** open the Start menu, type `Command Prompt` or `PowerShell`, press Enter. Either works for this book.
- **Linux:** usually Ctrl+Alt+T, or search your applications menu for "Terminal".

Once it's open, you'll see a prompt — some text followed by a blinking cursor, waiting for you to type. Try these four commands, one at a time, pressing Enter after each:

Command	What it does	Windows equivalent
`pwd`	Prints your current folder ("print working directory")	`cd` (typed alone)
`ls`	Lists the files and folders here	`dir`
`cd <folder>`	Moves into a folder, e.g. `cd Documents`	`cd <folder>` (same)
`mkdir <name>`	Creates a new folder	`mkdir <name>` (same)

Expected output: `pwd` prints a path like `/Users/yourname` (Mac/Linux) or `C:\Users\yourname` (Windows). `ls/dir` prints a list of whatever files and folders are in your current location — it may be empty if the folder is new.

If a command isn't recognised: double-check the spelling — these commands are typed exactly as shown, with no capital letters.

0.5 Creating the project folder

If you plan to use Git in the next step, you can skip this — `git clone` creates its own folder for you. This step is here for readers who'd rather set up a folder by hand first, or who will use the ZIP download option instead of Git.

In your terminal:

```
mkdir ddr-rag-book
cd ddr-rag-book
```

What this does: `mkdir ddr-rag-book` creates a new, empty folder named `ddr-rag-book` inside wherever your terminal currently is. `cd ddr-rag-book` moves your terminal *into* that folder, so every command you type next happens there.

Expected output: no message printed — silence means it worked. Run `pwd` (or `cd` alone on Windows) to confirm you're now inside `ddr-rag-book`.

0.6 Getting the companion repository

> Engineering Translation: Git and repository
>
> A **repository** ("repo" for short) is a folder of files — in this case, this book's chapters, code, and sample DDR PDFs — tracked and shared online. **Git** is the tool that copies a repository from the internet onto your own computer, and later lets you pull down any updates. Think of `git clone` as downloading a shared project folder in one command, instead of clicking "download" on dozens of individual files.

If Git is installed, run this in your terminal:

```
git clone https://github.com/djimrastephane/ddr-rag-book.git
cd ddr-rag-book
```

What this does: downloads the entire book repository — every chapter's code, the sample DDR PDFs, everything — into a new folder named `ddr-rag-book`, then moves your terminal into it.

Expected output: a handful of progress lines (`Cloning into 'ddr-rag-book'...`, `Receiving objects...`) ending without an error.

If `git` isn't recognised: you don't need to install it just for this. Go to the repository's page on GitHub, click the green **Code** button, choose **Download ZIP**, then extract the ZIP file anywhere on your computer. `cd` into the extracted folder (Section 0.4 shows how) and continue from there — everything else in this book works identically either way.

0.7 Creating a Python environment

> Engineering Translation: Virtual environment
>
> A **virtual environment** is a clean toolbox for this book. It keeps this project's packages separate from any other Python project on your computer, so installing something here can never conflict with, break, or get mixed up with anything else you have installed.

From inside the `ddr-rag-book` folder:

```
python -m venv .venv
```

What this does: creates a new, empty toolbox named `.venv` inside your project folder. Nothing is installed into it yet — that's the next section.

Expected output: no message, and a new `.venv` folder appears (check with `ls` or `dir`).

If it fails with something like `No module named venv` on Linux, your Python installation split the `venv` module into a separate package — install it with `sudo apt install python3-venv` (Ubuntu/Debian) and try again.

Now activate it — this tells your terminal "use tools from *this* toolbox until I say otherwise":

Mac/Linux:

```
source .venv/bin/activate
```

Windows:

```
.venv\Scripts\activate
```

Expected output: your terminal prompt changes to show `(.venv)` at the start of the line. That's your confirmation it worked — you'll need to see that `(.venv)` every time you come back to work on this book.

0.8 Installing packages

> Engineering Translation: pip and packages
>
> A **package** (also called a library or a dependency) is a piece of software someone else already wrote and tested, ready for you to use instead of writing it yourself — this book uses one to read PDFs, one to compare text meaning, and so on. **pip** is Python's own tool for downloading packages into your active virtual environment.

With your virtual environment active (you should still see `(.venv)` in your prompt), run:

```
pip install -r requirements.txt
```

What this does: reads `requirements.txt` — a plain list of every package this book needs, one per line — and installs all of them in one go, into your `.venv` toolbox only.

Expected output: a scrolling list of `Collecting ...` and `Successfully installed ...` lines. This can take a minute or two the first time; some packages (starting with Chapter 4's) are large.

If it fails: see Section 0.11 below — the most common causes are being in the wrong folder, or forgetting to activate `.venv` first.

0.9 Running the first test

Before touching any real DDR, confirm the whole chain — Python, the virtual environment, the installed packages — actually works together. This book's repository already includes a one-line check at `code/setup_check.py`:

```
print("DDR RAG workshop is ready.")
```

Run it:

```
python code/setup_check.py
```

Expected output:

```
DDR RAG workshop is ready.
```

If you see that exact line, everything from Sections 0.3 through 0.8 worked. If you see an error instead, it names exactly what's still missing — jump to Section 0.11.

> **Notebook users**
>
> If you're working in Jupyter Notebook or another notebook environment instead of running files from the terminal, you don't need `setup_check.py` as a file at all — just open a new notebook cell, type `print("DDR RAG workshop is ready.")`, and run that cell directly. Every code example in this book can be run either way.

0.10 Running the first book script

Now run the actual script Chapter 1 builds — this book's very first real, working tool:

```
python code/chapter_01/read_ddr.py
 ↪  datasets/sample_ddrs/FORGE-16A-78-32_Drilling_038_2020-11-26.pdf
```

What this does: opens one real Daily Drilling Report PDF (already included in the repository you downloaded in Section 0.6) and prints its full text straight to your terminal.

Expected output: several dozen lines of real report text, starting with a header block (`RPT DATE :11/26/2020`, `WELL NAME:FORGE 16A [78]-32`…) and ending with a time breakdown that includes the line `During the slide lost tool face and became assembly became stuck`.

If it fails:

- `ModuleNotFoundError: No module named 'pdfplumber'` — your virtual environment isn't active, or Section 0.8 didn't complete. Re-run Sections 0.7 and 0.8.
- `No such file or directory` — you're not inside the `ddr-rag-book` folder, or the path is mistyped. Run `pwd` (or `cd` alone on Windows) to check where you are.

If this printed real report text, Part 0 is complete — Chapter 1 explains exactly how this script works, line by line.

0.11 Troubleshooting

Symptom	Likely cause	What to do
`python: command not found`	Python isn't installed, or isn't on your system's PATH	Try `python3` instead. If that also fails, reinstall Python from python.org, checking "Add to PATH" on Windows.
`pip: command not found`	Same as above, or your virtual environment isn't active	Confirm `(.venv)` shows in your prompt (Section 0.7); if not installed at all, try `python -m pip` instead of `pip` directly.
Commands run, but nothing looks like it worked	You're in the wrong folder	Run `pwd` (or `cd` alone on Windows) and compare it to where you expect to be; `cd` back into `ddr-rag-book`.
`No such file or directory` / `FileNotFoundError`	Wrong folder, or a typo in the file path	Run `ls` (or `dir`) to see what's actually there; file and folder names are case-sensitive on Mac/Linux.
A package import fails (`Module NotFoundError`)	Virtual environment isn't active, or Section 0.8 wasn't run	Re-activate `.venv` (Section 0.7), then re-run `pip install -r requirements.txt`.

Symptom	Likely cause	What to do
`Permission denied`	Your user account doesn't have write access to that folder	Don't use `sudo`/administrator install for this project — instead, work inside a folder you own, like your home directory or Documents.
Windows path errors (e.g. backslashes look wrong)	Windows uses `\` in paths, this book's examples are written with `/`	Both usually work in a modern Windows terminal, but if a command fails specifically because of slashes, replace `/` with `\` in that one command.

CHECKPOINT — Part 0

- ☒ `python --version` (or `python3 --version`) shows 3.11 or later
- ☒ Your terminal prompt shows `(.venv)`
- ☒ `pip install -r requirements.txt` finished without errors
- ☒ `python code/setup_check.py` printed `DDR RAG workshop is ready.`
- ☒ `python code/chapter_01/read_ddr.py datasets/sample_ddrs/FORGE-16A-78-32_Drilling_038_2020-11-26.pdf` printed real report text

WHAT YOU BUILT

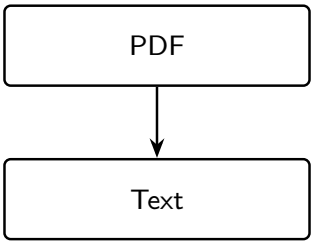
A working Python environment — Python, an active virtual environment, every package this book needs, and a folder with the real Utah FORGE DDR archive already in it, ready to run the companion repository's tests and every chapter that follows.

If all five checkpoint items are true, you have a working Python workshop, and Chapter 1 starts exactly where this section leaves off. If you'd like a step-by-step guide specific to the environment you chose in Section 0.2, see Appendices A1 (Jupyter Notebook), A2 (VS Code), A3 (PyCharm Community), A4 (Positron), or A5 (Terminal only).

Part II.

Part I — Foundations

1. Reading Your First DDR



Progress `#.....` **1 / 12** · **Estimated time:** 45–60 min · **Difficulty:** Beginner

1.1. Learning objectives

By the end of this chapter, you will be able to:

- Explain why DDR PDFs are hard for software to read, and the difference between a *digital-native* PDF and a *scanned* one.
- Extract raw text from a digital-native PDF using Python.
- Write a script that processes a single DDR, or an entire folder of them, from the command line.
- Save extracted text so it's ready for the cleaning work in Chapter 2.

! Before you start

This chapter assumes your Python workshop is already set up. If you haven't done that yet, complete Part 0 first and confirm:

- Python runs (`python --version` or `python3 --version` shows 3.11 or later).
- Your virtual environment is active (your terminal prompt shows `(.venv)`).
- Packages are installed (`pip install -r requirements.txt` finished without errors).
- The sample DDR files exist (`datasets/sample_ddrs/` has ten PDFs).
- `python code/setup_check.py` prints `DDR RAG workshop is ready.`

If all five are true, you're ready for this chapter. If any of them aren't, Part 0, Section 0.11 has a troubleshooting table for exactly this situation.

1.2. Operational Problem

It's Monday morning. Oumy, the drilling engineer, asks on the call: *"Have we had a stuck pipe event on this well before? What happened, and how was it resolved?"* The answer is somewhere in the DDR archive — but that archive is 76 PDF files, named inconsistently by whatever software produced them, and the one report that matters is buried in there like a single page in a filing cabinet with no index. Nobody wants to open 76 PDFs one at a time to answer a question that took ten seconds to ask.

This book uses a real, publicly available archive to solve exactly that problem: the Daily Drilling Reports from **Utah FORGE** — a Department of Energy-funded research project at the University of Utah, drilling and testing an enhanced geothermal system (EGS) well, **FORGE 16A(78)-32**, in Beaver County, Utah. Because it's a public research well rather than a commercial operator's confidential asset, its DDRs are openly available — which means every example in this book is real, traceable text, not an invented stand-in.

Before we can search this archive, summarize it, or ask it questions, we need the words inside those PDFs available to a computer as plain text. That's the whole job of this chapter: given one PDF, get its text into something Python can work with. Nothing clever yet — no search, no abbreviation handling, no AI. Just a reliable way to turn a PDF into text. Everything in this book is built on top of getting this first step right.

1.3. Example DDR extract

Real DDR excerpt — report #38, FORGE 16A(78)-32, 2020-11-26

```
RPT DATE:11/26/2020
DAILY DRILLING REPORT RPT NUM.:38
WELL NAME:FORGE 16A [78]-32 JOB:Drilling 65° Tangent
PRESENT OPERATIONS:PIPE FREE, TRIP OUT OF HOLE FOR BHA INSPECTION

TIME BREAKDOWN
23:30 04:00 4.5 Drill From 6,360' to 6,507', (147') Total,
32.6' feet per hour.
WOB 20 TO 35k, Rotary 50, Torque 6,500, SPP 3200-3400 GPM 560,
DIFF 200-300psi
During the slide lost tool face and became assembly became stuck
04:00 06:00 2.0 Work pipe, circulate lube sweep, work tool back in position
Pipe free
```

This is one of ten real reports curated for Part I, spanning the well's actual timeline from rig-up in October 2020 through this stuck-pipe day in November and a fishing operation in December. Every excerpt in Part I is genuine, unedited report text — no anonymisation was needed, because this data is already public. Notice that report #38 answers the morning call question directly — *if* you happen to open this one file out of 76. That's the gap this chapter closes.

1.4. Theory

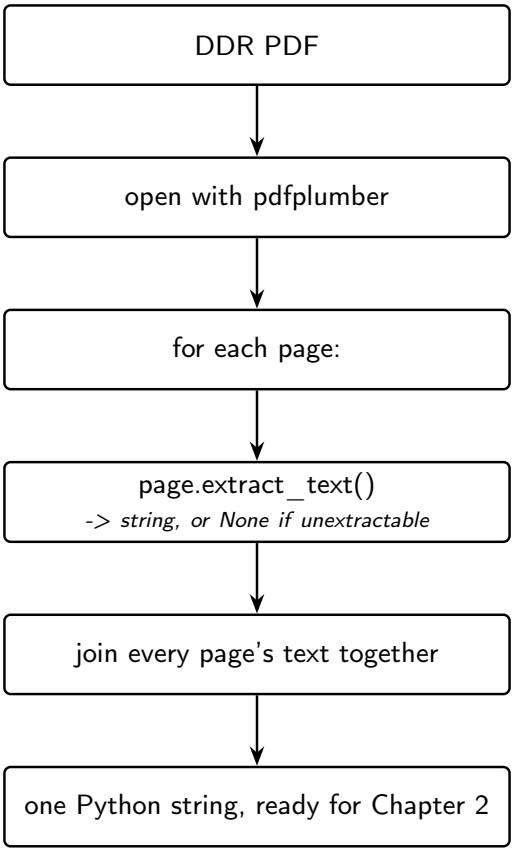
A PDF is not a text file wearing a costume. It's a page-description format: instructions for *drawing* characters at specific coordinates, not a stream of text with a beginning and an end. Two DDRs that look identical on screen can be built completely differently under the hood:

- **Digital-native PDFs** were generated directly from software (in this case, WellEz, the drilling data system that produced every report in this archive). The text was placed on the page as actual character data, so a library can recover it.
- **Scanned PDFs** are photographs of paper reports. There is no character data at all — just a picture of a page. Extracting text from these requires optical character recognition (OCR), which we cover in Chapter 6.

The Utah FORGE archive is entirely digital-native — every report was produced by the same software, so extraction is reliable throughout Part I. This won't always be true of a real archive (Chapter 6 deals with the scanned-report case), but it's the right place to start.

We'll use `pdfplumber`, a Python library that reads a PDF's internal structure and hands back the text (and, later in the book, tables and layout) on each page. There are other libraries that do this — `PyPDF2`, `pymupdf` — and they're worth knowing about, but `pdfplumber` is a good default: actively maintained, and its text extraction is reliable on the kind of structured reports we're working with.

> Engineering Translation: Library
>
> A **library** is borrowed tooling someone else already built and tested — the code equivalent of using a manufacturer's torque chart instead of deriving it yourself. `pdfplumber` is a library that already knows how to open a PDF and read the characters on its pages, so you don't have to write that from scratch.



1.5. Implementation

1.5.1. Step 1: get the sample archive

What problem are we solving?

Before extracting anything, confirm the report archive is actually there and see what you're working with — the same way you'd check a folder of scanned tickets exists before writing a script against it.

Inputs

- The folder `datasets/sample_ddrs/`, containing ten curated Utah FORGE DDR PDFs (already included in this repository — no download needed).

Expected Output

A list of ten file paths, sorted, something like:

```
FORGE-16A-78-32_Drilling_003_2020-10-22.pdf
FORGE-16A-78-32_Drilling_038_2020-11-26.pdf
...
FORGE-16A-78-32_Drilling_050_2020-12-08.pdf
```

```
import pathlib
sorted(pathlib.Path("datasets/sample_ddrs").glob("*.pdf"))
```

> **Notebook, script, and IDE users**
>
> This is the first code you'll run in this book — however you set up Part 0, here's how each option runs it:
>
> - **Notebook users:** paste this into a new cell and run it directly (Shift+Enter in Jupyter or Positron).
> - **Script users:** save it as a `.py` file and run it from the terminal with `python your_file.py`.
> - **IDE users:** use your editor's Run button, or the terminal command shown above — both produce identical output.
>
> Every code block in this book works the same way; this note won't repeat after this point.

What just happened?

You pointed Python at a folder and asked it to list every file ending in `.pdf`. Nothing was opened or read yet — this is just confirming the folder and its contents exist before you build anything on top of them.

> **Engineering Translation: Path and glob**
>
> A `Path` is Python's way of pointing at a file or folder location — think of it as a shortcut or bookmark to somewhere on disk. `glob("*.pdf")` is a filter: "show me everything in this folder whose name ends in `.pdf`," the same way you might filter a file browser to show only PDFs.

If you want to build this curated subset yourself from a full local copy of the public archive (or extend it with more reports), see `code/chapter_01/build_sample_archive.py` — it's documented in Appendix A.

1.5.2. Step 2: extract text from one PDF

What problem are we solving?

Prove that we can pull real text out of a single PDF before building anything more elaborate — the smallest possible version of the whole chapter's job.

Inputs

- One PDF: `datasets/sample_ddrs/FORGE-16A-78-32_Drilling_038_2020-11-26.pdf` (report #38, the stuck-pipe day).

Expected Output

The full text of the report's first page, printed to your terminal — header block, casing and mud tables, BHA component list, survey data, and the time breakdown quoted above, all as plain text.

```
import pdfplumber

with pdfplumber.open(
↪  "datasets/sample_ddrs/FORGE-16A-78-32_Drilling_038_2020-11-26.pdf") as pdf:
    first_page = pdf.pages[0]
    print(first_page.extract_text())
```

What just happened?

You opened the PDF, grabbed its first page, and asked `pdfplumber` to hand back everything readable on it as plain text. Real DDRs pack in far more than a simple narrative — this one report alone holds seven distinct data tables (casing, mud, drill bits, pumps, BHA, survey data, consumables), plus a header and a narrative time breakdown — and all of it came back as one ordinary block of text you can now search, print, or save.

> Engineering Translation: Opening a file safely
>
> `with pdfplumber.open(...) as pdf:` is like signing a piece of equipment out and back in automatically. Everything inside the indented block happens while the file is "checked out"; once the block ends, Python closes it for you — so you never forget to release the file, even if something goes wrong partway through.

Every DDR in this archive is a single page, but that won't be true of every report you'll ever encounter. Loop over every page and keep track of which page each chunk of text came from — you'll need that later for citations:

What problem are we solving?

Handle reports with more than one page, and remember which page each piece of text came from — information you'll need in later chapters to point back to the exact source of an answer.

Inputs

- Any DDR PDF path, single-page or multi-page.

Expected Output

One combined string containing every page's text, with a marker like `--- Page 1 ---` showing where each page starts.

```
from pathlib import Path
import pdfplumber

def extract_text(pdf_path: Path) -> str:
    pages_text = []
    with pdfplumber.open(pdf_path) as pdf:
        for page_number, page in enumerate(pdf.pages, start=1):
            text = page.extract_text() or ""
            pages_text.append(f"--- Page {page_number} ---\n{text}")
    return "\n\n".join(pages_text)
```

What just happened?

This packages the previous step into a reusable procedure: for every page in the PDF, pull its text, label it with a page number, and stitch all the labelled pages together into one string. Instead of writing this logic out every time, you now have one named tool — `extract_text` — you can hand any PDF path to.

> Engineering Translation: Function and loop
>
> A **function** is a standard operating procedure: a named set of steps (`extract_text`) you can hand a new input to and trust it will follow the same procedure every time, instead of re-writing the steps by hand for each report. A **loop** (`for page_number, page in enumerate(...)`) means "repeat this operation for every page in the report" — the same way an SOP says "repeat for each joint" instead of listing every joint by hand.

Note the `or ""`. If a page has no extractable text — a scanned page mixed into an otherwise digital report, for example — `extract_text()` returns `None`, and every downstream step in this book expects a string, not `None`. This is a safety check, the code equivalent of confirming a valve position before proceeding instead of assuming it: assume real data is messier than the happy path, and this pattern shows up throughout the book.

1.5.3. Step 3: make it a script you can run from the command line

What problem are we solving?

Turn this into a tool you can point at any file or folder from the terminal, without opening or editing the code each time.

Inputs

- A single PDF path, or a folder of PDFs plus a `--batch` flag and an `--out` destination folder.

Expected Output

Printed text to the terminal (single-file mode), or a folder of `.txt` files, one per PDF (batch mode).

```
# One file, printed to the terminal
python code/chapter_01/read_ddr.py
 ↪  datasets/sample_ddrs/FORGE-16A-78-32_Drilling_038_2020-11-26.pdf

# A whole folder, saved as .txt files
python code/chapter_01/read_ddr.py datasets/sample_ddrs/ --batch --out
 ↪  datasets/ddr_text
```

What just happened?

The full script, `code/chapter_01/read_ddr.py`, wraps the `extract_text` function in a command-line interface: flags like `--batch` and `--out` work like switches on a control panel, letting you choose "one file, printed" or "whole folder, saved" without touching the code itself.

Try the first command now. You should see the full report text, including the line: `During the slide lost tool face and became assembly became stuck` — real language from a real rig report, not a paraphrase.

1.6. Production Reality

This chapter assumes every DDR is a clean, digital-native PDF produced by the same software — true for this entire archive, and the easiest possible starting point. A real multi-well, multi-operator archive usually isn't that tidy. Expect:

- scanned reports with no character data at all (Chapter 6)
- rotated or skewed pages
- duplicate reports filed under two different names
- handwritten notes or annotations layered on top of a digital report
- reports that are simply missing for a given day

None of that shows up in Part I's ten curated reports, on purpose — you need a reliable extraction script before you can handle the messy cases. Chapter 6 comes back to this list once you have somewhere solid to stand.

1.7. Practical exercise

Beginner

Try it yourself: Help Oumy check the rest of the archive the way she asked: run `read_ddr.py` in batch mode against `datasets/sample_ddrs/`, saving the output to `datasets/ddr_text/`. Then open `FORGE-16A-78-32_Drilling_050_2020-12-08.txt` and find the line describing what the crew milled up that day.

You'll know it worked when: you have ten `.txt` files in `datasets/ddr_text/`, one per PDF, and you can find the word "Fishing" in report 050's text without opening the original PDF.

1.8. Field notes

⚠ Field notes: a table that reads like nonsense until you know why

Action: extract report #38's `BHA` section and read it as plain text.
Result:

```
# COMPONENT OD ID LENGTH # COMPONENT OD ID LENGTH
1 Drill Bit-Reed SKC613M-01C 8.75 1 7 Monel Collar-NMDC
with MWD 6.75 3.25 30.35
2 Motor-6.5'' 7/8, 5.7, 0.242 RPG 1.50,* Fixed 6.5 33.01
8 Monel Collar-NMDC (2 joints) 6.75 3.25 60.66
```

Why: the real BHA table on this report is laid out as **two side-by-side sub-tables** —

components 1–6 on the left half of the page, components 7–10 on the right half. Visually, that's obvious: two neat columns. But `extract_text()` reads left to right, top to bottom, so component 1's row and component 7's row — which just happen to sit on the same horizontal line of the page — get concatenated into one line of output text, with no marker showing where the left sub-table ends and the right one begins.

Lesson: `extract_text()` gives you the words on the page, not the *layout* those words depend on to make sense. A two-column table isn't a special case — it's normal DDR formatting — and reading its flattened output at face value would have you believe component 1 and component 7 are the same row. This book's chapters work around it by favouring the narrative `TIME BREAKDOWN` text over the structured tables for search and retrieval — real table-aware parsing (detecting table boundaries and reconstructing rows correctly) is its own discipline, and the companion pipeline's `src/rag_pdf/table_detect.py` and `table_extract.py` handle it properly. For now, just know that "extracted text" and "correct reading order" are not the same guarantee.

1.9. Challenge exercise

Intermediate

Challenge: Extend the script to also print, for each PDF, the total number of pages and the total character count extracted. Then compare report #3 (October 22, before the well was spudded — `PRESENT OPERATIONS: RIGGING UP...`) against report #38 (the stuck-pipe day): how much richer is the later report's text, and why might that be? (Hint: check the `SPUD DATE` and `DFS/DOL` fields in each report's header.) A reference solution is in `code/chapter_01/challenge/`.

1.10. Key takeaways

- A PDF's internal structure, not its on-screen appearance, determines whether text can be extracted directly. Digital-native PDFs can; scanned PDFs need OCR (Chapter 6).
- `pdfplumber` turns a PDF into plain Python strings, one page at a time.
- Handle `None` from `extract_text()` explicitly — real-world DDR archives will contain pages that don't extract cleanly, and your code should never crash on them.
- A five-line function (`extract_text`) plus a thin command-line wrapper is already a genuinely useful tool. You don't need machine learning to save an engineer an hour of searching PDFs by hand — and it works identically whether the PDF is a synthetic example or, as here, a real report from a real well.

1.11. Repository files

File	Purpose
`code/chapter_01/read_ddr.py`	Extracts text from one PDF or a folder of PDFs

File	Purpose
`code/chapter_01/build_sample_archive.py`	Reproduces the curated Part I subset from the full public archive
`datasets/sample_ddrs/`	Ten real, curated Utah FORGE DDR PDFs used throughout Part I
`notebooks/chapter_01_explore.ipynb`	Interactive notebook version of this chapter

CHECKPOINT — Chapter 1

- ☒ Told a digital-native DDR PDF apart from a scanned one
- ☒ Extracted the full text of a real report, including its BHA table
- ☒ Wrote a reusable `extract_text` function that handles multi-page reports
- ☒ Turned it into a command-line script that runs on one file or a whole folder

WHAT YOU BUILT

`read_ddr.py` — a PDF text extraction pipeline: point it at one DDR or an entire folder, and it hands back clean, page-numbered text, ready for Chapter 2's abbreviation work.

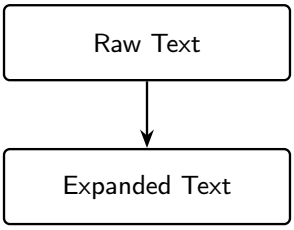
1.12. What can you do now that you couldn't do before?

You can turn any digital-native DDR PDF — or a whole folder of them — into plain text you can read, search, or hand to another script, without opening a single PDF viewer.

1.13. Suggested next step

Coming up in Chapter 2: The text you just extracted uses real oilfield shorthand and drilling terminology — `BHA`, `WOB`, `SPP`, `PJSM` — some spelled out, some abbreviated, exactly as the field actually writes it. Chapter 2 builds an abbreviation expansion engine grounded in what's actually in this archive, turning this raw text into something both humans and machines can search reliably.

2. Cleaning Operational Text



Progress ##.......... **2 / 12** · **Estimated time:** 30–45 min · **Difficulty:** Beginner

2.1. Learning objectives

By the end of this chapter, you will be able to:

- Explain why oilfield shorthand breaks naive text search and NLP.
- Build a dictionary-based abbreviation expander that respects word boundaries so it doesn't corrupt unrelated text.
- Apply the expander across every `.txt` file produced in Chapter 1.

2.2. Operational Problem

Mike, the completions engineer, wants next chapter's search tool to answer: *"Show me every report that mentions the bottom hole assembly."* Will it find report #38? Open the `.txt` file you extracted from it in Chapter 1 and check: the report never once writes "bottom hole assembly." It writes `BHA` — nine times. A search tool that only knows the literal words you typed will come back empty, on a report that was exactly the one you needed.

2.3. Why keyword search fails

Question: *"Show me every report that mentions the bottom hole assembly."*

Keyword search for "bottom hole assembly" finds nothing in report #38, because the report never uses that phrase. Instead it says `BHA` — the same equipment, different word. A computer matching text literally has no way to know these mean the same thing. Report #38 also talks about `WOB`, `MWD`, `SPP`, and a dozen other abbreviations a keyword search would need to be told about, one by one, before it could ever find them.

Before we can search, summarise, or hand this text to a language model, we need to expand it into language that means the same thing to a human and to a machine. That's the whole job of this chapter.

2.4. Example DDR extract

Real excerpt — report #38, showing both patterns

```
PJSM, pre job safety meeting.                          <- self-expanded by source
...
Pick up Curve assembly 2, BHA 21, Reed Hycalog      <- BHA never expanded
SKC613M-01C Trip in hole to 5,200'
...
WOB 20 TO 35k, Rotary 50, Torque 6,500, SPP         <- WOB, SPP never expanded
3200-3400 GPM 560, DIFF 200-300psi
```

2.5. Theory

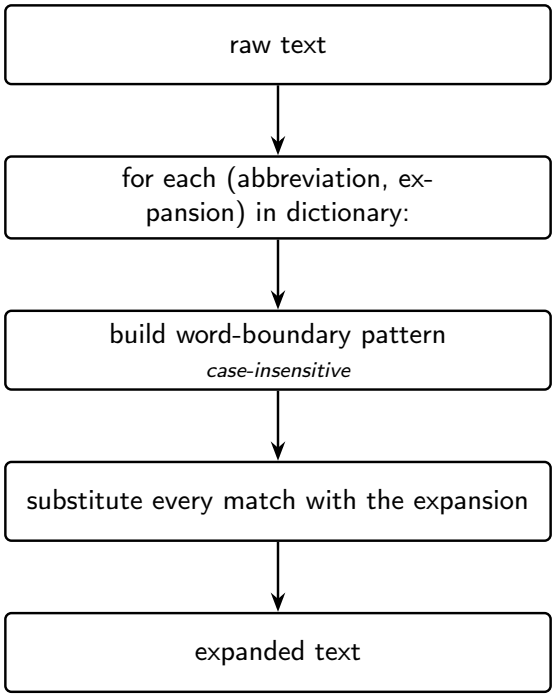
An abbreviation expander is just a dictionary lookup — `{"BHA": "bottom hole assembly", "WOB": "weight on bit", ...}` — but two details make or break it:

1. **Word boundaries.** `str.replace("MD", ...)` will happily mangle a word that merely *contains* "MD" as a substring. Match on whole words using regex `\b...\b`, not raw substring replacement.
2. **Case.** DDRs mix cases inconsistently — this archive's headers are often all-caps while narrative text is mixed-case — so expansion needs to be case-insensitive without forcing the rest of the sentence to change case.

Engineering Translation: Dictionary

A Python **dictionary** is an equipment lookup table: you give it a tag (`"BHA"`) and it gives you back what that tag means (`"bottom hole assembly"`) — the same way a lookup table on the rig floor maps a short code stamped on a piece of equipment to its full specification.

This is deliberately not a machine-learning problem. A few dozen well-chosen abbreviations, expanded correctly, solve most of the readability problem — and unlike a model, a dictionary is auditable: you can list every expansion it will ever make.



2.6. Implementation

2.6.1. Step 1: build the lookup table

What problem are we solving?

Give the computer the same knowledge a drilling engineer already carries in their head: what each abbreviation stands for.

Inputs

- A hand-built list of oilfield abbreviations and their full-text expansions, drawn from what actually appears in this archive.

Expected Output

A Python dictionary — no output printed yet, just a lookup table in memory, ready for the next step to use.

```
# code/chapter_02/expand_abbreviations.py
ABBREVIATIONS = {
    "BHA": "bottom hole assembly",
    "WOB": "weight on bit",
    "RPM": "revolutions per minute",
    "ROP": "rate of penetration",
    "MD": "measured depth",
    "TVD": "true vertical depth",
```

```
    "SPP": "stand pipe pressure",
    "GPM": "gallons per minute",
    "DLS": "dogleg severity",
    "MWD": "measurement while drilling",
    "NMDC": "non-magnetic drill collar",
    "UBHO": "universal bottom hole orientation sub",
    "NPT": "non-productive time",
    "ECD": "equivalent circulating density",
    "MW": "mud weight",
    "DFS": "days from spud",
    "DOL": "days on location",
    "PDC": "polycrystalline diamond compact",
}
```

What just happened?

You wrote down, once, what every abbreviation in this archive means. This list is the entire "knowledge" the expander needs — no training, no model, just a table you can read top to bottom and check by eye.

2.6.2. Step 2: expand every abbreviation safely

What problem are we solving?

Turn `BHA` into `bottom hole assembly` everywhere it appears as its own word — without also mangling words that merely happen to contain those same letters.

Inputs

- Raw report text (a Python string).
- The `ABBREVIATIONS` lookup table from Step 1.

Expected Output

The same text, with every whole-word abbreviation replaced by its expansion — for example, `Pick up Curve assembly 2, BHA 21` becomes `Pick up Curve assembly 2, bottom hole assembly 21`.

```
import re

def expand_text(text: str, abbreviations: dict[str, str] = ABBREVIATIONS) -> str:
    for abbr, expansion in abbreviations.items():
        pattern = r"\b" + re.escape(abbr) + r"\b"
        text = re.sub(pattern, expansion, text, flags=re.IGNORECASE)
    return text
```

What just happened?

For every entry in the lookup table, this checks the text for that exact word — not just those letters anywhere, but that whole word on its own — and swaps in the full expansion, regardless of whether it was typed in capitals or lowercase.

> Engineering Translation: Word boundary matching
>
> `\b...\b` is a stricter kind of find-and-replace: instead of "contains these letters anywhere," it means "matches this exact word, with a space or punctuation on either side." That's the difference between correctly expanding the word `MD` and accidentally mangling it inside an unrelated word that merely contains those two letters.

2.6.3. Step 3: run it across the whole archive

What problem are we solving?

Apply the expander to every report at once, not just one file at a time.

Inputs

- The folder of `.txt` files Chapter 1 produced: `datasets/ddr_text/`.

Expected Output

A new folder, `datasets/ddr_text_expanded/`, with one expanded `.txt` file per input file, same filenames.

```
python code/chapter_02/expand_abbreviations.py \
    --in-dir datasets/ddr_text --out-dir datasets/ddr_text_expanded
```

What just happened?

The script read every `.txt` file Chapter 1 produced, ran each one through `expand_text`, and saved the result under the same filename in a new folder — so you can always find the expanded version of a report by matching its name against the original.

2.7. Production Reality

This chapter's dictionary covers the abbreviations that actually appear in this archive — because it's all one well, produced by one operator's reporting software. A larger, multi-well or multi-operator archive is rarely that uniform. Expect:

- different operators abbreviating the same equipment differently
- typos and inconsistent spacing (`BHA`, `B.H.A.`, `B H A`)
- unit mismatches hiding behind the same abbreviation (`MW` as mud weight on one report, something else entirely on another)
- abbreviations this dictionary has simply never seen before

None of that appears in Utah FORGE's reports, which is exactly why this is the right place to learn the technique before facing a messier archive.

2.8. Practical exercise

Beginner

Try it yourself: Help Mike get his answer — run the expander against `FORGE-16A-78-32_Drilling_038_2020-11-26.txt`.

You'll know it worked when: `Pick up Curve assembly 2, BHA 21` becomes `Pick up Curve assembly 2, bottom hole assembly 21`, and no unrelated word (check the `MD`/`TVD` survey table carefully — short abbreviations are the ones most likely to collide with real words) gets corrupted.

2.9. Field notes

⚠ Field notes: the source already expands some abbreviations — but not the ones that matter

Action: search every report for the literal string `PJSM`.
Result: it appears in six of the ten sample reports (not logged on rig-up or trip days when no fresh crew safety meeting was held) — but every single time it does appear, it's immediately followed by its own expansion, written by the source software itself: `PJSM, pre job safety meeting`.
Why: this archive's report-generation software (WellEz) evidently has a template that spells "pre job safety meeting" out in full every time, right after the abbreviation. But run the same check against `BHA`, `WOB`, `MWD`, or `SPP` — the terms that actually carry operational meaning in the time breakdown — and none of them ever get that treatment, anywhere in the archive.
Lesson: don't assume a real archive is consistently either abbreviated or expanded. This one is both, unevenly, by accident of whatever template a report-writing tool happened to use for one specific phrase. An automated expander has to cover every term regardless — you can't rely on the source to have already done the job for the ones that matter.

2.10. Challenge exercise

Intermediate

Challenge: Extend `ABBREVIATIONS` using Appendix B, and handle the mud-table abbreviations (`FV`, `WL`, `PV`, `YP`, `CL`) — notice how short these are, and think carefully about whether blindly expanding a two-letter token like `PV` is actually safe against this archive's real text, or whether it needs a narrower match context. A reference solution is in `code/chapter_02/challenge/`.

2.11. Key takeaways

- Word-boundary regex, not substring replacement, is the difference between a correct expander and a silently broken one.
- Real archives are inconsistent by nature — the same report can spell one abbreviation out in full while never expanding another. Don't assume the source data will document itself.
- A transparent dictionary beats an opaque model here: every expansion is inspectable and testable.
- Cleaning text isn't optional busywork — the expanded text is what Chapter 3's keyword search matches against. (Chapter 4's semantic search works from the raw text instead; its Field Notes measure exactly how much expansion does and doesn't help there.)

2.12. Repository files

File	Purpose
`code/chapter_02/expand_abbreviations.py`	Word-boundary abbreviation expander
`datasets/ddr_text_expanded/`	Expanded output from Chapter 1's extracted text
`notebooks/chapter_02_explore.ipynb`	Interactive companion notebook

CHECKPOINT — Chapter 2

- ☒ Explained why oilfield shorthand breaks naive text search
- ☒ Built a word-boundary-safe abbreviation dictionary
- ☒ Expanded every `.txt` file from Chapter 1 without corrupting unrelated text

WHAT YOU BUILT

`expand_abbreviations.py` — an abbreviation expansion engine: point it at Chapter 1's extracted text and it hands back plain-language text, ready for Chapter 3's search index.

2.13. What can you do now that you couldn't do before?

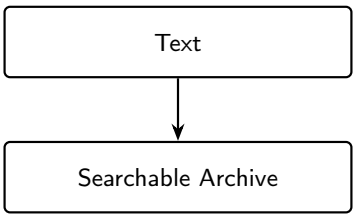
You can take the raw text Chapter 1 extracted and turn every shorthand term in it into plain language — so a report that only ever says `BHA` can now be found by anyone (or anything) searching for "bottom hole assembly."

2.14. Suggested next step

Coming up in Chapter 3: With clean, expanded text in hand, Chapter 3 builds the first tool an engineer would actually reach for: a keyword search engine that answers "which reports mention

losses?" in milliseconds.

3. Searching DDRs



Progress ###..... **3 / 12** · **Estimated time:** 45–60 min · **Difficulty:** Beginner

3.1. Learning objectives

By the end of this chapter, you will be able to:

- Build an inverted index over a folder of cleaned DDR text.
- Answer keyword queries like "which reports mention losses?" in milliseconds instead of minutes.
- Explain why exact keyword matching, while fast and simple, will eventually miss things a drilling engineer would consider an obvious hit.

3.2. Operational Problem

You now have clean, expanded text for every DDR. Sarah, the intervention engineer, asks the question out loud on the next call: *"Which reports mention losses?"* or *"Show me the report where they had to fish for something."* Opening files one by one doesn't scale past a handful of reports — you need an index.

3.3. Example DDR extract

> **i** Real query and expected hit

```
Query: "losses"
Expected hit: FORGE-16A-78-32_Drilling_019_2020-11-07.txt
  → "Mud fluids seepage losses in six hours 9 bbls"
```

That's report #19 — a minor, real seepage-loss event, easy to miss by eye across 76 reports, trivial to find once indexed.

3.4. Theory

An **inverted index** flips the natural document → words relationship around: instead of asking "what words are in this document," it stores, for every word, *which documents contain it*. Looking up "losses" becomes a single dictionary lookup instead of scanning every file. This is the same core idea behind every search engine, from `grep` to Google — the difference is scale and ranking sophistication, not the fundamental mechanism.

> Engineering Translation: Index
>
> An **index** here is exactly what it sounds like from the front of a manual: a table of contents for reports. Instead of "chapter 4, see page 112," it says "the word 'losses', see report_019." Looking a word up in that table is instant; reading every report to find the same word is not.

Tokenization matters here: "losses" and "LOSSES" should be the same token, and punctuation shouldn't create spurious tokens (`6,507'` should not prevent `6507` or nearby words from matching cleanly).

```
documents                          inverted index
------------------------------     --------------------------
report_038: "...became stuck..."   "stuck"   -> {report_038}
report_019: "...seepage losses..." "pipe"    -> {report_038}
report_050: "...milled up lost..." "losses"  -> {report_019}
                                    "milled"  -> {report_050}

query "stuck" -> look up "stuck" -> {report_038}   (one dict lookup,
                                                      not ten file scans)
```

3.5. Implementation

3.5.1. Step 1: split text into searchable words

What problem are we solving?

Before we can look anything up, every report's text needs to be broken into a consistent set of words — so "Losses", "losses," and "LOSSES" all count as the same match.

Inputs

- Any string of report text.

Expected Output

A list of lowercase words with punctuation stripped out, for example:

```
["mud", "fluids", "seepage", "losses", "in", "six", "hours", "9", "bbls"]
```

```python
# code/chapter_03/keyword_search.py
import re

def tokenize(text: str) -> list[str]:
    return re.findall(r"[a-z0-9]+", text.lower())
```

What just happened?

This lowercases the whole report, then pulls out every run of letters or digits as one word, throwing away punctuation, quotes, and apostrophes. `6,507'` becomes just `6507` — no comma, no foot mark — so it can be matched consistently whichever way it was typed in the original report.

3.5.2. Step 2: build the lookup table

What problem are we solving?

Build a lookup table that remembers which reports contain each word, so a search never has to open and re-scan every file.

Inputs

- A folder of cleaned, expanded report text: `datasets/ddr_text_expanded/`.

Expected Output

A dictionary mapping each word to the set of report filenames that contain it, for example:

```
{"losses": {"FORGE-...-32_Drilling_019_2020-11-07.txt"},
 "stuck":  {"FORGE-...-32_Drilling_038_2020-11-26.txt"},
 ...}
```

```python
from collections import defaultdict
from pathlib import Path

def build_index(text_dir: Path) -> dict[str, set[str]]:
    index: dict[str, set[str]] = defaultdict(set)
    for path in sorted(text_dir.glob("*.txt")):
        tokens = set(tokenize(path.read_text(encoding="utf-8")))
        for token in tokens:
            index[token].add(path.name)
    return index
```

What just happened?

We built a lookup table that remembers which reports contain each word. For every report, we found its unique words and, for each one, noted "this word shows up in this report." The result is one table you can check instead of seventy-six files you'd otherwise have to open.

3.5.3. Step 3: answer a query

What problem are we solving?

Given a query like "stuck pipe," find every report that contains all of those words — instantly, using the lookup table instead of opening a single file.

Inputs

- The lookup table from Step 2.
- A query string, e.g. `"stuck pipe"`.

Expected Output

The set of filenames containing every word in the query, for example `{"FORGE-...-32_Drilling_038_2020-11-26.txt"}` for the query `"stuck"`.

```
def search(index: dict[str, set[str]], query: str) -> set[str]:
    query_tokens = tokenize(query)
    if not query_tokens:
        return set()
    results = index.get(query_tokens[0], set()).copy()
    for token in query_tokens[1:]:
        results &= index.get(token, set())
    return results
```

What just happened?

For each word in the query, we looked up its list of reports in the table, then kept only the reports that appeared on *every* one of those lists. That's why this is called an AND query: a report only matches if it contains every word you searched for, not just one of them. The challenge exercise below asks you to add an OR option too.

3.6. Production Reality

This index lives entirely in memory and rebuilds itself every time you run the script — fine for 76 reports, and still fine for a few thousand. A real multi-well archive stretches this in ways worth knowing about early:

- thousands of reports mean rebuilding the index on every run gets slow — real systems save the index to disk or a database instead
- OCR'd scanned reports (Chapter 6) introduce garbled tokens that were never real words, bloating the index with noise
- singular/plural and tense variants (`loss` vs `losses`, `drilled` vs `drilling`) are treated as completely different words by this simple tokenizer, so a search can still miss obvious matches
- numbers and units (`6,507'` vs `6507 ft`) can drift apart across reports from different software or operators

None of this changes the core idea — a lookup table beats scanning every file — but it's why production search systems add stemming, on-disk indexes, and cleanup steps this chapter deliberately leaves out.

3.7. Practical exercise

Beginner

Try it yourself: Build the index over `datasets/ddr_text_expanded/` and run `search(index, "fishing")`.

You'll know it worked when: the result set contains **two** reports, not one: `FORGE-16A-78-32_Drilling_049_2020-12-07.txt` (the day the crew picked up a fishing BHA and logged `ACTIVITY PLANNED: FISHING OPERATIONS`) and `FORGE-16A-78-32_Drilling_050_2020-12-08.txt` (the day they actually milled up lost pieces of bit). If you expected only one result, that's a useful surprise: the word "fishing" genuinely appears in both the planning day and the execution day, and an AND/OR keyword index has no concept of "the day it actually happened" versus "the day it was planned" — it just finds every document containing the word.

3.8. Field notes

⚠ Field notes: the search that misses the day after

Query: `"stuck pipe"`
Result: one hit — report #38 (2020-11-26), the day the assembly actually got stuck. Report #39, the very next day, is invisible to this query.
Why: report #39's own words are:

- "Work tight hole at 6,526'."
- "Due to high torque decision to pull out of hole"
- "Hole drag from 6,050' to 5,901' no issues"

but the word "stuck" never appears anywhere in report #39. `pipe` does (buried in a BHA table row, `Drill Pipe-5'' HWDP`), but the AND search still fails because `stuck` alone has zero matches outside report #38.
Lesson: an engineer researching *"how did the stuck-pipe situation on this well develop"* would want report #39 too — tight hole, high torque, and a decision to pull out of hole are exactly the kind of continued-risk language that matters the day after an incident. Keyword search can't surface it, because report #39 never uses the word the query asked for. This isn't a bug in the code above — it's the fundamental limit of matching on exact words at all. Chapter 4 comes back to this same query.

3.9. Challenge exercise

Intermediate

Challenge: Add phrase search (query tokens must appear *adjacent* and in order, not just co-occur anywhere in the document) and an OR operator. Test `search(index, "stuck")` against report #38, then try `"packers OR fishing"` and confirm it returns both report #49 (packers) and #50 (fishing). A reference solution is in `code/chapter_03/challenge/`.

3.10. Key takeaways

- An inverted index turns "search every file" into "look up a word," which is the difference between milliseconds and minutes at scale.
- Tokenization choices (case folding, punctuation handling) determine what counts as a match — get this wrong and searches silently fail.
- Exact keyword matching is fast, simple, and fundamentally limited: it cannot find report #49 ("packers did not set… pick up Fishing BHA") when you search "lost circulation," even though report #19's "seepage losses" line and report #49's packer failure are both, in a loose sense, "things going wrong downhole" — a human would connect them faster than exact keywords can.

3.11. Repository files

File	Purpose
`code/chapter_03/keyword_search.py`	Inverted index and AND-query search
`notebooks/chapter_03_explore.ipynb`	Interactive companion notebook

CHECKPOINT — Chapter 3

- ☒ Built an inverted index over a folder of cleaned DDR text
- ☒ Answered keyword queries in milliseconds instead of minutes
- ☒ Identified exactly where and why exact-word matching misses related reports

WHAT YOU BUILT

`keyword_search.py` — an inverted-index keyword search engine over the whole cleaned archive, fast enough to answer a query while someone's still on the call.

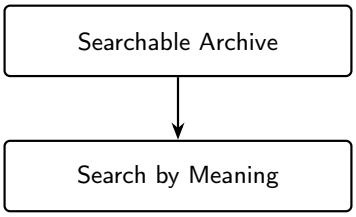
3.12. What can you do now that you couldn't do before?

You can search an archive of reports for exact words or phrases and get an answer in milliseconds, instead of opening files one at a time.

3.13. Suggested next step

Coming up in Chapter 4: Keyword search is exact and unforgiving — it doesn't know that "high torque decision to pull out of hole" and "stuck pipe" describe related trouble. Chapter 4 replaces exact matching with semantic search, so retrieval works by meaning, not just spelling — and comes back to this exact "stuck pipe" query to see how much that actually fixes.

4. Semantic Search



Progress **####.....** **4 / 12** · **Estimated time:** 60–75 min · **Difficulty:** Intermediate

4.1. Learning objectives

By the end of this chapter, you will be able to:

- Explain, without heavy math, what a text embedding is and why nearby vectors mean similar meaning.
- Generate embeddings for a folder of DDR text using `sentence-transformers`.
- Retrieve the most semantically similar chunks to a query using cosine similarity, without any exact keyword overlap required.
- Recognise that semantic search over whole documents is an improvement, not a complete fix — and that granularity is what actually determines how well it works.

4.2. Operational Problem

Chapter 3 ended on a real miss: Sarah's query `"stuck pipe"` found report #38 (the actual incident) but was blind to report #39, which describes tight hole, high torque, and a decision to pull out of hole — real continued-risk language, written the very next day — without ever using the word "stuck." Sarah, reading both reports herself, says "yes, obviously, day two of the same problem." A keyword search can't see that at all. Let's find out, honestly, how much of that gap semantic search actually closes.

4.3. Example DDR extract

The same query, now by meaning

```
Query: "stuck pipe"  (same query Chapter 3 used)

Keyword search (Chapter 3): report #39 not in results at all.

Semantic search, whole-report embeddings (this chapter): report #39
  ranks 5th out of 10 - findable, but not prominent. Report #38 (the
  actual incident) ranks 2nd.
```

4.4. Theory

An **embedding model** converts a piece of text into a fixed-length vector of numbers — typically a few hundred dimensions — such that texts with similar meaning end up as vectors that are close together in that space. You don't need to understand the neural network that produces this vector; you need to understand what it gives you: a numeric representation of meaning you can compare with simple arithmetic.

Engineering Translation: Vector / embedding

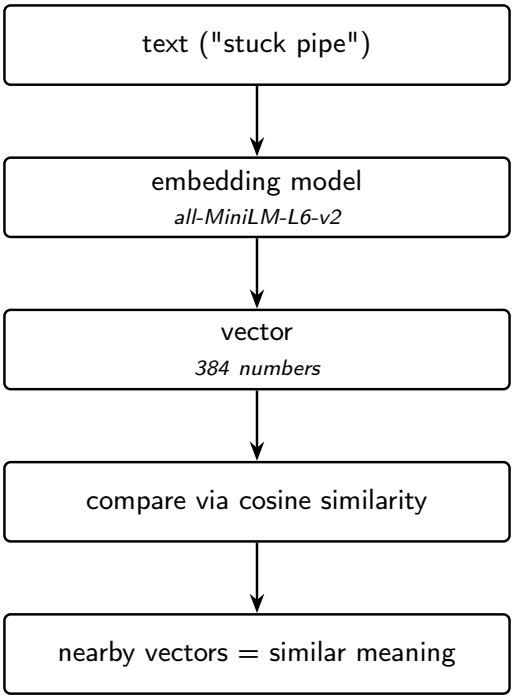
Think of an embedding as GPS coordinates for meaning. Instead of latitude and longitude, it's a few hundred numbers — but the idea is the same: two sentences with similar meaning end up with "coordinates" close together, the same way two rigs in the same field end up with similar latitude and longitude, even if their names have nothing in common.

Cosine similarity measures the angle between two vectors — 1.0 for identical direction, 0 for unrelated, negative for opposite. It's the standard way to compare embeddings because it ignores vector *length* and focuses purely on *direction*, which is what carries the meaning here.

Engineering Translation: Cosine similarity

Cosine similarity is a compass bearing, not a ruler. It asks "which way is this thing pointing?", not "how big is it?" — so a short sentence and a long paragraph about the same topic can still score as pointing the same direction, the same way two surveys can report the same azimuth regardless of how long each run was.

We use `sentence-transformers` (Reimers and Gurevych 2019) with a small, fast model (`all-MiniLM-L6-v2`) — good enough to prove the concept and small enough to run on a laptop CPU. Chapter 8 replaces the brute-force search here with a proper vector database once the corpus grows past what fits comfortably in memory.



4.5. Implementation

4.5.1. Step 1: load every report's text

What problem are we solving?

Get every report's text into memory, alongside its filename, so we know which score belongs to which report later.

Inputs

- A folder of cleaned report text, e.g. `datasets/ddr_text/`.

Expected Output

Two matching lists: report filenames, and each report's full text.

```
# code/chapter_04/semantic_search.py
from pathlib import Path

def load_chunks(text_dir: Path) -> tuple[list[str], list[str]]:
    filenames, texts = [], []
    for path in sorted(text_dir.glob("*.txt")):
        filenames.append(path.name)
        texts.append(path.read_text(encoding="utf-8"))
    return filenames, texts
```

What just happened?

Nothing about meaning yet — this just reads every text file into memory and keeps a matching list of filenames, so that whatever score we calculate next can be traced back to the report it came from.

Note the folder: `datasets/ddr_text/`, Chapter 1's raw extraction — *not* Chapter 2's `datasets/ddr_text_expanded/`. That's deliberate, and the Field Notes at the end of this chapter measure exactly why.

> Engineering Translation: Type hints
>
> The `-> tuple[list[str], list[str]]` after the function name looks cryptic but is only a label: it says this function hands back a pair of lists of text strings — the filenames, and the report texts. Python doesn't enforce it; it's a note for the next reader and your editor, the same way a valve tag states what flows through without changing the valve. You'll see these `: Path` and `-> str` annotations throughout the book — you can read straight past them without losing the thread.

4.5.2. Step 2: convert text into vectors

What problem are we solving?

Turn each report's text into a numeric representation of its meaning, so that "similar meaning" can be measured with arithmetic instead of exact word matching.

Inputs

- `MODEL_NAME = "all-MiniLM-L6-v2"`, a pre-trained embedding model.
- The list of report texts from Step 1.

Expected Output

One 384-number vector per report, all packed into a single array.

```
import numpy as np
from sentence_transformers import SentenceTransformer

MODEL_NAME = "all-MiniLM-L6-v2"

def embed_texts(model: SentenceTransformer, texts: list[str]) -> np.ndarray:
    embeddings = model.encode(texts, normalize_embeddings=True)
    return np.asarray(embeddings)
```

What just happened?

The model read each report and produced its "GPS coordinates for meaning" — one vector per report. `normalize_embeddings=True` scales every vector to the same length, so later we can compare direction only, which is exactly what cosine similarity needs.

4.5.3. Step 3: rank reports against a query

What problem are we solving?

Given a query like "stuck pipe," find which reports are closest in meaning — not which reports contain those exact words.

Inputs

- The query string.
- The array of report vectors from Step 2.

Expected Output

A ranked list of `(filename, score)` pairs, highest similarity first.

```
def search(model: SentenceTransformer, query: str, filenames: list[str],
           embeddings: np.ndarray, top_k: int = 3) -> list[tuple[str, float]]:
    query_vec = model.encode([query], normalize_embeddings=True)[0]
    scores = embeddings @ query_vec  # cosine similarity, since vectors are
↪   normalized
    top_indices = np.argsort(-scores)[:top_k]
    return [(filenames[i], float(scores[i])) for i in top_indices]
```

What just happened?

The query got turned into its own vector, then compared against every report's vector at once. Because every vector was normalized to the same length in Step 2, one matrix multiplication (`embeddings @ query_vec`) gives back the cosine similarity for every report simultaneously — that's the "arithmetic" that replaces exact word matching.

> Engineering Translation: Sorting and list comprehensions
>
> Two lines of shorthand do the ranking. `np.argsort(-scores)[:top_k]` returns the *positions* that would put the scores in order — the minus sign flips it to highest-first — and `[:top_k]` keeps just the first few. Then `[(filenames[i], float(scores[i])) for i in top_indices]` is a *list comprehension*: read it left to right as "for each position `i` in the top list, pair its filename with its score." It's a compact way to build a list without writing out a full loop, and it's a pattern you'll meet often from here on.

Run this against all ten sample reports with `query="stuck pipe"` and `top_k=10` (the whole ranking, not just the top few), and you get the real numbers behind the callout above:

```
1. 0.2416  Completion_003_2021-01-06
2. 0.1978  Drilling_038_2020-11-26   <- the actual stuck-pipe day
3. 0.1713  Drilling_049_2020-12-07
4. 0.1657  Drilling_003_2020-10-22
5. 0.1498  Drilling_039_2020-11-27   <- tight hole / high torque, findable now
6. 0.1406  Drilling_048_2020-12-06
```

```
7. 0.1342  Drilling_019_2020-11-07
8. 0.1326  Drilling_037_2020-11-25
9. 0.1319  Drilling_050_2020-12-08
10. 0.1169 Drilling_036_2020-11-24
```

Report #39 went from *absent* (Chapter 3) to *rank 5 of 10* — a real improvement, worth having. But it's not a clean win: the scores are bunched close together (0.12–0.24), and a busy engineer scanning only the top 2 or 3 results would still miss it.

4.6. Production Reality

This chapter runs the embedding model locally, on a laptop CPU, against ten short reports. Real deployments hit constraints this setup never sees:

- some teams call a hosted embedding API instead of running a model locally — which means sending report text, possibly confidential well data, to a third party. That's a data-governance conversation, not just a technical choice, and it's worth having before it's a surprise.
- embeddings from two different model versions aren't comparable — if you re-embed your archive with a newer model, old and new vectors can't be compared against each other, and everything needs re-embedding together.
- embedding a full multi-well archive (thousands of reports, not ten) takes real time and, for hosted APIs, real money — a cost that scales with archive size, not query volume.
- a similarity score like 0.24 is only meaningful *relative to other scores in the same search* — it is not "24% similar" in any absolute sense, and comparing scores across different queries or models is not valid.

4.7. Field notes

⚠ Field notes: why whole-document embeddings are noisy

Query: `"stuck pipe"`, same as above.
Result: report #39 ranks 5th of 10 when you embed each *entire report* as one vector — an improvement over keyword search's zero, but not a clean fix.
Why: each report is one page, but that page holds seven distinct data tables (casing, mud, drill bits, pumps, BHA, survey data, consumables) plus the narrative time breakdown. Embedding the whole thing blends a handful of sentences of relevant narrative with a few hundred words of numeric table content. The relevant signal — "Work tight hole… high torque… pull out of hole" — is a small fraction of what actually goes into the vector.
Isolate just the narrative lines instead of the whole report, and the picture changes — though not as cleanly as you might hope:

```
0.6244  report #38 - "Pipe free"
0.3359  report #38 - "During the slide lost tool face and became
                      assembly became stuck"
0.2402  report #39 - "Due to high torque decision to pull out of hole"
```

```
0.2331  report #39 - "Hole drag from 6,050' to 5,901' no issues"
0.2206  report #49 - "Attempted multiple times to set packers"
0.2043  report #36 - "Trip out of hole with BHA #17 core assembly."
        (genuinely unrelated coring operation)
0.2008  report #39 - "Work tight hole at 6,526'."
```

The two lines that directly describe the stuck-pipe event itself separate clearly from everything else (0.62 and 0.34 vs. everything else under 0.25) — something whole-document embedding couldn't show at all. But below that, there's no clean cutoff: report #39's own `"hole dr ag...no issues"` line — a routine trip note, not part of the incident — scores almost as high as its `"high torque"` line (0.23 vs. 0.24), while report #36's genuinely unrelated coring line (0.20) lands in the same narrow band as report #39's own `"tight hole"` line (0.20), the actual precursor to the stuck-pipe event. Shared drilling vocabulary ("hole," "trip," "torque") pulls topically-similar-but-irrelevant lines close to genuinely relevant ones once you're past the two standout lines.

Lesson: semantic search doesn't fail because the *technique* is wrong — it fails here because the *unit of retrieval* is too coarse. Line-level granularity narrows the field dramatically, but it doesn't hand you a clean similarity threshold to filter on — the middle of the ranked list still mixes relevant and irrelevant lines together. This is exactly the problem Chapter 7's chunking (and later, reranking) works on, and it's worth remembering the next time a semantic search "isn't working": check what's actually inside each embedded vector before blaming the model — but don't expect a single score cutoff to solve it either.

⚠ Field notes: does expanding abbreviations actually help semantic search?

Action: embed the ten reports twice — once from Chapter 1's raw text (`datasets/ddr_tex t/`), once from Chapter 2's expanded text (`datasets/ddr_text_expanded/`) — and compare the same two queries at `top_k=3`.

Result:

```
Query "bottom hole assembly":
  raw text:      report #38 ranks 2nd  (0.302)
  expanded text: report #38 ranks 1st  (0.372)   <- BHA now reads in full

Query "stuck pipe":
  raw text:      report #38 ranks 2nd
  expanded text: report #38 ranks 2nd            <- essentially unchanged
```

Why: expansion only helps a query whose wording matches a term you actually expanded. `"bottom hole assembly"` improves because the reports now literally contain that phrase instead of only `BHA` — so report #38 climbs from 2nd to 1st. `"stuck pipe"` doesn't move, because "stuck" was never an abbreviation to begin with.

Lesson: expansion earns its keep for keyword search (Chapter 3, which matches literal words), but its benefit to *semantic* search is real yet narrow — confined to queries that happen to use a term you expanded. That narrowness is why this book feeds the raw text to the embedding model from here on, rather than adding a whole pipeline stage the semantic path barely uses. Keyword search keeps using the expanded text; embeddings use the raw. Neither choice is

> arbitrary, and now you've measured why.

4.8. Practical exercise

Beginner

Try it yourself: Embed all ten sample DDRs and run `search(model, "stuck pipe", filenames, embeddings, top_k=10)`.

You'll know it worked when: `FORGE-16A-78-32_Drilling_039_2020-11-27.txt` appears somewhere in your ranked results — unlike Chapter 3, where it didn't appear at all — even if it isn't near the top.

4.9. Challenge exercise

Intermediate

Challenge: Reproduce the Field Notes line-level result yourself. Manually pull three lines of text, verbatim from the source reports — report #39's `"Due to high torque decision to pull out of hole"`, report #39's `"Hole drag from 6,050' to 5,901' no issues"`, and report #36's `"Trip out of hole with BHA #17 core assembly."` (a genuinely unrelated line) — embed just those three strings, and confirm the high-torque line scores highest against `"stuck pipe"`, report #39's own hole-drag line scores close behind it, and the genuinely unrelated line trails both by a real margin. A reference solution is in `code/chapter_04/challenge/`.

4.10. Key takeaways

- Embeddings represent meaning as geometry: similar meaning, nearby vectors.
- Cosine similarity is the standard comparison because it isolates direction (meaning) from magnitude.
- Semantic search over whole documents is a real improvement on keyword search — report #39 goes from unfindable to rank 5 of 10 — but "an improvement" and "solved" are different claims. Don't oversell it.
- Granularity is often the actual lever, not the embedding model. The same query, same model, same text — just isolated to individual narrative lines instead of a whole noisy report — turns a middling rank-5 result into a clear, checkable win.
- Brute-force cosine search (a matrix multiply) is entirely adequate at ten documents. It stops being adequate well before you reach the full 76-report archive's scale — that's Chapter 8's problem to solve.

4.11. Repository files

File	Purpose
`code/chapter_04/semantic_search.py`	Embedding generation and brute-force cosine search
`notebooks/chapter_04_explore.ipynb`	Interactive companion notebook

CHECKPOINT — Chapter 4

- ☒ Explained what a text embedding is without needing the underlying math
- ☒ Generated embeddings for a folder of DDR text
- ☒ Ranked reports by semantic similarity to a query using cosine similarity
- ☒ Diagnosed granularity, not the model, as the real bottleneck

WHAT YOU BUILT

`semantic_search.py` — a brute-force semantic search engine: embed a folder of reports once, then rank any query against them by meaning, not spelling.

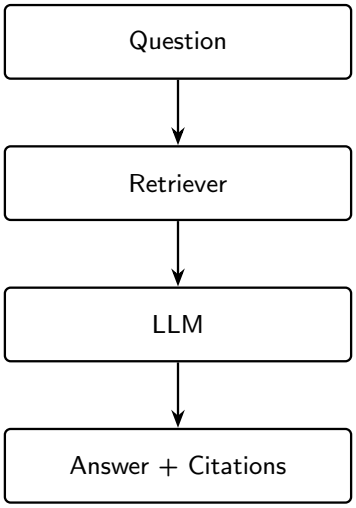
4.12. What can you do now that you couldn't do before?

You can retrieve relevant report passages by meaning instead of exact wording — finding report #39's continued-risk language even though it never says "stuck" — and you know precisely why the result isn't perfect yet: granularity, not the model, is the real bottleneck.

4.13. Suggested next step

Coming up in Chapter 5: You can now retrieve relevant passages by meaning, and you've learned the honest limits of doing it at whole-document granularity. Chapter 5 turns retrieval into an answer: given a question, retrieve the right evidence, and generate a response that cites exactly where each claim came from. Chapter 7 comes back to fix the granularity problem directly.

5. First RAG System



Progress ##### . . . . . **5 / 12** · **Estimated time:** 60–90 min · **Difficulty:** Intermediate

5.1. Learning objectives

By the end of this chapter, you will be able to:

- Explain retrieval-augmented generation (RAG) as two separate, inspectable steps: retrieve, then generate.
- Assemble retrieved chunks into a prompt that forces a language model to answer only from the evidence it's given.
- Generate that answer with a real local model (via Ollama) and return it with an explicit evidence list — the source reports and excerpts — rather than an unsupported paragraph.

5.2. Operational Problem

Retrieval (Chapter 4) gets you the right passages. What Mike, the completions engineer, actually wants is an answer: *"What led to the fishing operation on report #50?"* — not two ranked text chunks he has to read and synthesise himself. But a generated answer that isn't traceable back to specific reports is worse than no answer at all in an engineering context, because it invites misplaced trust. This chapter builds the smallest system that does both: synthesise an answer, and show exactly where every part of it came from.

5.3. Example DDR extract

Target interaction, grounded in two real, adjacent reports

```
Question: What led to the fishing operation on report #50?

Answer: On report #49 (2020-12-07), the crew attempted multiple times
to set packers on BHA #32, but pressure readings showed the ball did
not seat and the packers failed to set. They tripped out with the
packer assembly and picked up a fishing BHA (#33) that same day. On
report #50 (2020-12-08), that fishing run milled up lost pieces of bit.

Evidence:
  FORGE-16A-78-32_Drilling_049_2020-12-07.pdf
  FORGE-16A-78-32_Drilling_050_2020-12-08.pdf
```

Both claims here are directly quoted from real report text — nothing in this answer was inferred beyond what the two reports state. This is the *target*. Keep reading — the practical exercise below checks whether the simple system this chapter builds actually reaches it.

5.4. Theory

RAG (retrieval-augmented generation) (Lewis et al. 2020) is exactly the two chapters you just built, chained together: retrieve relevant evidence (Chapter 4), then hand that evidence to a language model and ask it to answer *using only that evidence*. The model is not answering from what it memorised during training — it's answering from the specific passages you retrieved, which is what makes citation possible at all.

Engineering Translation: LLM

An **LLM** (large language model) is like a very well-read new hire: it has absorbed an enormous amount of general drilling and engineering writing, so it writes fluently and knows the vocabulary — but it has never seen *your* wells or *your* reports. Left to its own memory, it will guess. RAG is how you make sure it answers from the actual report in front of it, not from a guess dressed up in confident language.

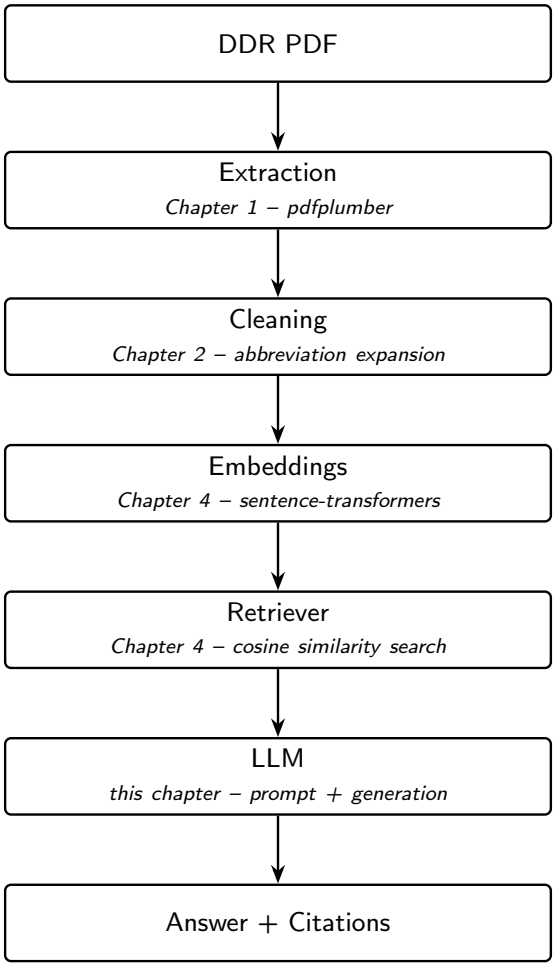
Engineering Translation: Prompt

A **prompt** is the instructions-plus-evidence packet you hand that new hire before asking them to write anything — the equivalent of handing them the specific job's DDRs and a clear brief, instead of just asking "what usually happens in a fishing job?" and hoping they don't improvise.

The critical design decision is the prompt: instruct the model explicitly to answer only from the provided context, and to cite which source each part of the answer came from. This doesn't make

hallucination impossible — Chapter 10 covers stronger mitigations — but it is the foundation everything else builds on.

Here's the complete pipeline you've actually built, chapter by chapter — every arrow below is real, working code you've already written:



This is deliberately the simplest version of this pipeline that could possibly work — brute-force search, whole-document embeddings, no reranking, no traceability guardrails beyond a citation list. Part II rebuilds nearly every arrow in this diagram to survive a real, full-scale archive.

5.5. Implementation

5.5.1. Step 1: write the instructions the model must follow

What problem are we solving?

Make sure the model answers from the evidence you hand it, not from whatever it remembers from training — and make sure it tells you which report each part of the answer came from.

Inputs

- None yet — this is a fixed template, filled in for each question later.

Expected Output

A reusable prompt template with two blanks: the evidence, and the question.

```
# code/chapter_05/first_rag.py
PROMPT_TEMPLATE = """Answer the question using ONLY the evidence below.
If the evidence doesn't contain the answer, say so - do not guess.
Cite which report each part of your answer comes from.

Evidence:
{evidence}

Question: {question}
Answer:"""
```

What just happened?

This is the "brief" you hand the model every single time: answer only from what's provided, admit it when the evidence doesn't cover the question, and always say which report backs each claim. Everything else in this chapter exists to fill in `{evidence}` and `{question}` correctly.

5.5.2. Step 2: assemble the evidence into that template

What problem are we solving?

Take the reports Chapter 4's retriever found and package them, labelled by filename, into the prompt template's `{evidence}` blank.

Inputs

- A question string.
- The retrieved `(filename, score)` pairs from Chapter 4's `search`.
- The full filename and text lists from Chapter 4's `load_chunks`.

Expected Output

One complete prompt string, evidence labelled by source filename, ready to send to a model.

```
from pathlib import Path

from semantic_search import embed_texts, load_chunks, search  # from Chapter 4

def build_prompt(question: str, retrieved: list[tuple[str, float]],
                 filenames: list[str], texts: list[str]) -> str:
    evidence_blocks = []
    for filename, _score in retrieved:
        text = texts[filenames.index(filename)]
        evidence_blocks.append(f"[{filename}]\n{text}")
```

```
    return PROMPT_TEMPLATE.format(
        evidence="\n\n".join(evidence_blocks),
        question=question,
    )
```

What just happened?

For every report Chapter 4's retriever returned, this tags its full text with its own filename in square brackets, then stitches all of them together into the `{evidence}` section of the prompt. Every fact the model can possibly cite is now labelled with exactly which report it came from.

5.5.3. Step 3: retrieve, then generate, then report the sources

What problem are we solving?

Chain retrieval and generation into one function: given a question, find the evidence, ask the model to answer from it, and hand back both the answer and the exact list of reports it was allowed to use.

Inputs

- A question string.
- The embedding model, filenames, texts, and embeddings from Chapter 4.
- `llm_call`: any function that takes a prompt string and returns the model's answer.

Expected Output

A tuple: the generated answer text, and the list of report filenames that were actually retrieved and shown to the model.

```
def answer_question(question: str, model, filenames, texts, embeddings,
                    llm_call) -> tuple[str, list[str]]:
    retrieved = search(model, question, filenames, embeddings, top_k=3)
    prompt = build_prompt(question, retrieved, filenames, texts)
    answer_text = llm_call(prompt)  # plug in your LLM of choice
    evidence = [filename for filename, _score in retrieved]
    return answer_text, evidence
```

What just happened?

This is the whole system end to end: retrieve the top matching reports, build a prompt from them, generate an answer, and return that answer alongside the exact evidence list it was built from — so you (or an engineer reviewing the answer) can always check the citation against the real reports, rather than taking the model's word for it.

`llm_call` is intentionally a plain function argument, not a hardcoded API client: plug in a local model, an API-based model, or — while you're first testing the retrieval and prompt logic — a stub that just echoes the evidence back, so you can verify the pipeline end to end before spending a single API call.

5.5.4. Step 4: plug in a real local model

The stub proved the plumbing. Now generate for real — locally, so no report text ever leaves your machine. Ollama runs a small open model on your own hardware. Install it, pull a model, and start the server:

```
# install Ollama from https://ollama.com, then:
ollama pull qwen2.5:7b-instruct
ollama serve            # leave this running in its own terminal
```

`qwen2.5:7b-instruct` is a small, capable default, but nothing here is tied to it — pass any model you've pulled to `ollama_llm_call(model=...)`. Pick whatever runs comfortably on your hardware: `llama3.1:8b` or `mistral` are similar in size, `qwen2.5:14b` is stronger if you have the memory for it, and swapping the one HTTP call for a hosted API's would let you point at a cloud model instead. The retrieval and prompt work above doesn't change either way — only which model reads the evidence.

`ollama_llm_call` talks to that local server over plain HTTP — standard library only, no extra package to install, and a clear message instead of a crash if Ollama isn't running:

```
import json
import urllib.error
import urllib.request

def ollama_llm_call(prompt: str, model: str = "qwen2.5:7b-instruct") -> str:
    payload = json.dumps({"model": model, "prompt": prompt, "stream":
↪   False}).encode()
    request = urllib.request.Request("http://localhost:11434/api/generate",
↪   data=payload,
                                     headers={"Content-Type":
↪   "application/json"})
    try:
        with urllib.request.urlopen(request, timeout=120) as response:
            return json.loads(response.read())["response"].strip()
    except (urllib.error.URLError, OSError) as error:
        return f"[Ollama not reachable: {error}. See setup above; retrieval still
         ↪  ran.]"
```

Run the whole system against the single best-matching report:

```
python code/chapter_05/first_rag.py "What led to the packers failing to set?"
```

> One real run — qwen2.5:7b-instruct, top_k=1 (your wording will differ)
>
> ```
> Question: What led to the packers failing to set?
> ```

```
Answer:
According to the report, pressures indicated that the ball did not seat
and the packers did not set during an attempt to set packers multiple
times.

Reference:
[FORGE-16A-78-32_Drilling_049_2020-12-07.txt] Page 15: "Production
Drilling Other Pressures indicated that ball did not seat and packers
did not set."

Sources:
  FORGE-16A-78-32_Drilling_049_2020-12-07.txt
    "--- Page 1 --- RPT DATE:12/07/2020 DAILY DRILLING REPORT RPT NUM.:49
     ... WELL NAME:FORGE 16A [78]-32 ..."
```

That's a real RAG answer: a plain-English cause, grounded in report #49's own words, with the source report named and an excerpt you can open and check. **Your exact wording will differ** — LLM output isn't deterministic (the Production Reality note below says why) — but the grounded fact, the ball that didn't seat, comes back every time, because it's sitting in the evidence the model was handed.

Two honest limits are already visible in that single run:

- **Evidence width matters.** This ran at `top_k=1` — the one closest report. Widen it to `top_k=3` and the model is handed three full pages of tables at once; in testing it lost the packer line in the noise and answered "the report doesn't mention it." That isn't the model being dim — it's the whole-document granularity problem from Chapter 4, biting generation now instead of retrieval. Chapter 7's chunking is what fixes it.
- **The model can still invent a citation.** Notice the answer cites "Page 15." These reports are a single page — there is no page 15. The *fact* is grounded; the *page number* is fabricated. A prompt asking for citations gets citation-shaped text, not guaranteed-correct citations. Chapter 10 wires the real page in from metadata so the model can't make one up.

5.6. Production Reality

This chapter's prompt says "answer only from the evidence" — but an instruction in a prompt is a strong nudge, not a guarantee. Real systems have to plan around what happens when that nudge isn't enough:

- models don't follow instructions perfectly every time — occasionally one will still answer from general knowledge instead of the evidence, or blend the two without saying so
- the same question, asked twice, can produce two differently worded answers — LLM output isn't fully deterministic, which matters when someone asks "why did it say something different yesterday?"
- long evidence sections eventually hit the model's context window limit — a real archive's retrieved chunks can't just grow without bound

- every call to a hosted LLM API costs money and takes time; a system answering hundreds of engineering questions a day needs a cost and latency budget, not just a working prompt

None of this means RAG doesn't work — it means "answer only from evidence" is a design goal you build toward, not a switch you flip. Chapter 10 covers the guardrails that close this gap further.

5.7. Practical exercise

Beginner

Try it yourself: Wire up `answer_question` with a stub `llm_call` that simply returns the evidence text, run it for "What led to the fishing operation on report #50?" with `top_k=3`, and print the evidence list.

You'll know it worked when: you can see exactly which three reports came back — and, before reading further, decide for yourself whether that list actually contains what the target answer above needs.

5.8. Field notes

⚠ Field notes: the report the question is about isn't in the evidence

Query: `"What led to the fishing operation on report #50?"`, `top_k=3`.
Result: retrieval returns report #49 (rank 1) plus two other reports — but **not report #50**, the report the question is literally about. The full ranking:

```
1. 0.4540  Drilling_049_2020-12-07   <- packers fail (correctly retrieved)
2. 0.3639  Drilling_038_2020-11-26
3. 0.3469  Drilling_003_2020-10-22
...
7. 0.3190  Drilling_050_2020-12-08   <- the fishing report itself
```

Why: report #50's own text — bit-mill fishing narrative buried among casing tables, mud tables, and a full BHA component list — is, at whole-document granularity, more similar to *several other reports'* tables than it is to a question about fishing. This is the same granularity problem Chapter 4 already flagged; here it has a sharper consequence.
Lesson: this is exactly the failure mode this chapter's own Key Takeaways warn about — and it's a dangerous one specifically *because* the question names "report #50" directly. A careless `llm_call` implementation could echo that report number back in its answer without ever having actually retrieved report #50's content, producing a citation-shaped sentence that *looks* grounded and isn't. Always check what's actually in the evidence list — not what the question happens to mention — before trusting an answer. Chapter 7's chunking and Chapter 9's hybrid retrieval are what actually close this gap; Part I's simple system doesn't, and pretending otherwise would defeat the entire point of this book.

5.9. Challenge exercise

Advanced

Challenge: Connect a real LLM (local via `transformers`/`ollama`, or a hosted API) as `llm_call`, and add a check that rejects the answer if it mentions a report filename that wasn't in the retrieved evidence list — a first, crude hallucination guard. Then ask it a question the ten-report sample archive genuinely can't answer (e.g. "what happened on report #100?") and confirm it says so rather than inventing an answer. A reference solution is in `code/chapter_05/challenge/`.

5.10. Key takeaways

- RAG is retrieval, then generation — two separate, individually debuggable steps, not one opaque black box.
- If retrieval is wrong, generation cannot save you — always verify the evidence list before trusting the generated answer.
- A prompt that instructs "answer only from evidence, cite sources" is necessary but not sufficient for trustworthy answers. Part II builds the guardrails that make it sufficient.

5.11. Repository files

File	Purpose
`code/chapter_05/first_rag.py`	Prompt assembly and evidence-tracked question answering
`notebooks/chapter_05_explore.ipynb`	Interactive companion notebook

CHECKPOINT — Chapter 5

- ☒ Explained RAG as two separate, inspectable steps: retrieve, then generate
- ☒ Assembled retrieved evidence into a prompt that forces answers from that evidence only
- ☒ Generated a real answer with a local model (Ollama), returned with its source reports and excerpts
- ☒ Caught a real case where the cited report wasn't actually in the evidence

WHAT YOU BUILT

`first_rag.py` — an Industrial RAG system: ask it a question in plain English, and it retrieves evidence, generates an answer with a local model, and hands back the source reports and excerpts that answer is traceable to. This is Part I's capstone artifact.

> **Try it in the browser: the companion app**
>
> The optional DDR RAG Companion App (`book/app/`) wraps this same retrieve-then-generate-then-cite flow in a small Streamlit screen. Run `streamlit run book/app/streamlit_app.py`, pick a question, and see the evidence cards, the local-model answer, and a "Why this answer?" panel side by side. It reuses this chapter's code (plus Chapter 9's hybrid retrieval) — nothing new to learn, just the payoff on one screen.

5.12. What can you do now that you couldn't do before?

You can ask an engineering question in plain English and get back a generated answer with an explicit, checkable list of exactly which reports it came from — not just ranked passages you'd still have to read and synthesise yourself.

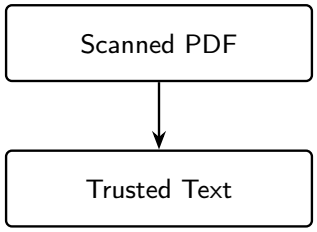
5.13. Suggested next step

Coming up in Chapter 6: What you've built in Part I works — on ten clean, digital, hand-picked reports out of the full 76-report Utah FORGE archive. Part II industrialises this system against the reality of a full archive: retrieval that has to be both fast and precise across every report, and answers that have to survive scrutiny from someone who wasn't in the room when they were generated. Chapter 6 starts with the first thing that breaks in a typical real-world archive: scanned reports with no extractable text at all.

Part III.

Part II — Industrialising the System

6. Scanned Reports and OCR Quality Gates



Progress `######.....` **6 / 12** · **Estimated time:** 45–60 min · **Difficulty:** Intermediate

6.1. Learning objectives

By the end of this chapter, you will be able to:

- Recognise when a DDR PDF has no extractable text and needs OCR.
- Make a scanned-style page from a digital report, recover its text with OCR, and score whether that text is trustworthy enough to index.
- Build a quality gate that rejects bad OCR output *before* it enters your search index, rather than trying to fix it after the fact.

6.2. Operational Problem

Sean, the production engineer, asks a question that matters long before any answer gets generated: *"Can you trust the text this archive extracted — even from a report that was scanned instead of born digital?"* Every one of the 76 Utah FORGE reports used in this book is digital-native — extracting all 76 with Chapter 1's own `extract_text()` confirms zero unparsed PDFs. (The companion pipeline's QA/QC pass reports the same result across 227 source files — the larger, pre-deduplication count from the raw public archive before Chapter 8's `build_full_archive.py` collapsed it down to 76; that raw archive isn't part of this book's repository, so the 227-file figure isn't independently checkable here.) You got lucky: WellEz, the software that generated these reports, embeds real character data on every page. Not every archive is this well-behaved. Faxed reports, reports scanned from paper files, and photos of a logbook page all produce PDFs with no character data at all — just a picture of text. Chapter 1's `extract_text()` handled this gracefully (an empty string, not a crash), but a real, larger archive needs more than graceful failure: it needs OCR to recover the text, and a way to know whether to trust what OCR gives back.

6.3. Example DDR extract

OCR output before and after quality gating

```
Clean digital extraction (this is what every Utah FORGE report gives):
  "During the slide lost tool face and became assembly became stuck"

Degraded OCR output from a hypothetical poor scan of the same line:
  "Dur1ng the s1ide los+ t00l face and became assembIy became stu(k"
```

The second block is exactly the kind of text a naive pipeline would embed and index without complaint — and exactly the kind of text a quality gate should catch and route to a review queue instead. That degraded line is illustrative; further down, "Seeing it on a real scanned page" makes an actual scanned page from report #38 and runs the whole extract-fails → OCR → gate round-trip on it, no hypotheticals.

6.4. Theory

Getting OCR to run is the easy part — `pytesseract`, cloud OCR APIs, and PDF-rendering libraries all do a competent job of turning a page image into text. The hard part, and the one that actually determines whether your downstream system is trustworthy, is deciding **whether to trust the output at all**. A quality gate scores OCR text against a handful of cheap heuristics — is there enough text, is the alphabetic-to-symbol ratio sane, is there suspicious repeated garbage — and rejects anything that fails, routing it to manual review instead of silently indexing noise.

Engineering Translation: Quality gate

A **quality gate** is a mud check before you commit to pumping downhole: a handful of cheap tests run before you trust what comes next. Passing doesn't prove the text is perfect, the same way a good mud check doesn't prove there's no problem downhole — it just screens out the obviously bad cases before they cost you.

Engineering Translation: Flag

A **flag** here is one checkbox on an inspection form — "too little text" either applies or it doesn't. One flag firing on its own usually isn't grounds for rejection (a short, sparse-but-real report can trip one check honestly); it's when *several* flags fire together that the page is actually suspect.

This chapter's companion pipeline, **DDR_UTAH_FORGE**, implements exactly this pattern in `src/rag_pdf/ocr_quality.py` — even though, as noted above, this particular archive never needs to exercise it. That's by design: the quality gate runs unconditionally on every extracted page, digital or scanned, as a defensive check. Its `evaluate_ocr_quality()` function checks four independent flags — `low_text_density`, `high_symbol_ratio`, `repeated_garbage`, `low_line_cou`

`nt` — and rejects a page only once **two or more** flags fire, so no single noisy-but-legitimate page (e.g. a short report with a lot of numeric data) gets discarded on one false positive.

```
extracted text
   ↓
compute four flags
(low_text_density, high_symbol_ratio,
 repeated_garbage, low_line_count)
   ↓
count how many are True
   ↓
2 or more active?
   ↓
  yes -> reject, flag for review
  no  -> accept into index
```

6.5. Implementation

Build a simplified version of the same idea.

What problem are we solving?

Decide whether a page's extracted text is trustworthy enough to add to the search index, or whether it should be set aside for a human to check instead.

Inputs

- Extracted text (from OCR, or from Chapter 1's digital extraction).
- Thresholds: minimum characters, minimum alphabetic words, maximum symbol ratio, and how many flags must fire before rejecting.

Expected Output

A dictionary reporting the decision and which specific checks failed, for example:

```
{"reject_ocr": True, "flags": {"low_text_density": True, "high_symbol_ratio": True}}
```

```python
# code/chapter_06/ocr_quality_gate.py
import re

def evaluate_ocr_quality(text: str, min_chars: int = 200,
                         min_alpha_words: int = 30,
                         max_symbol_ratio: float = 0.35,
                         reject_min_flags: int = 2) -> dict:
    alpha_tokens = re.findall(r"[A-Za-z]+", text)
    non_ws = [c for c in text if not c.isspace()]
    noisy = sum(1 for c in non_ws if not c.isalnum() and c not in
↪   set("%.,()/-:'"))
```

```
    symbol_ratio = noisy / max(len(non_ws), 1)

    flags = {
        "low_text_density": len(text) < min_chars or len(alpha_tokens) <
        ↪  min_alpha_words,
        "high_symbol_ratio": symbol_ratio > max_symbol_ratio,
    }
    active = [k for k, v in flags.items() if v]
    return {"reject_ocr": len(active) >= reject_min_flags, "flags": flags}
```

What just happened?

This measures two things about the text: how much of it is real, readable words (`low_text_density`), and how much of it is odd symbols that shouldn't be there (`high_symbol_ratio`). Each check is a simple yes/no. Only when two or more of those checks say "yes, this looks wrong" does the function recommend rejecting the page — a single unusual character never sinks an otherwise legitimate report.

Test it against a real Utah FORGE report's clean text (it should pass easily — that text has a healthy word count and almost no noise symbols) and against the corrupted example above (it should fail on both flags).

The full production version — including the repeated-garbage detector and line-count check — is in the companion repository at `src/rag_pdf/ocr_quality.py`, alongside `src/rag_pdf/rotation_handler.py` for detecting and correcting upside-down or sideways scans, and `src/rag_pdf/table_ocr_handoff.py` for the case where a *table* (not narrative text) needs OCR fallback.

6.6. Seeing it on a real scanned page

The gate scores text — but where does bad OCR text actually come from? This archive is entirely digital, so to see the whole problem end to end you have to make a scanned page first. `make_scanned_example.py` rasterizes a real report to a page image and writes it back out as an image-only PDF: pictures of text, no character data, exactly what a scan of a paper report looks like to software.

Install the OCR tools (Chapter 6 only — nothing earlier needs them):

```
pip install pytesseract pdf2image
# plus the system tools they wrap:
#   macOS:  brew install tesseract poppler
#   Ubuntu: apt install tesseract-ocr poppler-utils
```

Then run the round-trip — make the scanned page from report #38, try to extract it, OCR it, and gate both:

```
python code/chapter_06/make_scanned_example.py
```

A real digital-vs-OCR round-trip on report #38

```
Digital extraction:   3851 chars   stuck line present: True
pdfplumber on scan:      0 chars   stuck line present: False
OCR on scan:          3376 chars   stuck line present: True

Quality gate:
  digital  reject=False  flags={'low_text_density': False, 'high_symbol_ratio': False}
  OCR      reject=False  flags={'low_text_density': False, 'high_symbol_ratio': False}
```

Three real facts fall out of that run:

- **Chapter 1's extraction returns nothing on a scan.** `pdfplumber` on the image-only PDF recovers **0 characters** — there's no character data to read, only a picture. That's the exact failure a real archive's scanned reports hit, reproduced here on a real FORGE page instead of a hypothetical one.
- **OCR recovers the text.** Tesseract reads 3,376 of the digital version's 3,851 characters back off the image, and the stuck-pipe line comes through intact: `During the slide lost tool face and became assembly became stuck`.
- **The gate passes clean OCR.** This scan was sharp, so the recovered text is good and the gate accepts it — which is the point of a gate as much as rejection is: it has to wave good OCR through as readily as it stops bad OCR. The degraded line back in the Example section is the other half — what the gate catches when a scan *isn't* this clean.

6.7. Production Reality

This chapter's quality gate answers one narrow question: should this specific page be trusted? It doesn't answer what happens next. A real archive needs a plan for that too:

- a flagged page needs an actual person to review it — a gate only earns its keep if someone owns the review queue it creates, otherwise flagged pages just pile up unread
- rotated or upside-down scans (common when a paper report goes into a scanner sideways) need correcting *before* OCR runs at all — that's a separate step from quality scoring, handled by the companion `rotation_handler.py`
- handwritten margin notes on an otherwise-typed report often don't OCR into anything usable — the gate will correctly flag them as low-density, but "OCR failed" and "this needs a human to transcribe handwriting" call for different follow-up
- an archive spanning multiple regions or operators may include reports in more than one language, which changes which OCR language pack — and which quality thresholds — actually apply

None of this shows up in the all-digital Utah FORGE archive, which is exactly why it's easy to forget about until a real archive containing any of the above lands on your desk.

6.8. Practical exercise

Beginner

Try it yourself: Run `evaluate_ocr_quality()` against the extracted text of report #38 (clean) and the degraded example string above.

You'll know it worked when: the real report text passes cleanly, and you can explain, from the flags returned, exactly why the degraded text fails.

6.9. Field notes

⚠ Field notes: checking the gate doesn't cry wolf

Action: run `evaluate_ocr_quality()` against all ten real sample reports, including report #3 — the sparsest one in the archive, written before the well was even spudded, with most of its tables entirely empty.
Result: every single report passes cleanly. Report #3, the thinnest of the ten, still has 2,235 characters and 361 alphabetic words — comfortably clear of the 200-character / 30-word thresholds:

```
chars=2235 alpha_words=361 symbol_ratio=0.006  reject=False
```

Why: even a nearly-empty DDR carries a full page of section headers, column labels, and boilerplate ("WELL/JOB INFORMATION", "PRESENT OPERATIONS", table column names) regardless of how little actually happened that day. That structural text alone is enough to clear the density thresholds.
Lesson: a quality gate is only trustworthy if it doesn't also reject legitimate sparse content — a gate that flagged every quiet day as "probably OCR garbage" would be useless in practice, burying real pages under false alarms. Checking the gate against your *most* sparse real documents, not just your noisiest fake ones, is how you find that out before it costs you.

6.10. Challenge exercise

Intermediate

Challenge: This archive is 100% digital, so you can't test the gate against a real scanned Utah FORGE page. Instead, write a small function that synthetically corrupts clean text (swap random characters for lookalikes, inject repeated tokens) at increasing severity, and find the corruption level at which `evaluate_ocr_quality()` starts rejecting it. This is a legitimate way to stress-test a quality gate before you ever encounter the real noisy data it's meant to catch.

6.11. Key takeaways

- OCR quality gating is cheap, heuristic, and worth running unconditionally — even an archive that's currently 100% digital may not stay that way, and the check costs almost nothing on clean text.
- Require multiple independent signals to agree before rejecting a page; a single heuristic alone produces too many false positives.
- Reject-and-flag-for-review beats silently-index-anyway every time traceability matters more than completeness.

6.12. Repository files

File	Purpose
`code/chapter_06/ocr_quality_gate.py`	Simplified OCR quality gate
`code/chapter_06/make_scanned_example.py`	Make an image-only PDF from a real report, then OCR and gate it
`DDR_UTAH_FORGE/src/rag_pdf/ocr_quality.py`	Production quality gate (companion repo)
`DDR_UTAH_FORGE/src/rag_pdf/rotation_handler.py`	Scan rotation detection (companion repo)

CHECKPOINT — Chapter 6

- ☒ Recognised when a DDR PDF needs OCR instead of direct extraction
- ☒ Made a real scanned-style page from a digital report and recovered its text with OCR
- ☒ Explained why raw OCR output can't be trusted without scoring it first
- ☒ Built a quality gate that rejects bad text before it reaches the index
- ☒ Verified the gate doesn't false-positive on legitimately sparse reports

WHAT YOU BUILT

`ocr_quality_gate.py` — an OCR quality gate: score any extracted text against four cheap heuristics and decide, automatically, whether it's trustworthy enough to index or needs a human to review it first.

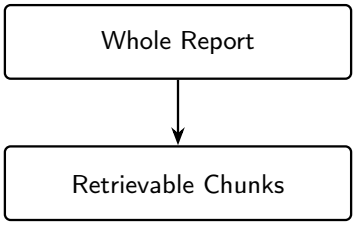
6.13. What can you do now that you couldn't do before?

You can tell, automatically, whether a page of extracted or OCR'd text is trustworthy enough to search on — and route the pages that aren't to a human for review, instead of silently indexing garbage alongside good reports.

6.14. Suggested next step

Coming up in Chapter 7: Once you trust the text coming out of OCR and digital extraction alike, the next question is how to break it into pieces small enough to embed and retrieve — without cutting a stuck-pipe narrative in half across two chunks. That's Chapter 7.

7. Chunking That Respects Report Structure



Progress ####### . . . . . **7 / 12** · **Estimated time:** 45–60 min · **Difficulty:** Intermediate

7.1. Learning objectives

By the end of this chapter, you will be able to:

- Explain why chunk size is measured in tokens, not characters or words.
- Implement token-based chunking with overlap using `tiktoken`.
- Implement segment-aware chunking that breaks at natural report boundaries (section headers) instead of at an arbitrary character count.

7.2. Operational Problem

Oumy, the drilling engineer, notices something odd while reviewing retrieval results: why does the exact sentence she needs keep landing right on a chunk boundary? Chapters 1–5 embedded each report as a single unit — fine for a ten-report sample corpus, unworkable at the full archive's scale. A single Utah FORGE report already has ten distinct sections (WELL/JOB INFORMATION, BOP, CASING, MUD, DRILL BITS, PUMPS, BHA, SURVEY DATA, CONSUMABLES, TIME BREAKDOWN) packed onto one page. Embed too much of that in one vector and the embedding blurs the casing programme together with the stuck-pipe narrative, hurting retrieval precision. Split naively at a fixed character count and you'll cut report #38's stuck-pipe sentence in half mid-word, right across a chunk boundary — the exact passage you most need to retrieve becomes the two chunks least likely to score well.

7.3. Example DDR extract

Why naive splitting fails, using real report #38 text

```
Naive fixed-character split, mid-sentence:

Chunk A: "...23:30 04:00 4.5 Drill From 6,360' to 6,507', (147') Total,
          32.6' feet per hour. WOB 20 TO 35k, Rotary 50, Torque 6,500,
          SPP 3200-3400 GPM 560, DIFF 200-300psi During the slide lost
          tool face and became assembly became st"
Chunk B: "uck 04:00 06:00 2.0 Work pipe, circulate lube sweep, work
          tool back in position Pipe free"
```

A query about "what happened when the assembly got stuck" now has its key evidence split across two separately-scored chunks — neither of which alone contains the complete sentence.

7.4. Theory

Two ideas, used together, fix this:

1. **Token-based sizing.** Language models and embedding models operate on tokens, not characters — and token count is what actually determines whether a chunk fits a model's context window and how much a given chunk "costs" downstream. Chunking by token count (with a library like `tiktoken`) is more accurate than chunking by character or word count.
2. **Segment-aware boundaries.** Rather than cutting at a fixed token count regardless of content, detect natural boundaries — section headers like `TIME BREAKDOWN` or `SURVEY DATA`, which every Utah FORGE report uses consistently — and prefer to split there. A chunk that ends at a genuine section break is far more likely to be a self-contained, retrievable unit of meaning than one that ends mid-sentence because a counter hit its limit.

Engineering Translation: Token

A **token** is the unit a language model actually counts in — roughly a word or a word-fragment, not a character and not quite a word either. It's the same idea as a rig measuring progress in footage drilled, not characters typed on the report: footage is the unit that actually determines what the operation costs and how far along it is.

Engineering Translation: Overlap

Overlap between chunks is like re-surveying the last few feet of the previous run before starting the next one: a small deliberate repeat at the seam, so nothing that mattered right at the boundary gets lost between the two pieces.

The companion pipeline, **DDR_UTAH_FORGE**, implements both in `src/rag_pdf/chunking.py`. `chunk_text_by_tokens()` does token-based splitting with configurable overlap (so context isn't lost entirely at a boundary), falling back to a word-based approximation if `tiktoken` isn't available.

`split_text_for_segment_aware_chunking()` goes further: it walks the text line by line, detects boundary patterns, and returns a list of `SegmentBlock` objects — each one bundling together a title, its text, and a `boundary_type` (`MATCH`, `INSERT`, or `CONTINUATION`) recording *why* the segment starts where it does, the same way an equipment spec sheet bundles a component's dimensions together instead of scattering them as loose numbers.

Run against the real 76-report archive, this pipeline's chunking produces **1,428 chunks** — an average of about 19 chunks per report, reflecting just how much structured content (header, seven data tables, and a narrative time breakdown) each single-page DDR actually contains.

Token-based chunking with overlap looks like this — each chunk shares a few tokens with its neighbour, so no boundary loses context entirely:

```
tokens:   t1  t2  t3  t4  t5  t6  t7  t8  t9  t10 t11 t12

chunk 1: [t1  t2  t3  t4  t5  t6]
chunk 2:              [t5  t6  t7  t8  t9  t10]
chunk 3:                          [t9  t10 t11 t12]
                       ^^^^^^         ^^^^^^
                       overlap        overlap
```

7.5. Implementation

What problem are we solving?

Split a report's text into pieces small enough to embed precisely, without severing the exact sentence a future query needs right at a chunk boundary.

Inputs

- Full report text (a Python string).
- `chunk_tokens`: how many tokens each chunk should hold.
- `overlap_tokens`: how many tokens each chunk repeats from the end of the previous one.

Expected Output

A list of text chunks, each roughly `chunk_tokens` tokens long, each overlapping the next by `overlap_tokens` tokens.

```python
# code/chapter_07/token_chunking.py
import tiktoken

def chunk_text_by_tokens(text: str, chunk_tokens: int = 200,
                         overlap_tokens: int = 40) -> list[str]:
    enc = tiktoken.get_encoding("cl100k_base")
    tokens = enc.encode(text.strip())
    chunks, start = [], 0
    while start < len(tokens):
        end = min(len(tokens), start + chunk_tokens)
```

```
        chunks.append(enc.decode(tokens[start:end]).strip())
        if end == len(tokens):
            break
        start = max(0, end - overlap_tokens)
    return chunks
```

What just happened?

The text got converted into tokens, then sliced into fixed-size windows that each step forward by less than a full window's width — that gap is the overlap, and it's what keeps a sentence sitting near a boundary from being fully lost to one side or the other.

This is a direct simplification of `chunk_text_by_tokens()` in the companion repository — same sliding-window-with-overlap logic, without the word-based fallback path. Read the full version, plus `split_text_for_segment_aware_chunking_with_patterns()`, in `src/rag_pdf/chunking.py`.

7.5.1. Step 2: give every chunk back its page number

What problem are we solving?

`chunk_text_by_tokens()` above returns plain strings — once a chunk is made, nothing about it remembers which page it came from. That's fine until Chapter 10 needs to cite a page number for a claim, and discovers there isn't one to cite. Chapter 1's `extract_text()` already writes a `--- Page N ---` marker before each page's text; this step reads those markers back out and chunks each page separately, so no chunk can ever straddle a page boundary or lose track of which one it belongs to.

Inputs

- The full `--- Page N ---`-marked text Chapter 1's `extract_text()` produces, not a single page's text in isolation.

Expected Output

A list of `(page_number, chunk_text)` pairs — the same chunks as before, each now labelled with the real page it came from.

```
import re

PAGE_MARKER = re.compile(r"--- Page (\d+) ---\n?")

def split_pages(pages_text: str) -> list[tuple[int, str]]:
    matches = list(PAGE_MARKER.finditer(pages_text))
    pages = []
    for i, match in enumerate(matches):
        page_number = int(match.group(1))
        start = match.end()
        end = matches[i + 1].start() if i + 1 < len(matches) else len(pages_text)
        pages.append((page_number, pages_text[start:end].strip()))
    return pages
```

```
def chunk_pages_by_tokens(pages_text: str, chunk_tokens: int = 60,
                          overlap_tokens: int = 15) -> list[tuple[int, str]]:
    chunks_with_pages = []
    for page_number, page_text in split_pages(pages_text):
        for chunk in chunk_text_by_tokens(page_text, chunk_tokens,
         ↪  overlap_tokens):
            chunks_with_pages.append((page_number, chunk))
    return chunks_with_pages
```

What just happened?

`split_pages` undoes Chapter 1's markers, turning the joined text back into `(page_number, page_text)` pairs. `chunk_pages_by_tokens` then chunks *each page's text separately* — reusing the exact function from Step 1 — and tags every resulting chunk with the page it came from. Chunking per page instead of chunking the whole joined document is what actually guarantees no chunk can straddle two pages: the token window in Step 1 never sees more than one page's text at a time.

Run this against report #38's real text and it produces **37 chunks**, every one correctly labelled `page 1` — because every DDR in this sample archive is a single page (Chapter 1). That's not a very demanding test of the page-tracking logic, and it's worth being honest about that: this step earns its keep the day someone points this code at a multi-page report, not on this archive. The `became stuck` sentence still lands whole inside one chunk, exactly as in Step 1 — page tracking didn't cost anything the chunking itself hadn't already paid for.

7.6. Production Reality

This chapter's Field Notes (below) shows the heading heuristic getting a perfect score — because Utah FORGE's reports all come from one piece of software, using one consistent template. A larger, multi-operator archive rarely offers that luxury:

- different reporting software formats section headers differently — all-caps, title case, bolded, or not visually marked at all — so a heading heuristic tuned to one archive's convention may need retuning, or replacing entirely, for another
- OCR'd pages (Chapter 6) can corrupt exactly the visual cues a heading heuristic depends on — a garbled, partly-misread line may no longer look like a clean heading at all
- `tiktoken`'s `cl100k_base` encoding matches a specific family of language models; a different embedding or generation model in your pipeline may tokenize the same text differently, so `chunk_tokens=200` isn't automatically the same size everywhere
- overlap improves retrieval at chunk boundaries, but it also multiplies how much text you store and embed — worth measuring deliberately once an archive holds thousands of reports instead of ten

7.7. Practical exercise

Beginner

Notice this uses smaller numbers than the `chunk_tokens=200, overlap_tokens=40` shown above — deliberately. Every DDR in this sample is a single page, so at 200/40 one report only splits into 11 chunks; at 60/15 it splits into 37. More chunks from one short report makes the idea of chunking itself easier to see and check by eye, which is the point of this exercise. Real archives with real budgets tend to land closer to 200/40 or larger — see the Production Reality note above.

Try it yourself: Chunk report #38's full text with `chunk_tokens=60, overlap_tokens=15`, and confirm the stuck-pipe sentence ("During the slide lost tool face and became assembly became stuck") appears complete within at least one chunk rather than split across two.

You'll know it worked when: you can point to a single chunk that contains the full sentence, from "During the slide" through "became stuck."

7.8. Field notes

⚠ Field notes: does the heading heuristic ever guess wrong?

Action: run the "all-caps line under 6 words = heading" rule against every line of all ten real reports, and list every match.
Result: every single match — across all ten reports — is a genuine section header: `BHA`, `BOP`, `CASING`, `CONSUMABLES`, `DRILL BITS`, `MUD`, `PUMPS`, `SURVEY DATA`, `TIME BREAKDOWN`, `WELL/JOB INFORMATION`, plus the completion report's `FLUID DATA`, `FRAC`, `PERFORATIONS`, `SAFETY`, and `TIME LOG`. Zero false positives — no data row anywhere in the archive happens to look like a heading.
Why: this isn't luck — it reflects something real about how DDR software generates these reports. Section headers are short, fixed, all-caps labels from a small controlled vocabulary; data rows always carry numbers, units, or lowercase narrative text that the pattern correctly excludes.
Lesson: a heuristic that "seems reasonable" is still a guess until you've run it against real data and checked every match by hand. This one happens to be exactly right on this archive — but that's a claim worth verifying yourself before trusting it on a different one, not an assumption to inherit.

7.9. Challenge exercise

Intermediate

Challenge: Implement a minimal segment-aware splitter that treats any all-caps line under 6 words (like `MUD`, `DRILL BITS`, `SURVEY DATA`, `TIME BREAKDOWN`) as a section heading and starts a new segment there. Run it against report #38's full text and confirm it produces one segment per section rather than one blended mass of casing, mud, and time-breakdown text. A reference solution

— and the full production logic — is in `code/chapter_07/challenge/` and `DDR_UTAH_FORGE/src/rag_pdf/chunking.py` respectively.

7.10. Key takeaways

- Chunk size belongs in tokens, because that's the unit every downstream model actually consumes.
- Overlap prevents context from vanishing at a chunk boundary, at the cost of some redundancy in the index.
- Segment-aware boundaries produce chunks that are more likely to be self-contained, retrievable units — worth the extra complexity once reports have real internal structure, which every DDR in this archive does.
- A chunk that doesn't know its own page number can't be cited by page later — chunk per page, not per document, if Chapter 10's citations need to mean anything.

7.11. Repository files

File	Purpose
`code/chapter_07/token_chunking.py`	Token-based chunker with overlap, plus page-aware chunking
`DDR_UTAH_FORGE/src/rag_pdf/chunking.py`	Production token + segment-aware chunking (companion repo)

CHECKPOINT — Chapter 7

- ☒ Explained why chunk size is measured in tokens, not characters
- ☒ Implemented token-based chunking with overlap
- ☒ Verified a key sentence survives intact inside one chunk instead of splitting across two
- ☒ Gave every chunk back the real page number it came from
- ☒ Sketched a segment-aware splitter that respects section headers

WHAT YOU BUILT

`token_chunking.py` — a token-based chunker with overlap, turning any whole report into retrievable pieces sized for precise embedding instead of one blurry whole-document vector, each one labelled with the exact page it came from.

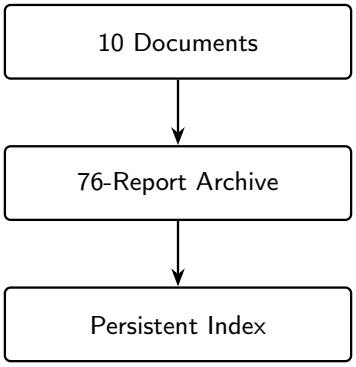
7.12. What can you do now that you couldn't do before?

You can split a report into chunks small enough to embed precisely, without cutting the exact sentence you need to retrieve in half at a chunk boundary — and every chunk still knows which page it came from, so nothing downstream has to guess.

7.13. Suggested next step

Coming up in Chapter 8: Well-formed chunks still need somewhere fast to live. Chapter 8 replaces Chapter 4's brute-force cosine search with a real vector database, and scales retrieval from ten documents to the full 76-report archive.

8. Vector Databases at Scale



Progress `########....` **8 / 12** · **Estimated time:** 45–60 min · **Difficulty:** Intermediate

8.1. Learning objectives

By the end of this chapter, you will be able to:

- Explain why brute-force cosine search stops being adequate as a corpus grows, in concrete terms (memory and latency).
- Build and query a FAISS index over DDR chunk embeddings.
- Understand the difference between a per-document index and a campaign-wide global index, and when each is the right tool.

8.2. Operational Problem

Mike, the completions engineer, has been happily querying all ten sample reports through Chapter 5's system — but what happens when his ten-document prototype meets the full archive? Chapter 4's `embeddings @ query_vec` brute-force search works fine over ten documents — it's a single matrix multiply. The full Utah FORGE archive is 76 reports. The companion pipeline's chunking and embedding steps, run against all 76, produce **1,428 chunks** — still small by industrial standards, but already the point where you want a persistent, queryable index rather than an in-memory NumPy array you rebuild every session. And a production system needs more than one query mode: per-document search (fast, scoped to one report) and campaign-wide search across every chunk are genuinely different operations.

8.3. Theory

A **vector database** (or vector index library, like FAISS (Johnson et al. 2017)) organises embeddings so that similarity search doesn't require comparing the query against every single vector.

> Engineering Translation: FAISS index
>
> Think of Chapter 4's brute-force search as walking every aisle of a warehouse to find a part. A **FAISS index** is that same warehouse with an addressing scheme: it can go straight to the right shelf instead of checking every one. At this book's scale the "shelf" it finds is still mathematically the exact same answer — it just gets there faster and without needing everything loaded into memory by hand each time.

At this book's scale, even the simplest FAISS index — `IndexFlatIP` (flat, inner-product) — is the right choice: it's an *exact* nearest-neighbour search, mathematically equivalent to Chapter 4's brute-force approach, just implemented in optimised C++ with a persistent, loadable index file. (Larger-scale approximate indexes — IVF, HNSW — trade a small amount of accuracy for speed at millions of vectors; not a tradeoff this book's scale needs.)

Because embeddings are normalized (Chapter 4), inner product *is* cosine similarity — which is exactly why `IndexFlatIP` is the right index type rather than a distance-based one.

> Engineering Translation: Persisting an index
>
> **Persisting** an index means saving it to a file, the same way you'd file a completed report in a binder instead of reconstructing it from memory every morning. Build it once, save it, and every future session just loads the finished file — no re-embedding, no rebuilding.

8.4. Building the real index

The companion pipeline, **DDR_UTAH_FORGE**, builds two tiers of index:

- **Per-document index** (`scripts/build_index.py`) — one FAISS index per DDR, used when a question is scoped to a specific report. All 76 reports already have this: chunks, embeddings, and a `faiss.index` file sit in `data/processed/<doc_id>/` for every report in the archive.
- **Global index** (`scripts/build_global_index.py`) — a single FAISS index over every chunk across every report, used for cross-document questions like "which reports involve fishing operations."

Running `build_global_index.py` against this archive's 76 pre-processed documents produces a real, verifiable result:

```
Found 76 docs with embeddings
Concatenating 1,428 chunks from 76 docs...
Building FAISS IndexFlatIP ((1428, 384))...
  faiss.index - 1,428 vectors × 384d
```

384 dimensions matches `all-MiniLM-L6-v2`, the same embedding model from Chapter 4 — this pipeline doesn't switch models between the toy version and the production one, it just changes how the resulting vectors are indexed and queried.

The two tiers branch from the same per-document work, then serve different questions:

```
76 DDR PDFs
   ↓
chunks + embeddings
(per report)
   ↓
built into two indexes:
   ↓
Per-doc FAISS index
(one per report)
-> search THIS report
   ↓
Global FAISS index
(1,428 chunks, all reports)
-> search ACROSS reports
```

8.5. Implementation

8.5.1. Step 1: build the index and save it to disk

What problem are we solving?

Turn a matrix of chunk embeddings into a fast, queryable structure, then save it so it never needs rebuilding from scratch in a future session.

Inputs

- `embeddings`: a NumPy array of chunk vectors, shape `(n_chunks, 384)`.
- A destination file path for the saved index.

Expected Output

A `faiss.index` file on disk, plus the same in-memory index object ready to query immediately.

```python
# code/chapter_08/build_faiss_index.py
from pathlib import Path

import faiss
import numpy as np

def build_index(embeddings: np.ndarray) -> faiss.IndexFlatIP:
    dimension = embeddings.shape[1]
    index = faiss.IndexFlatIP(dimension)
```

```
    index.add(embeddings.astype("float32"))
    return index

def save_index(index: faiss.Index, path: Path) -> None:
    faiss.write_index(index, str(path))
```

What just happened?

`build_index` creates an empty FAISS structure sized for 384-number vectors, then loads every chunk's embedding into it in one call. `save_index` writes that structure out to a single file — the "binder" from the Engineering Translation above — so the next session can load it back instantly instead of re-embedding every chunk again.

8.5.2. Step 2: reload the index and answer a query

What problem are we solving?

In a fresh session — no re-embedding, no rebuilding — load the saved index back and answer a query against it, the same way Chapter 4's `search` did, just backed by FAISS instead of a NumPy array in memory.

Inputs

- The saved `faiss.index` file path.
- A query vector (from Chapter 4's embedding model).
- `top_k`: how many results to return.

Expected Output

A ranked list of `(chunk_index, similarity_score)` pairs — the same shape of result Chapter 4's brute-force `search` returned.

```
def load_index(path: Path) -> faiss.Index:
    return faiss.read_index(str(path))

def search(index: faiss.Index, query_vec: np.ndarray, top_k: int = 5):
    scores, indices = index.search(query_vec.reshape(1, -1).astype("float32"),
↪   top_k)
    return list(zip(indices[0].tolist(), scores[0].tolist()))
```

What just happened?

`load_index` reads the saved file back into a ready-to-query FAISS index — no re-embedding step, because the vectors were already saved inside it. `search` hands it one query vector and asks for the `top_k` closest matches; FAISS returns their positions and scores, which get paired up into the same kind of ranked list Chapter 4 produced by hand.

This mirrors the indexing pattern in the companion pipeline's `src/rag_pdf/services/search_service.py`, which builds `IndexFlatIP` over chunk embeddings and persists it to `faiss.index` alongside each document's other artefacts.

8.5.3. Step 3: index real chunks, with real metadata attached

What problem are we solving?

Everything above indexes whole documents — Chapter 4's `load_chunks()` loads one vector per *report*, not per chunk, so a FAISS row can only ever answer "which report," never "which page of it," and never "which date." Chapter 7 already built a chunker that knows each chunk's page number; this step puts that chunker to work and adds one more thing while it's there — every DDR filename already carries its own report date (`..._2020-11-26.txt`), so reading it costs nothing extra and closes a second gap at the same time. One FAISS row per chunk, with a matching metadata record — report filename, page, and date — so a search result can finally point at more than just a whole file.

Inputs

- A folder of extracted `.txt` files: `datasets/ddr_text/`.
- Chapter 7's `chunk_pages_by_tokens()`.

Expected Output

A FAISS index with one row per chunk, and a parallel `metadata` list — `metadata[i]` is a `{"report": ..., "page": ..., "date": ...}` dict describing exactly what `embeddings[i]` is.

```
import re

REPORT_DATE = re.compile(r"_(\d{4}-\d{2}-\d{2})\.")

def report_date(filename: str) -> str | None:
    match = REPORT_DATE.search(filename)
    return match.group(1) if match else None

def build_chunk_metadata_index(text_dir: Path, model: SentenceTransformer,
                               chunk_tokens: int = 60, overlap_tokens: int =
 ↪  15):
    metadata, chunk_texts = [], []
    for path in sorted(text_dir.glob("*.txt")):
        pages_text = path.read_text(encoding="utf-8")
        date = report_date(path.name)
        for page_number, chunk in chunk_pages_by_tokens(pages_text, chunk_tokens,
         ↪  overlap_tokens):
            metadata.append({"report": path.name, "page": page_number, "date":
 ↪  date})
            chunk_texts.append(chunk)

    embeddings = embed_texts(model, chunk_texts)
    index = build_index(embeddings)
    return index, metadata
```

What just happened?

For every report, this runs Chapter 7's page-aware chunker, embeds each resulting chunk, and appends a matching `{"report", "page", "date"}` record to `metadata` in the same order the chunk's vector gets added to the index. Row `i` of the FAISS index and entry `i` of `metadata` describe the same chunk — that parallel-list discipline is the entire mechanism; there's no cleverness beyond keeping the two lists in lockstep. `report_date()` is a one-line regex against the filename, not a new data source: every filename in this archive already carries its report's date (confirmed against all 76 real filenames in the full archive — zero exceptions to the `_YYYY-MM-DD.` pattern), so extracting it is free once you're already reading the filename for `report`.

Run this against the real ten-report sample and it builds **333 chunks across 10 reports** — roughly ten times the rows the whole-document index above has, all from the exact same source text. Query it for `"stuck pipe"` and the difference isn't just more rows: report #38 — the actual stuck-pipe day — jumps to **rank 1** at a score of **0.4362**, clearly separated from its nearest competitor. The whole-document index earlier in this chapter ranked the same report **2nd**, at **0.1978**, behind a report that only mentions the topic in passing. Chunking doesn't just enable page-level citations — it measurably sharpens the ranking itself, for the same reason Chapter 4's Field Notes already showed: a whole report blends a few relevant sentences with pages of unrelated tables, and a smaller chunk can't hide relevant text inside irrelevant text the way a whole document can.

8.6. Production Reality

This chapter's `IndexFlatIP` scales cleanly to Utah FORGE's 1,428 chunks — both to build and to query. A single operator's *entire* well portfolio, or a service company handling multiple clients, changes the picture:

- millions of chunks make exact search too slow; that's when approximate indexes (IVF, HNSW) become the right tradeoff, not just an option mentioned in passing
- a flat index file rebuilt from scratch every time a new DDR arrives doesn't scale — production systems need to add new vectors incrementally, without reprocessing the whole archive
- multiple engineers querying the same index at once, from different applications, usually means a managed vector database (pgvector, Pinecone, Weaviate) instead of a single `faiss.index` file on one machine's disk
- if some wells' data is more sensitive than others (a joint-venture well, say, versus a wholly-owned one), a single global index needs an access-control layer on top of it — FAISS itself has no concept of "who's allowed to see this chunk"
- this chapter's own code deliberately restricts FAISS and the embedding model to a single CPU thread (`OMP_NUM_THREADS=1`). Not a hypothetical caution — a fix for a real, reproducible crash confirmed during this book's own testing: on macOS, FAISS and PyTorch (which the embedding model runs on) each bring their own separate copy of the same multi-threading library, and letting both use more than one thread at once corrupts memory and crashes the whole program. Invisible at ten documents. A genuinely large archive would want real multi-threaded performance back, which means revisiting that limit deliberately — checking whether your specific combination of library versions still hits the same conflict — rather than leaving every embedding job capped at one thread by default forever

8.7. Practical exercise

Beginner

Try it yourself: Build a FAISS index over the ten sample DDR embeddings from Chapter 4, save it to disk, reload it in a fresh Python session, and confirm a query for "stuck pipe" returns the same ranked order as Chapter 4's brute-force search — report #38 at rank 2, report #39 at rank 5.

You'll know it worked when: the FAISS-based and brute-force results agree — because, mathematically, `IndexFlatIP` over normalized vectors *is* cosine similarity, just implemented differently.

8.8. Field notes

⚠ Field notes: proving 'exact, not approximate' rather than asserting it

Action: run five different real queries — `"stuck pipe"`, `"fishing operation"`, `"packers failed to set"`, `"torque problems"`, `"losses"` — through both the brute-force NumPy search and a FAISS `IndexFlatIP` built over the same ten embeddings, and compare the full ranked order each one returns.
Result: all five queries produce byte-for-byte identical rankings between the two methods.

```
'stuck pipe':              brute-force order == FAISS order: True
'fishing operation':       brute-force order == FAISS order: True
'packers failed to set':   brute-force order == FAISS order: True
'torque problems':         brute-force order == FAISS order: True
'losses':                  brute-force order == FAISS order: True
```

Why: this is exactly what the Theory section claimed — `IndexFlatIP` computes the *same* inner-product comparison as `embeddings @ query_vec`, just in optimised C++ rather than NumPy. There's no approximation step anywhere in this index type for it to disagree with brute force on.
Lesson: "mathematically equivalent" is a claim worth testing, not just trusting because it sounds right — especially before you rely on it at a scale where you can no longer eyeball the difference. It took five lines of code to confirm this one.

8.9. Challenge exercise

Advanced

Challenge: If you have the full 76-report Utah FORGE archive (`datasets/forge_archive/`) and can run the heavier dependencies (`sentence-transformers`, `faiss-cpu`), chunk and embed all 76 reports yourself with the production pipeline's own parameters — 224 tokens, 56-token overlap — and compare against the real 1,428-chunk count above. Expect a real gap, not just rounding noise: this book's chunker uses `pdfplumber`'s plain text extraction, which pulls noticeably less text out of each DDR's data tables than the production pipeline's table-aware extraction does, so it produces

fewer chunks even with identical chunking parameters. A reference solution is in `code/chapter_08/challenge/`.

8.10. Key takeaways

- `IndexFlatIP` at this book's scale is exact, not approximate — you're not trading away accuracy, only reimplementing brute-force search more efficiently and persistently.
- Per-document and global indexes serve genuinely different questions; build both rather than forcing one index to do both jobs.
- Indexing real chunks instead of whole documents isn't just what makes page-level citations possible — it measurably improved this chapter's own "stuck pipe" ranking, from 2nd place to 1st.
- Scaling the *index* doesn't fix retrieval *quality* on its own — that's the next chapter's problem, though chunk-level indexing already closes part of the gap.

8.11. Repository files

File	Purpose
`code/chapter_08/build_faiss_index.py`	FAISS `IndexFlatIP` build/save/load/search, plus chunk+metadata indexing
`code/chapter_08/build_full_archive.py`	Reproduces the full 76-report archive from the public source
`datasets/forge_archive/`	The full 76-report Utah FORGE archive
`DDR_UTAH_FORGE/scripts/build_index.py`	Per-document index builder (companion repo)
`DDR_UTAH_FORGE/scripts/build_global_index.py`	Campaign-wide global index builder (companion repo)

CHECKPOINT — Chapter 8

- ☒ Explained why brute-force search stops scaling, in concrete memory/latency terms
- ☒ Built and queried a FAISS index over DDR chunk embeddings
- ☒ Persisted an index to disk and reloaded it in a fresh session
- ☒ Verified FAISS results match brute-force results exactly, not approximately
- ☒ Built a real chunk-level index with (report, page) metadata attached to every row

WHAT YOU BUILT

`build_faiss_index.py` — a persistent, queryable FAISS vector index: build it once, save it to disk, and every future session loads it back instantly instead of re-embedding the archive. Its chunk-level variant attaches real (report, page) metadata to every row, ready for Chapter 10's citations.

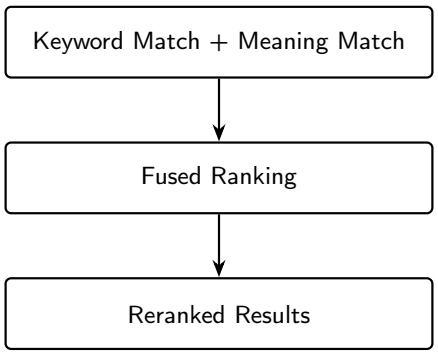
8.12. What can you do now that you couldn't do before?

You can build a persistent, queryable index over an entire report archive that survives a restart — and you've verified, not just assumed, that scaling up the infrastructure didn't change a single answer the retrieval gives back. Every row in the chunk-level index also carries a real page number, ready for Chapter 10 to actually cite.

8.13. Suggested next step

Coming up in Chapter 9: A fast index doesn't guarantee good retrieval. Chapter 9 looks at the retrieval scoring code itself — how sparse and dense signals get combined, and what it takes to make that combination numerically sound.

9. Hybrid Retrieval and Reranking



Progress **#########... 9 / 12** · **Estimated time:** 60–75 min · **Difficulty:** Advanced

9.1. Learning objectives

By the end of this chapter, you will be able to:

- Explain why semantic search alone underperforms on precise, entity- or number-heavy queries.
- Combine BM25 (sparse/exact) and dense (semantic) retrieval using Reciprocal Rank Fusion (RRF).
- Explain when and why production systems add cross-encoder reranking on top of fused retrieval, and the latency cost it trades for precision.
- Read production retrieval-scoring code closely enough to explain *why* a specific formula is written the way it is, not just what it computes.

9.2. Operational Problem

Mike, the completions engineer, is back with the packers question from Chapter 8: which report had packers fail to set — and why doesn't either search method alone find it reliably? A query like *"which report had packers fail to set?"* has both a semantic component (packers, setting, failure — a topic) and effectively an exact-match component (you want report #49 specifically, not every report that mentions packers in passing). Dense embeddings capture the topical meaning well but blur precise distinctions; keyword search (Chapter 3) nails the exact terms but misses paraphrase entirely. Neither alone is enough — production retrieval systems combine both.

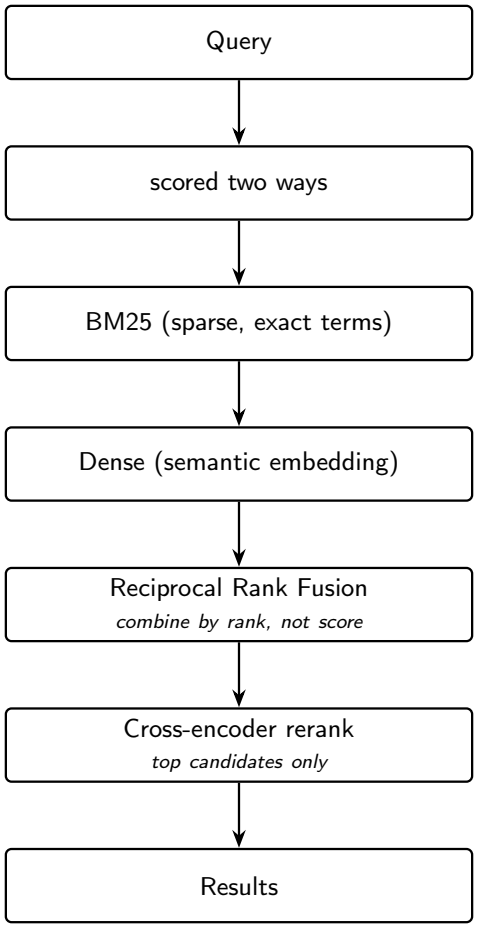
9.3. Theory

BM25 (Robertson and Zaragoza 2009) scores a document by term frequency, adjusted by how rare each term is across the corpus (inverse document frequency, IDF) and normalised for document length. It's decades old, computationally cheap, and excellent at exact and near-exact matches — the opposite profile from dense embeddings.

> Engineering Translation: BM25
>
> **BM25** is keyword search's more careful cousin. It still matches exact words, like Chapter 3's search, but gives more credit for rare, specific terms ("packers") than common ones ("the"), and adjusts for how long each report is so a longer report doesn't win just by containing more words overall.

Reciprocal Rank Fusion (RRF) combines two ranked lists by *rank*, not raw score: for each document, sum `weight / (k + rank)` across both lists. This sidesteps a real problem — BM25 scores and cosine similarities live on completely different, incomparable scales, so averaging raw scores directly would let whichever signal happens to have larger numbers dominate regardless of which is actually more informative.

> Engineering Translation: Reciprocal Rank Fusion
>
> Think of two judges ranking the same horse race. One scores out of 10, the other out of 100 — their raw scores can't be averaged meaningfully. But their *placings* (1st, 2nd, 3rd…) mean the same thing regardless of scale. **RRF** combines two search results the same way: by where each report placed in each ranking, not by the raw number each method happened to produce.



Cross-encoder reranking is the last, most expensive step, and deliberately runs on only the top handful of fused candidates, not the whole corpus. Unlike a dense embedding (which encodes the query and each document *separately*, then compares vectors), a cross-encoder scores the query and a candidate passage *together*, letting it capture interactions between them that no fixed vector representation can. This makes it substantially slower per comparison — which is exactly why it's applied after fusion has already narrowed the field, not instead of it.

> Engineering Translation: Cross-encoder
>
> A dense embedding is like summarizing the query and each report separately, then comparing the two summaries from a distance. A **cross-encoder** instead reads the query and one candidate passage *side by side, together*, and judges the pair directly — closer to how a person actually double-checks a match, but too slow to do for every report, which is why it only runs on the small shortlist fusion already produced.

9.4. Reading the real fusion code

The companion pipeline's `src/rag_pdf/retrieval/hybrid_utils.py` and `src/rag_pdf/services/search_service.py` implement exactly this pipeline. Two details in the real code are worth

understanding closely, because they're the kind of thing that looks like a stylistic choice until you see what breaks without it.

Detail 1: the IDF formula's shape isn't arbitrary

In plain terms: this line makes sure a very common word can never make a report's score go *negative*. Here's the code:

```
self.idf[term] = math.log(1.0 + ((self.n_docs - df + 0.5) / (df + 0.5)))
```

Compare this to the classic Robertson-Sparse-Selection IDF, `log((N - df + 0.5) / (df + 0.5))` *without* the `1.0 +`. For a term that appears in almost every document (`df` close to `n_docs`), the classic formula's ratio drops below 1 and the log goes **negative** — which then corrupts any downstream sum that assumes term contributions are non-negative. Wrapping the ratio as `1.0 + ratio` before taking the log guarantees the result stays non-negative for every possible `df`, because `1.0 + ratio` can never fall below 1. This single design choice removes an entire category of edge case, permanently, for a cost of one added constant.

Detail 2: normalization has to define its own tie-break

In plain terms: this guards against a division that would otherwise break the whole calculation whenever every candidate ties. Here's the code:

```
if hi <= lo:
    return {i: 0.0 for i in candidates}
```

Min-max normalization rescales scores into `[0, 1]` by subtracting the minimum and dividing by the range. If every candidate in a batch scores identically, `hi - lo` is zero — a division that, left unguarded, would raise or produce `NaN`. The guard above catches that case explicitly and returns a neutral value (0.0 for everyone) instead, so a tie in *this* signal doesn't crash the pipeline or corrupt the fused score — the other signal in the fusion is still free to discriminate between the tied candidates.

Neither of these edge cases shows up by running "more queries" against typical text — they only show up by reasoning about the math at its boundaries: what happens when a term is universal, what happens when every score ties. Retrieval scoring is numerical code, and it deserves the same edge-case discipline you'd apply to any other numerical code. The fusion weights themselves are also plain, readable constants in `search_service.py`: `RRF_K = 20`, `RRF_DENSE_WEIGHT = 0.5`, `RRF_BM25_WEIGHT = 2.0` — the exact-match signal is weighted four times higher than the semantic one in this pipeline's fused ranking, reflecting that for DDR text, exact-term matching often carries more precision than pure similarity.

9.5. Implementation

What problem are we solving?

Combine two independently-ranked lists of reports — one from BM25, one from dense/semantic search — into a single fused ranking, without letting whichever method happens to produce larger raw numbers unfairly dominate the result.

Inputs

- `ranked_lists`: two or more ranked lists of document IDs, best match first (e.g. one from Chapter 3's keyword search, one from Chapter 4's semantic search).
- `k`: a constant controlling how much rank position matters.
- `weights`: how much to trust each list relative to the others.

Expected Output

A single fused ranking: a list of `(doc_id, score)` pairs, sorted highest-scoring first.

```
# code/chapter_09/hybrid_search.py
from collections import defaultdict

def reciprocal_rank_fusion(ranked_lists: list[list[str]], k: int = 20,
                           weights: list[float] | None = None) ->
 ↪  list[tuple[str, float]]:
    weights = weights or [1.0] * len(ranked_lists)
    scores: dict[str, float] = defaultdict(float)
    for ranked_list, weight in zip(ranked_lists, weights):
        for rank, doc_id in enumerate(ranked_list, start=1):
            scores[doc_id] += weight * (1.0 / (k + rank))
    return sorted(scores.items(), key=lambda item: item[1], reverse=True)
```

What just happened?

For every report, this adds up a small credit from each ranked list based on *where it placed* — a high rank (near the top) contributes a bigger credit than a low rank, scaled by that list's weight — then sorts every report by its total credit. Because this uses each list's rank, not its own raw numbers, a method that happens to score everything on a bigger scale can't unfairly take over the fused result. (The `key=lambda item: item[1]` on the sort line is just Python's way of saying "order by the second thing in each pair" — here, the fused score rather than the report name.)

9.5.1. Turning that into a real hybrid search

`reciprocal_rank_fusion` fuses ranked lists — but it needs two of them to fuse. Chapter 4's semantic `search()` already returns one. The missing piece is a ranked *sparse* signal: Chapter 3's `search()` returns an unordered set — a report either contains every query word or it doesn't — which RRF has nothing to order. `code/chapter_09/sparse_ranking.py` closes that gap with `rank_bm25()`, a BM25 ranking over the same archive (with a plain term-frequency fallback if `rank-bm25` isn't installed):

```python
# code/chapter_09/sparse_ranking.py
from rank_bm25 import BM25Okapi

def rank_bm25(text_dir, query):
    paths = sorted(text_dir.glob("*.txt"))
    corpus = [tokenize(p.read_text(encoding="utf-8")) for p in paths]
    scores = BM25Okapi(corpus).get_scores(tokenize(query))
    ranked = sorted(zip(paths, scores), key=lambda ps: ps[1], reverse=True)
    return [p.name for p, _ in ranked]
```

With both ranked signals in hand, `hybrid_search()` in `code/chapter_09/hybrid_search.py` runs all three steps end to end: BM25 sparse ranking, Chapter 4's dense ranking, fused with the weights above. That's the function the practical exercise below calls.

One consistency note, tying back to Chapter 4: BM25 is a lexical method, so by that chapter's Field Note it belongs on the *expanded* text, while the dense signal uses the raw. This exercise runs both over the same raw sample archive for simplicity, and for its queries that's harmless — the report you're checking for tops the BM25 ranking either way (report #49 for `"packers fishing"`, report #38 for `"stuck pipe"`). Where it *would* matter is an abbreviation query like `"bottom hole assembly"`: on expanded text BM25 surfaces report #38, which only ever writes `BHA`; on raw text it misses it entirely. A production pipeline points the sparse signal at the expanded text for exactly that reason.

9.6. Production Reality

This chapter tunes two signals with fixed weights (BM25 × 2.0, dense × 0.5) chosen for DDR text specifically. Real systems have to keep that tuning honest over time, not treat it as a one-time decision:

- these weights were tuned for this domain's language — a different report type (well completions versus drilling, say, or a different operator's writing style) may need genuinely different weights, not the same ones reused out of habit
- cross-encoder reranking adds real per-query latency, the same way an LLM call does (Chapter 5) — it has to be budgeted for, not just switched on and forgotten
- production systems often fuse more than two signals — metadata filters like well name, date range, or report type frequently join BM25 and dense scores in the same fusion step, and each added signal is one more way the weighting can go wrong
- as this chapter's own Field Notes show below, adding a signal isn't automatically an improvement — a production system needs ongoing evaluation (Chapter 11), not a one-time tuning pass assumed to hold forever

9.7. Practical exercise

Beginner

Try it yourself: Rank the ten sample DDRs for `"packers fishing"` two ways — `rank_bm25()` for the sparse signal, Chapter 4's `search()` for the dense one — then fuse them with `reciprocal_rank_fusion([sparse, dense], k=20, weights=[2.0, 0.5])`, matching the companion pipeline's real weighting. `hybrid_search()` does all three steps in one call if you'd rather run it end to end.

You'll know it worked when: report #49 (packers) and #50 (fishing) both rank in the top two of the fused list, and you can explain why the weighting favours the keyword signal here.

9.8. Field notes

⚠ Field notes: hybrid fusion isn't automatically better — check both directions

Query: the same `"stuck pipe"` query from Chapters 3 and 4.
Result: with the pipeline's real fusion weights (BM25 × 2.0, dense × 0.5), report #39 ranks **9th of 10** — *worse* than Chapter 4's dense-only search, which ranked it 5th. Equal weighting (1.0 / 1.0) still only gets it to 7th.

```
BM25 alone:    report #39 ranks 9th of 10
Dense alone:   report #39 ranks 5th of 10   (Chapter 4)
RRF (2.0/0.5): report #39 ranks 9th of 10   (pipeline default weights)
RRF (1.0/1.0): report #39 ranks 7th of 10
```

Compare that to `"packers fishing"` (the practical exercise above), where fusion helps exactly as expected — report #49 stays 1st and report #50 jumps from 8th (dense alone) to 2nd (fused).
You can reproduce every rank above with this book's own code: `rank_bm25()` and `hybrid_search()` (from the implementation section) give report #39 exactly these positions over the sample archive — 9th on BM25 alone, 9th fused at 2.0/0.5, 7th at 1.0/1.0. The companion pipeline's fuller BM25 implementation (`DDR_UTAH_FORGE/src/rag_pdf/retrieval/hybrid_utils.py`) lands on the same ordering for this query, which is a good sign the simplified version here captures what actually matters.
Why: BM25 has essentially nothing useful to say about report #39 for `"stuck pipe"` — it shares almost no vocabulary with the query, so BM25 ranks it 9th on its own. Fusing that weak, near-random signal in with a weight of `2.0` doesn't cancel out — it actively drags a mediocre dense result down further. For `"packers fishing"`, by contrast, both signals carry real information (the word "fishing" genuinely appears in report #50), so combining them helps.
Lesson: hybrid retrieval's benefit depends on *both* signals being informative for the query at hand — not on combining more signals being unconditionally better. A sparse signal with nothing to contribute isn't neutral when fused; it's noise with a weight attached, and can make a weak dense ranking worse. This is exactly why Chapter 11 insists on per-category evaluation rather than one aggregate score — "hybrid beats dense on average" can still be actively wrong for a specific, operationally important query like this one.

9.9. Challenge exercise

Advanced

Challenge: Implement the two guards from the code study above against deliberately constructed edge cases: a term that appears in every sample document (test that your IDF stays non-negative), and a batch where every candidate scores identically (test that your normalization doesn't divide by zero). Confirm an unguarded version fails first. A reference solution is in `code/chapter_09/challenge/`.

9.10. Key takeaways

- Sparse and dense retrieval fail on different, complementary query types — combine them rather than picking one.
- Rank-based fusion (RRF) avoids the scale-mismatch problem that plagues naive score-averaging across different retrieval methods.
- Cross-encoder reranking buys precision at a real latency cost — apply it only to a small, already-fused candidate set.
- Retrieval scoring is numerical code. Boundary conditions — universal terms, tied scores — deserve explicit guards, and reading *why* a formula is shaped the way it is tells you more than reading what it computes.

9.11. Repository files

File	Purpose
`code/chapter_09/sparse_ranking.py`	Ranked BM25 sparse retriever — the ranked signal RRF fuses
`code/chapter_09/hybrid_search.py`	Reciprocal Rank Fusion, plus `hybrid_search()` end-to-end
`DDR_UTAH_FORGE/src/rag_pdf/retrieval/hybrid_utils.py`	BM25 index, IDF formula, score fusion (companion repo)
`DDR_UTAH_FORGE/src/rag_pdf/retrieval/canonical_hybrid.py`	Full hybrid retrieval pipeline (companion repo)
`DDR_UTAH_FORGE/src/rag_pdf/services/search_service.py`	Cross-encoder rerank + fusion weights (companion repo)

CHECKPOINT — Chapter 9

- ☒ Explained why semantic search alone underperforms on precise, entity-heavy queries
- ☒ Combined BM25 and dense retrieval using Reciprocal Rank Fusion
- ☒ Read production scoring code closely enough to explain why its guards exist
- ☒ Confirmed fusion helps on one query and actively hurts on another, using real evidence

WHAT YOU BUILT

`hybrid_search.py` — a Reciprocal Rank Fusion engine combining keyword and semantic retrieval into one ranked list that plays to both methods' strengths.

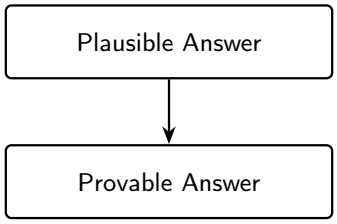
9.12. What can you do now that you couldn't do before?

You can combine exact keyword matching and semantic search into a single ranked list that plays to each method's strengths — and, from real evidence rather than assumption, you know that combining signals only helps when both of them actually have something useful to say about the query at hand.

9.13. Suggested next step

Coming up in Chapter 10: Better retrieval still isn't the same as a trustworthy answer. Chapter 10 tackles the harder problem: making sure every generated claim traces back to real evidence, and that the system tells you honestly when it doesn't know.

10. Traceable Answers and Hallucination Mitigation



Progress **##########.. 10 / 12** · **Estimated time:** 60–75 min · **Difficulty:** Advanced

10.1. Learning objectives

By the end of this chapter, you will be able to:

- Distinguish questions that should be answered by generation from questions that should be answered by structured data lookup.
- Attach a verifiable citation (report number, page, and date) to every claim a system produces.
- Detect and surface gaps in your own source archive, so the system's silence about a period isn't mistaken for "nothing happened."

10.2. Operational Problem

Sean, the production engineer, asks a pointed question after seeing a generated number in a draft report: *"Can you actually prove that number, or did the model just make it sound right?"* Chapter 5 built a prompt that *asks* a language model to cite sources — necessary, but not sufficient. A model can still cite a plausible-sounding report number for a number it invented, especially for arithmetic questions ("how many hours of non-productive time in November?") that language models are statistically bad at answering exactly, no matter how good the prompt is. Separately, this archive — like any real archive — has a genuine gap, and a system that answers confidently from what it *has*, without flagging what it's *missing*, can quietly mislead an engineer who assumes silence means nothing happened.

10.3. Structured facts, already extracted

Language models can produce fluent, plausible claims that are simply false — a well-documented failure mode called hallucination (Ji et al. 2023).

> Engineering Translation: Hallucination
>
> A **hallucination** is a confident wrong answer — the language-model equivalent of a new hire who guesses a number rather than admitting "I don't know," and says it fluently enough that nobody thinks to double-check. It isn't lying on purpose; it's producing text that sounds right without anything underneath actually verifying it.

The single most effective mitigation in this book is a rule of thumb, not a model trick: **if a question can be answered by looking something up or computing it from structured data, don't generate the answer — compute it.**

Every report in this archive has already been parsed into a real, structured `ddr_facts.parquet` table by the companion pipeline — a file that lives in `DDR_UTAH_FORGE`, not in this book's repository, so the row below is a reported fact, not something you can query yourself without that pipeline. Here's an actual row from report #38 (2020-11-26):

> A real row from ddr_facts.parquet, report #38

```
report_date: 2020-11-26        wellbore: FORGE-16A-78-32
start_time:  06:00              end_time:      11:00
duration_hr: 5.0                 phase:         Production Drilling
op_code:     Wire Line Logs      is_npt:        False
operation_text: "PJSM, pre job safety meeting. Rig up Schlumberger
                 run the UBI log from 5.200 to surface. Rig down
                 logging tools."
```

A question like *"how many hours did report #38 spend on wireline logging?"* has an exact answer sitting in this table — `duration_hr = 5.0` — computed once during extraction, not re-derived by a language model each time it's asked. Generation is reserved for what it's actually good at — narrative synthesis across multiple passages — and even then, every claim carries its source.

```
question
   ↓
answerable from structured data?
   ↓
yes -> query ddr_facts.parquet
    -> exact, re-derivable answer
   ↓
no  -> retrieve + generate (Ch.5)
    -> answer + citations
       (verify before trusting)
```

10.4. A real, honest gap in this archive

The second mitigation is **corpus-completeness awareness**. This archive has a genuine one: the last Drilling report is #76, dated **2021-01-03**. The Completion report (a DFIT — Diagnostic Fracture Injection Test) is dated **2021-01-06**. Two calendar days — January 4th and 5th — have no report of any kind. Every other day in the drilling programme, from spud on October 30, 2020 through report #76, has exactly one report, so this gap is real and specific to the Drilling-to-Completion handover, not an artefact of messy source data.

A RAG system only knows what's in its index. If you ask it "what happened on January 5th, 2021?" without gap-awareness, a poorly built system might either hallucinate an answer or simply retrieve the nearest chunks (report #76 or the Completion report) without ever telling you neither actually covers that date. Detecting the gap is a matter of checking the *sequence* of report numbers and dates for holes — not the content, just the presence.

10.5. Implementation

10.5.1. Step 1: give every claim a checkable tag

What problem are we solving?

Attach a checkable source to every claim — which report, which page — instead of a citation-shaped sentence a reader has to take on faith.

Inputs

- A report name/number, and an optional page number.
- A list of such citations to attach to one answer.

Expected Output

A plain-text "Evidence:" block listing each source, ready to show alongside a generated answer.

```
# code/chapter_10/traceable_answers.py
from dataclasses import dataclass

@dataclass
class Citation:
    report: str
    page: int | None = None

def format_evidence(citations: list[Citation]) -> str:
    lines = [f"  {c.report}" + (f" page {c.page}" if c.page else "") for c in
 ↪  citations]
    return "Evidence:\n" + "\n".join(lines)
```

What just happened?

`Citation` bundles a report and a page number together as one unit — the equivalent of an evidence tag on an exhibit, so a claim's source can never get separated from the claim itself by accident. `format_evidence` turns a list of those tags into a plain block of text a reader can check against the real reports, one line per source.

> Engineering Translation: Dataclass
>
> A **dataclass** like `Citation` is a small, labelled bundle of related facts — the code equivalent of an equipment tag that keeps a part number and its location together on one card, instead of as two separate loose numbers you could mix up.

10.5.2. Step 2: check the archive for real gaps

What problem are we solving?

Detect whether the archive has days with no report at all, so the system can say "I have nothing for that date" instead of guessing or silently retrieving the nearest available report.

Inputs

- A list of report dates, as ISO date strings, covering a period that should have one report per day.

Expected Output

A list of `(start, end)` date ranges with no report — empty if the archive is complete.

```
def find_date_gaps(report_dates: list[str]) -> list[tuple[str, str]]:
    """Return (start, end) date ranges with no report, given a sorted
    list of ISO date strings covering a continuous daily-reporting period."""
    from datetime import date, timedelta

    dates = sorted(date.fromisoformat(d) for d in report_dates)
    gaps = []
    for prev, curr in zip(dates, dates[1:]):
        missing_days = (curr - prev).days - 1
        if missing_days > 0:
            gaps.append(((prev + timedelta(days=1)).isoformat(), (curr -
↪   timedelta(days=1)).isoformat()))
    return gaps
```

What just happened?

The dates get sorted, then checked one consecutive pair at a time: if more than one day passes between two reports, everything in between gets recorded as a missing range. No content is inspected here at all — this only checks whether a report *exists* for each day, which is exactly enough to catch the real two-day gap in this archive.

10.5.3. Step 3: build a citation from a real search, not by hand

What problem are we solving?

Every `Citation` shown so far in this chapter was typed in by hand — `Citation(report="...", page=1)` — which proves the data structure works but not that the *system* can produce one. A citation is only actually trustworthy if it comes from the same retrieval the answer itself used. This step closes that loop: run a real query against Chapter 8's chunk-level index, and read the report, page, and date straight off whichever chunk matched — never re-typed, never assumed.

Inputs

- Chapter 8's `build_chunk_metadata_index()` and `search()`.
- A query string.

Expected Output

A list of `Citation` objects built entirely from the search results — real reports, real pages, real dates, in ranked order.

```
@dataclass
class Citation:
    report: str
    page: int | None = None
    report_date: str | None = None  # "report_date": date is the stdlib class

def citations_from_search(chunk_results: list[tuple[int, float]],
                          metadata: list[dict]) -> list[Citation]:
    citations = []
    seen_report_pages = set()
    for idx, _score in chunk_results:
        report_page = (metadata[idx]["report"], metadata[idx]["page"])
        if report_page in seen_report_pages:
            continue
        seen_report_pages.add(report_page)
        citations.append(Citation(report=metadata[idx]["report"],
↪   page=metadata[idx]["page"],
                                  report_date=metadata[idx].get("date")))
    return citations
```

What just happened?

For every `(row_index, score)` pair a search returns, this looks up `metadata[row_index]` — the real report, page, and date Chapter 8 recorded for that exact chunk — and wraps it in a `Citation`. There's no separate "figure out the source" step: the metadata was attached back when the chunk was indexed, so reading it is all citing requires. The field is named `report_date`, not `date` — `date` is already the name of the standard library class this file imports below for `find_date_gaps`, and shadowing it would be its own quiet bug waiting to happen.

The `seen_report_pages` set is doing real work, not defensive boilerplate. A long page gets split into several chunks by Chapter 7's chunker, and more than one of those chunks can land in the same top-k result — so without this check, one report page can show up in the Evidence list two, three, or four times, drowning out how many *distinct* sources actually back the answer. Only the first — and therefore best-ranked — chunk from each `(report, page)` pair is kept; later duplicates are skipped.

Run this against the real archive for `"stuck pipe"` at `top_k=3` and the top result is:

```
Evidence:
  FORGE-16A-78-32_Drilling_038_2020-11-26.txt page 1 (2020-11-26)
```

That's the same report #38 this whole book has used for the stuck-pipe story, cited automatically from a live search — not because the example was written to come out that way, but because Chapter 8 already verified report #38 ranks 1st for this exact query against the real chunk-level index. Every DDR in this sample archive is a single page, so `page` reads `1` every time here; the code doesn't know that in advance, and would report page 4 just as readily on a report that actually had one, which is the entire point of building it this way instead of typing `page=1` by hand. The date, unlike the page, isn't a Chapter 7 invention at all — it was always sitting in the filename; this step is the first place anything actually reads it.

Widen the same query to `top_k=10` and the gap the deduplication closes becomes visible: 10 chunk results come back, but only 6 distinct `(report, page)` pairs are behind them — the Completion report alone supplies four of the ten chunks. Without deduplication, the Evidence list would repeat that one report four times and imply four independent sources instead of one.

10.6. Production Reality

This chapter's citation and gap-detection code both assume the underlying extraction is trustworthy — but structured extraction is still code, written by someone, and can encode a judgment call worth checking (see this chapter's own Field Notes on `is_npt` below). A deployed system has more to plan for:

- gap detection has to run continuously as new reports arrive, not just once at setup — a system that only checked for gaps on day one won't notice a new one appearing three months later
- not every gap is administrative silence — a missing day could mean a missing safety report, which is worth escalating to a person, not just logging quietly as "no data for this date"
- a citation is only useful if someone actually reads it — a system that buries the evidence list at the bottom of a long answer, in small print, earns the same blind trust as a system with no citations at all
- some engineering decisions (regulatory submissions, incident reports) require a human sign-off no matter how good the system's citations are — traceability supports a human reviewer, it doesn't replace one

10.7. Practical exercise

Beginner

Try it yourself: Run `find_date_gaps()` against the report dates in this book's ten-report sample — `["2020-10-22", "2020-11-07", "2020-11-24", "2020-11-25", "2020-11-26", "2020-11-27", "2020-12-06", "2020-12-07", "2020-12-08", "2021-01-06"]`.

You'll know it worked when: the function reports several gaps — most of them simply reflecting that Part I's curated subset skips most of the archive's days on purpose. Then run it against the *full* 76-report archive's dates (`datasets/forge_archive/`) and confirm it finds exactly one real gap: January 4th–5th, 2021, right before the Completion report.

10.8. Field notes

⚠ Field notes: verify the automatic label, not just the automatic lookup

Action: check the real, computed `is_npt` flag — set once during extraction, not re-derived on every query — for every operations row on report #38, the stuck-pipe day.
Result: out of ten rows covering the full 24 hours, exactly **one** is flagged `is_npt = True`:

```
23:30-04:00  is_npt=True   "Drill From 6,360' to 6,507'... During the
                            slide lost tool face and became assembly
                            became stuck"
04:00-06:00  is_npt=False  "Work pipe, circulate lube sweep, work tool
                            back in position, Pipe free"
```

Why: the classifier flagged the 4.5-hour block where the pipe *became* stuck, but not the following 2-hour block where the crew actually recovered it. That's a defensible reading — the drilling operation is what got interrupted — but it's also a real judgment call: an engineer asking "how many hours did this stuck-pipe event cost?" would probably expect all 6.5 hours counted, not 4.5.
Lesson: a structured field being *automatically extracted* doesn't mean it's automatically *correct for your question.* Trusting `ddr_facts.parquet` blindly here would silently undercount the incident by 2 hours, every time someone asks. The fix isn't to distrust structured data — Chapter 10's whole argument is that it beats generation — it's to read the classification logic once, understand exactly what it counts, and know the edge it sits on before you build a report or a chart on top of it.
(`ddr_facts.parquet` and its `is_npt` classifier live in `DDR_UTAH_FORGE`, not in this book's repository — the rows above are reported from that pipeline, not something you can query with this book's own bundled code.)

10.9. Challenge exercise

Intermediate

Challenge: Extend `find_date_gaps()` to classify each gap's severity (e.g. a gap under 3 days is Low; a gap that crosses a report-type boundary — Drilling to Completion, as this one does — is High regardless of length, since it likely represents an unrecorded operational transition). A reference solution is in `code/chapter_10/challenge/`.

10.10. Key takeaways

- Prefer structured lookup over generation whenever the question is answerable from data you already extracted — it's exact, not just plausible.
- Every generated claim needs a citation an engineer can independently check, not just a citation-shaped sentence.
- A citation is only as trustworthy as where its fields come from — a hand-typed `page=1` proves the data structure works, not that the system can produce one. Chapters 7-10 wire the real value through: Chapter 1's page markers, Chapter 7's per-page chunking, Chapter 8's chunk metadata, read back here. Not every field needs that much plumbing, though — the report date was sitting in the filename the whole time; the work there was reading it, not producing it.
- A system that can tell you what it *doesn't have* is more trustworthy than one that answers fluently regardless of coverage — this archive's real two-day gap at the Drilling-to-Completion handover is exactly the kind of silence worth surfacing explicitly, not filling in.
- Deduplicate citations by `(report, page)`, not just by chunk — a chunk-level index can return several chunks from the same page among the top-k results, and an Evidence list that repeats one source four times looks like four independent sources unless it's collapsed down to the distinct ones.

10.11. Repository files

File	Purpose
`code/chapter_10/traceable_answers.py`	Citation formatting, real search-to-citation wiring, and date-gap detection
`DDR_UTAH_FORGE/data/processed/qc/raw_pdf_missing_reports.csv`	Real, computed gap-detection output (companion repo)

CHECKPOINT — Chapter 10

- ☒ Distinguished questions to compute from data versus questions to generate an answer to
- ☒ Attached a checkable citation to every generated claim
- ☒ Built a real Citation from a live search result, not typed in by hand
- ☒ Deduplicated citations so a repeated report/page isn't miscounted as independent evidence
- ☒ Detected a real gap in the archive instead of silently assuming completeness
- ☒ Caught a structured field's judgment call, not just trusted it blindly

WHAT YOU BUILT

`traceable_answers.py` — a citation and gap-detection layer: every claim carries a real, live-search-sourced citation — report, page, and date, never hand-typed — and every silence in the archive gets surfaced instead of hidden.

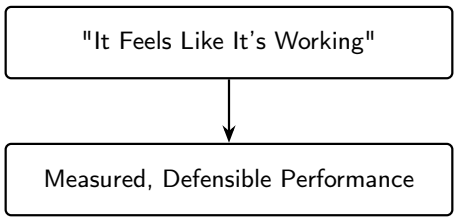
10.12. What can you do now that you couldn't do before?

You can tell the difference between a question you should compute from data you already have and one you should generate an answer to — and you can prove, with a citation built from a real search result (not typed in by hand), exactly where every part of a generated answer came from, including telling an engineer honestly when the archive simply has nothing to say.

10.13. Suggested next step

Coming up in Chapter 11: You now have a system that retrieves well and answers honestly. Chapter 11 asks the question every system like this eventually needs answered with numbers, not impressions: *how good is it, really* — and where specifically does it still fail?

11. Evaluating Retrieval Quality



Progress **##########. 11 / 12** · **Estimated time:** 60–90 min · **Difficulty:** Advanced

11.1. Learning objectives

By the end of this chapter, you will be able to:

- Build a hand-labeled evaluation set that records, per question, the expected source report and page.
- Compute `recall@k`, Mean Reciprocal Rank (MRR), and `NDCG@k`, and explain what each number actually tells you.
- Break results down by question category and difficulty to find *where* a system fails, not just *whether* it does.

11.2. Operational Problem

Oumy, the drilling engineer, has one blunt question before this system goes anywhere near her rig: "it feels like it's working" is not a standard you should ship an engineering decision-support tool against. You need a defensible answer to "how good is it" — and, more usefully, "which kinds of questions does it get wrong." The companion pipeline ships an evaluation harness for exactly this, `scripts/run_eval.py`, built against a real question set for a different (private, North Sea) campaign it was originally developed on. It has **not yet been run against the Utah FORGE archive** — no evaluation question set exists for this well yet. That's not a gap in the tooling; it's the natural next step, and this chapter walks you through doing it for real, against the same archive you've used throughout this book.

11.3. Building your own question set

Rather than presenting borrowed results, build a small evaluation set against the ten-report Part I sample, where you already know the ground truth from having read the reports yourself in Chapters 1–5:

A starter evaluation set for the Part I sample

```
Q1: "Which report describes pipe becoming stuck during a slide?"
    expected: FORGE-16A-78-32_Drilling_038_2020-11-26.pdf

Q2: "Which report shows packers failing to set?"
    expected: FORGE-16A-78-32_Drilling_049_2020-12-07.pdf

Q3: "Which report describes milling up lost pieces of bit?"
    expected: FORGE-16A-78-32_Drilling_050_2020-12-08.pdf

Q4: "Which report mentions mud fluid seepage losses?"
    expected: FORGE-16A-78-32_Drilling_019_2020-11-07.pdf

Q5: "Which report shows the crew rigging up before spud?"
    expected: FORGE-16A-78-32_Drilling_003_2020-10-22.pdf
```

Each question has exactly one verifiable correct answer, because you picked these five reports for their distinct, checkable events in Chapters 1–5. This is the right way to start an evaluation set: begin with cases you can verify by eye, before scaling to enough volume that you can no longer read every answer yourself.

11.4. Theory

Three metrics, each answering a slightly different question:

- **`recall@k`** — did the correct answer appear *anywhere* in the top k results? Simple, and the right first metric to compute.
- **Mean Reciprocal Rank (MRR)** — averages `1 / rank` of the correct answer across all questions. Rewards getting the right answer to rank #1, not just somewhere in the top 10 — closer to what a user actually experiences.
- **`NDCG@k`** (Normalized Discounted Cumulative Gain) — like recall, but weights a hit at rank 1 more than a hit at rank 5, and normalizes so the score is comparable across queries with different numbers of correct answers.

Engineering Translation: recall@k

`recall@k` asks a plain yes/no question: did the right report make your shortlist of the top `k` results at all? It's like asking whether the correct casing point made your shortlist of candidate depths — it doesn't care whether it was the first one you'd pick or the last, only whether it

was on the list.

Engineering Translation: Mean Reciprocal Rank

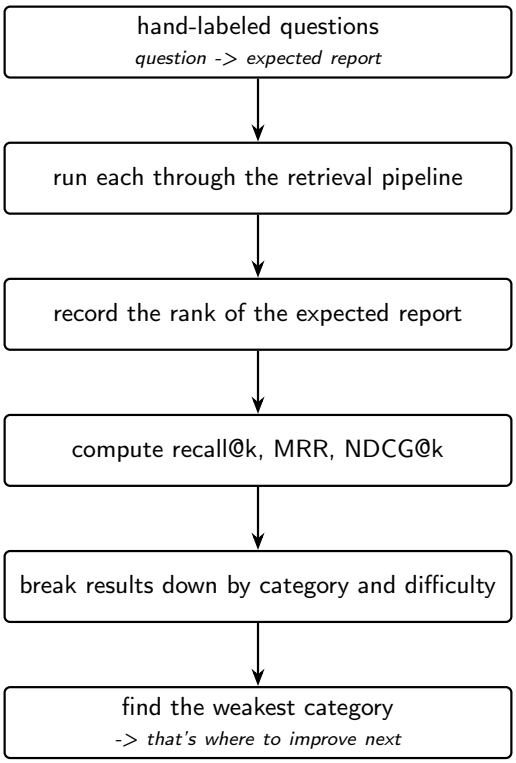
MRR rewards a correct answer for landing near the top, not just somewhere on the list. It's the difference between "the right depth was in my top 5" and "the right depth was my very first pick" — the second is worth more, and MRR is the metric that actually reflects that.

Engineering Translation: NDCG@k

NDCG@k is a more careful version of `recall@k`: instead of a flat yes/no, it gives partial credit based on exactly where the right answer landed — full credit at rank 1, a bit less at rank 2, less still further down — rather than treating "found it at #1" and "found it at #5" as equally good.

None of these require deep statistics to use well — they require a good evaluation *set*: real questions, with a real, known correct source recorded in advance. Building that set by hand, from your own archive, is more valuable than the metric computation itself, because writing the questions forces you to think like the engineer who will actually use the system.

One caveat applies to all three at this book's scale: with five questions over ten reports, every one of these numbers is *coarse*. `recall@k` can only move in whole 20% steps (one question out of five), and a single answer sliding from rank 1 to rank 2 swings MRR and NDCG noticeably. Treat the values you compute here as showing how the metrics *behave*, not as a benchmark — they only start to mean something once the question set is large enough that no single answer can move them much. The challenge exercise's fifteen questions over all 76 reports is the first step in that direction.



11.5. Implementation

11.5.1. Step 1: did the right answer make the shortlist at all?

What problem are we solving?

Check, for every evaluation question, whether the correct report showed up anywhere in the top `k` retrieved results — a simple yes/no per question.

Inputs

- A list of result dicts, one per evaluation question, each recording the rank the correct report was found at (or `None` if it wasn't found).
- `k`: how far down the results to look.

Expected Output

A single number between 0 and 1 — the fraction of questions where the correct report was within the top `k`.

```python
# code/chapter_11/eval_metrics.py
def recall_at_k(results: list[dict], k: int) -> float:
    hits = [r for r in results if r.get("rank") is not None and r["rank"] <= k]
    return len(hits) / len(results) if results else 0.0
```

What just happened?

For each question, this checks whether the correct report's rank is within `k` — a plain hit or miss — then reports what fraction of all questions counted as a hit. It says nothing about *how close to the top* a hit was, which is exactly the gap the next two metrics fill.

11.5.2. Step 2: reward landing near the top, not just anywhere

What problem are we solving?

Give more credit to a correct answer that ranked first than one that merely appeared somewhere in the results — closer to what a user actually experiences when scanning a results list top to bottom.

Inputs

- The same list of result dicts as Step 1.

Expected Output

A single average score: higher when correct answers tend to rank near the top, lower when they're buried further down or missing.

```
def mrr(results: list[dict]) -> float:
    scores = [1.0 / r["rank"] if r.get("rank") else 0.0 for r in results]
    return sum(scores) / len(scores) if scores else 0.0
```

What just happened?

Each question's score is `1 / rank` — rank 1 scores a full 1.0, rank 5 scores only 0.2, and a miss scores 0. Averaging those scores across every question produces one number that rewards *first-place* correct answers far more than answers merely present somewhere in the list.

11.5.3. Step 3: give partial credit based on exact position

What problem are we solving?

Combine the best of both previous metrics: a graded score, like MRR, but one that treats "found in the top k" as the relevant window, like recall@k.

Inputs

- The same list of result dicts, plus `k`.

Expected Output

A single average score, weighted so a hit at rank 1 counts more than a hit at rank 5, and anything beyond `k` counts as zero.

```python
def ndcg_at_k(results: list[dict], k: int) -> float:
    import math
    scores = []
    for r in results:
        rank = r.get("rank")
        scores.append(1.0 / math.log2(rank + 1) if rank and rank <= k else 0.0)
    return sum(scores) / len(scores) if scores else 0.0
```

What just happened?

Instead of `1 / rank` (MRR), each hit's credit shrinks by `1 / log2(rank + 1)` — a gentler slope that still rewards a rank-1 hit the most, still penalizes a rank-5 hit, but doesn't fall off quite as sharply. Anything outside the top `k` scores zero, same as `recall@k`.

Each `result` dict records, per evaluation question, the rank at which the known-correct report was retrieved (or `None` if it wasn't in the top k considered). This mirrors the shape of `_recall_at_k()`, `_mrr()`, and `_ndcg_at_k()` in the companion repository's `scripts/run_eval.py`.

11.6. Production Reality

The evaluation set in this chapter has five questions, and the challenge exercise's has fifteen — genuinely useful, but still small next to a real deployment. A few realities worth planning for:

- an evaluation set needs to keep growing as new report types, wells, and question styles show up — five questions that were exhaustive for the Part I sample won't stay representative of a live system fielding real engineers' questions
- tuning the retrieval pipeline specifically to score well against your own evaluation set risks overfitting to it — the same way memorizing an exam's specific questions isn't the same as knowing the material. Holding some questions back, or continuously adding new ones, keeps the score honest
- an aggregate metric is a summary, not a substitute for periodically reading real answers by eye — especially before a decision with real consequences rides on the system's output
- who writes the evaluation questions matters — a question set written entirely by the person who built the retrieval pipeline tends to test what that person already thought to check, not what a working engineer will actually ask

11.7. Practical exercise

Beginner

Try it yourself: Run the five questions above through Chapter 9's `hybrid_search()` over the ten-report sample archive, record each correct report's rank, and compute `recall@3` by hand.

You'll know it worked when: you have a small CSV or JSON file mapping question → expected report → observed rank, and a `recall@3` score you can quote and defend — because you wrote and verified every question yourself. On this five-question set every expected report lands in the

top three, so `recall@3` comes out to 1.0; don't read that as "retrieval is solved," though — five hand-picked, distinctly-worded questions over ten reports is the easiest case there is, which is exactly why the challenge below scales it up to fifteen questions over all 76.

11.8. Field notes

! Field notes: 'better' isn't monotonic — track one hard question across every chapter

Action: track report #39's rank for the query `"stuck pipe"` through every retrieval method this book has built so far, instead of just looking at one final aggregate score.
Result:

```
Chapter 3  keyword (AND)          not found at all   (0 of 10)
Chapter 4  dense, whole-document  rank 5 of 10
Chapter 9  hybrid, pipeline weights (BM25x2.0/dense x0.5)  rank 9 of 10
Chapter 9  hybrid, equal weights (1.0/1.0)                 rank 7 of 10
```

Why: each chapter's technique is a genuine improvement *in general* — that's not in question. But this specific, operationally important question doesn't improve monotonically as the pipeline gets more sophisticated. It gets a little better (Chapter 4), then gets worse twice in a row (Chapter 9's two weightings) before hybrid retrieval's other strengths (Chapter 9's `"pack ers fishing"` example) show up clearly.
Lesson: a single aggregate metric — `recall@5` averaged across your whole evaluation set — can climb steadily from chapter to chapter while a specific, real question you care about gets worse. This is exactly why Chapter 9 insisted on per-category breakdowns and why this chapter opened by building your own question set rather than trusting someone else's summary number: the only way to know if *your* hard questions are improving is to track them individually, the way this table just did.

11.9. Challenge exercise

Advanced

Challenge: Extend your five-question set to at least 15 questions against the *full* 76-report archive (`datasets/forge_archive/`), and compute `recall@5`. This is genuinely useful, unpublished work: as far as this book's companion pipeline is concerned, you'll be building the first real evaluation set for Utah FORGE. Save your question set alongside your results — it's the seed of exactly the kind of evaluation harness `scripts/run_eval.py` runs at scale once one exists. A reference solution approach is in `code/chapter_11/challenge/`.

Don't be alarmed if `recall@5` comes back much lower than you'd expect from Part I — the reference solution's own 15-question set, run against all 76 reports with Chapter 4's plain whole-document embeddings, scores `recall@5` of only **0.47**. That's not a bug: at 10 candidates, chance alone gets you partway there, and whole-document dilution barely matters. At 76 candidates, both problems compound. A low score here is the honest, measured version of the same lesson Chapter 9 already

taught — it's exactly what motivates hybrid retrieval and reranking before this system sees a real, full-scale archive.

11.10. Key takeaways

- An evaluation set with known correct answers is more valuable than any single retrieval technique — it's what tells you whether a technique helped.
- `recall@k`, MRR, and `NDCG@k` answer different questions; report more than one.
- Don't trust a system's evaluation results from a different dataset as a proxy for how it performs on yours — a harness that scored well on one campaign says nothing certain about a new well until you build and run a question set against that well's own archive, which is exactly what this chapter had you do.

11.11. Repository files

File	Purpose
`code/chapter_11/eval_metrics.py`	`recall@k`, MRR, `NDCG@k` implementations
`DDR_UTAH_FORGE/scripts/run_eval.py`	Evaluation harness (companion repo) — ready to point at a Utah FORGE question set once you build one

CHECKPOINT — Chapter 11

- ☒ Built a hand-labeled evaluation set with known correct answers
- ☒ Computed `recall@k`, MRR, and `NDCG@k`, and can explain what each one tells you
- ☒ Tracked one hard question across every retrieval method in this book
- ☒ Reported results broken down by category instead of one aggregate score

WHAT YOU BUILT

`eval_metrics.py` plus a real, hand-verified evaluation set — the first ever built against this archive, with a defensible number attached to "how good is it."

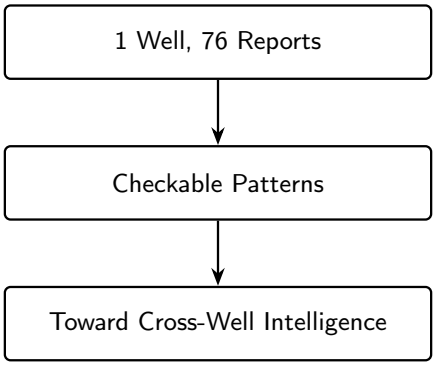
11.12. What can you do now that you couldn't do before?

You can put a defensible number on how well your retrieval system actually works, broken down by the kinds of questions it handles well and the ones it doesn't — instead of shipping a decision-support tool on the strength of "it feels like it's working."

11.13. Suggested next step

Coming up in Chapter 12: Every discipline from Chapters 6–11 — OCR gating, structured chunking, hybrid retrieval, traceability, evaluation — exists in service of a single payoff: finding things in an archive a human reading it linearly would miss. Chapter 12 builds toward that payoff, using a real sequence of events already in this archive.

12. Sequence Detection: Building Toward Cross-Well Intelligence



Progress ########## **12 / 12** · **Estimated time:** 60–90 min · **Difficulty:** Advanced

12.1. Learning objectives

By the end of this chapter, you will be able to:

- Combine structured numeric trends (torque, ROP) with narrative operation text to build a single, checkable operational story.
- Explain how a windowed leading-indicator detector works — and verify, from real thresholds in production code, exactly what "causal" and "escalating" mean before trusting either label.
- Understand honestly what this book's single-well archive can and can't demonstrate, and what it would take to scale this technique to genuine cross-well intelligence.

12.2. Operational Problem

Sarah, the intervention engineer, asks the payoff question every previous chapter has been building toward: *"Did something earlier predict trouble later?"* A human reading 76 reports in sequence can, with effort, spot a pattern — but a system that has structured, classified, and indexed the same text can hold every report in mind at once, and check numeric trends a human would have to read a table on every single page to notice.

This chapter is deliberately honest about scale: Utah FORGE's public archive is **one well, one continuous drilling programme** — not the multi-well, multi-phase campaign this book's companion pipeline was originally built to analyse. So instead of presenting a large statistical

finding, this chapter shows you exactly how to check a real, small pattern by hand, using the same technique — and shows you precisely what production code exists to scale that technique up once you have more than one well's worth of data.

12.3. A real, checkable pattern

Report #38's `TORQUE` header field — a single number recorded every day regardless of what happens — tells a real story across three consecutive days:

Real header data, reports #36-38

```
Report #36 (2020-11-24): TORQUE: 4,400
Report #37 (2020-11-25): TORQUE: 4,200
Report #38 (2020-11-26): TORQUE: 5,615 (header) / 6,500 (during the slide)
                          "During the slide lost tool face and became
                           assembly became stuck"
```

Torque was flat — even slightly down — for two days, then jumped by roughly a third on the same day the assembly became stuck.

Engineering Translation: Leading indicator vs. same-day correlation

A **leading indicator** is a signal that shows up *before* the trouble — like a rising torque trend days ahead of an actual stuck-pipe event, which would genuinely give you warning. A **same-day correlation** just means two things moved together on the same day — useful evidence that they're related, but it confirms trouble, it doesn't predict it. This chapter's torque pattern is the second kind, and it matters to be precise about which one you're actually looking at.

This is a same-day correlation between a structured number and a narrative event, not a multi-day advance warning — and that distinction matters. Don't claim a leading indicator you haven't actually verified extends before the event; this pattern says "torque and the stuck-pipe event moved together," not "torque predicted the stuck pipe two days out."

Chapter 5's other real sequence — report #49's packers failing to set, followed same-day by picking up a fishing BHA, followed by report #50's actual fishing operation — is a genuine two-report causal chain, directly traceable and already fully verified with real text.

12.4. Theory: how the production tool would check this at scale

The companion pipeline's `src/ddr_rag/causality_analyzer.py` — part of `DDR_UTAH_FORGE`, not this book's repository — implements exactly this kind of check, generalized: for a given term, it compares frequency in a window before and after a boundary date, and labels a term **causal** only above a defined post-boundary NPT rate, and **escalating** only above a defined frequency-increase ratio. The thresholds below are reported directly from that module, not something you can inspect or run with this book's own bundled code:

```
TRANSITION_WINDOW_DAYS = 14   # days before/after a boundary to compare
MIN_TERM_FREQ = 3             # minimum occurrences before a term counts at all
NPT_SIGNAL_THRESHOLD = 0.40   # post-boundary NPT rate to call a term "causal"
ESCALATION_FREQ_RATIO = 1.5   # frequency increase to call a term "escalating"
```

Engineering Translation: Threshold constants

These four lines are named, readable numbers you could point to on a whiteboard and defend in a meeting — "we call it escalating once frequency rises by 50%" — rather than a judgment buried invisibly inside a trained model's weights. Anyone can read them, question them, and change them without touching the logic around them.

Every one of these is a plain, readable number you can inspect, change, and justify — not a weight buried inside a trained model.

A real limitation worth understanding, not hiding

This module's default phase configuration — `["MIRU", "COND1", "INTRM1", "INTRM2", "PROD1", "COMPZN"]` — comes from `operator_alpha`, a different (private, North Sea) campaign's phase vocabulary. Utah FORGE's real data doesn't have that phase structure: its `ddr_facts.parquet` phase field is `"Production Drilling"` for essentially the entire programme, right up to the Completion-DFIT report. Running `causality_analyzer.py` against this archive as-is would compare windows around phase boundaries that don't meaningfully exist in this well's data — the code would run, but the output wouldn't mean anything useful yet.
This is a genuinely common situation in real engineering work: code built and validated on one dataset doesn't automatically transfer to another just because the file paths line up. Before trusting output from an adapted pipeline, check that the assumptions baked into its configuration (here, a phase vocabulary) actually match your data. The `src/ddr_rag/npt_classifier.py` module, by contrast, *has* been adapted for Utah FORGE specifically — `classify_utah_forge_npt()` exists and recognises this archive's real fields — which tells you adaptation here is partial, not absent: some layers of this pipeline are ready for this well's data, and some aren't yet.

12.5. Implementation

12.5.1. Step 1: measure day-over-day change, not just the raw numbers

What problem are we solving?

Turn a list of daily torque readings into a measured day-over-day percentage change, so "roughly a third" becomes an exact, checkable number.

Inputs

- `readings`: a chronological list of `(date, torque_value)` pairs.

Expected Output

A list of `(date, torque_value, percentage_change)` — one entry for every day after the first, since a percentage change needs a previous day to compare against.

```
# code/chapter_12/torque_trend_check.py
def torque_trend(readings: list[tuple[str, float]]) -> list[tuple[str, float,
 ↪  float]]:
    """Given [(date, torque_value), ...] in chronological order, return
    each day's percentage change from the previous day."""
    trend = []
    for (prev_date, prev_val), (date, val) in zip(readings, readings[1:]):
        pct_change = (val - prev_val) / prev_val if prev_val else 0.0
        trend.append((date, val, pct_change))
    return trend
```

What just happened?

This walks through the readings one consecutive pair at a time and computes how much each day's torque changed relative to the day before, as a percentage — the exact arithmetic behind the "flat, then a third higher" story told in prose above.

12.5.2. Step 2: run it against report #38's real numbers

What problem are we solving?

See the actual percentage change for reports #36–38, printed as real numbers instead of an approximate description.

Inputs

- The three real `TORQUE` header values from reports #36, #37, and #38.

Expected Output

```
2020-11-25: 4200 (-4.5%)
2020-11-26: 5615 (+33.7%)
```

```
readings = [
    ("2020-11-24", 4400),
    ("2020-11-25", 4200),
    ("2020-11-26", 5615),
]
for date, val, pct in torque_trend(readings):
    print(f"{date}: {val} ({pct:+.1%})")
```

What just happened?

Running the real header values through `torque_trend` confirms the pattern precisely: a 4.5% *dip* the day before, then a 33.7% jump on the stuck-pipe day itself — turning "roughly a third" into an exact, reproducible number sourced from the report headers themselves.

12.6. Production Reality

This chapter's torque check works because Utah FORGE's header format is completely consistent, and because a single well gives you exactly one example of the pattern. Genuine cross-well intelligence has different demands entirely:

- different rigs and BHA configurations have different normal torque ranges — a 25% jump threshold tuned on this well could be meaningless, or far too sensitive, on a well drilled with different equipment
- not every operator's DDR software populates a header field as cleanly as WellEz does here — a real multi-operator archive may need per-source parsing before any threshold-based check is even possible
- one well gives you one instance of a pattern, which is a hypothesis, not a finding — genuine cross-well intelligence needs enough wells to tell a real, recurring precursor apart from coincidence, which is exactly why the companion pipeline's `causality_analyzer.py` was built against a many-well campaign in the first place
- as the callout above shows, code built and tuned for one dataset's phase vocabulary doesn't just work on a new dataset because the file paths match — every threshold and configuration assumption needs re-checking against the new data before you trust its output

12.7. Practical exercise

Beginner

Try it yourself: Extend `readings` with report #39's header torque value (check the PDF yourself — Chapter 1's `read_ddr.py` will get it for you) and confirm whether the elevated torque persisted after the pipe was freed, or dropped back toward the pre-event baseline.

You'll know it worked when: you can state the real percentage change for each day, sourced from header fields you extracted yourself, not from this chapter's text.

12.8. Field notes

⚠ Field notes: the pattern doesn't stop at the stuck-pipe day

Action: read report #39's narrative text — the day *after* the stuck-pipe day this chapter's torque numbers track — the same lines Chapter 3's Field Notes already flagged as invisible to keyword search.
Result: report #39 says:

- "Work tight hole at 6,526'."
- "Due to high torque decision to pull out of hole"
- "Hole drag from 6,050' to 5,901' no issues"

Continued elevated-torque language, one calendar day after report #38's header `TORQUE` value jumped 33.7% on the stuck-pipe day itself.
Why: this chapter's structured trend check only looks at the `TORQUE` header field across three days, and stops at report #38. But report #39's narrative — high torque, a tight hole, a deliberate decision to pull out of hole — reads like the same operational story continuing in prose, one day after the structured number spiked.
Lesson: a single structured signal, however precise, is one thread of a larger story. Report #39 is exactly the kind of continuation a torque-only check would miss entirely, but that hybrid retrieval and narrative search (Chapters 3-9 combined) can still surface. That's the whole book's argument, compressed into one field note: no single technique in this book is sufficient alone — structured checks, narrative retrieval, and traceable generation each catch what the others miss.

12.9. Challenge exercise

Advanced

Challenge: Pull the `TORQUE` header value for all 76 reports in `datasets/forge_archive/` (Chapter 1's extraction plus a small regex), plot it as a time series, and identify every day where torque jumps more than 25% from the previous day. Cross-reference each jump against that day's narrative text — how many of them correspond to a real operational event like report #38's, and how many are unremarkable? A reference solution is in `code/chapter_12/challenge/`.

12.10. Key takeaways

- A single structured number, tracked day over day, can corroborate a narrative event — but be precise about what you've actually shown: same-day correlation is not the same claim as a multi-day leading indicator, and this chapter deliberately didn't claim more than the real data supports.
- Production causality-detection code carries real, inspectable thresholds — read them before trusting output, and check that its configuration (like a phase vocabulary) actually matches your dataset before running it.
- Genuine cross-well intelligence needs more than one well's worth of data to compare against — Utah FORGE's `data/fields/UtahForge/wells/` structure in the companion pipeline is already organized to support additional wells as this research programme (or your own multi-well archive) grows.

12.11. Repository files

File	Purpose
`code/chapter_12/torque_trend_check.py`	Day-over-day trend check on a structured header field
`DDR_UTAH_FORGE/src/ddr_rag/causality_analyzer.py`	Windowed leading-indicator detection (companion repo)
`DDR_UTAH_FORGE/src/ddr_rag/npt_classifier.py`	Utah-FORGE-adapted NPT classification (companion repo)
`DDR_UTAH_FORGE/data/fields/UtahForge/wells/`	Multi-well data structure, ready for expansion (companion repo)

CHECKPOINT — Chapter 12

- ☒ Combined a structured numeric trend with narrative text into one checkable story
- ☒ Read real thresholds from production causality-detection code before trusting its labels
- ☒ Distinguished a same-day correlation from an unverified leading indicator
- ☒ Identified exactly what this single-well archive can't yet demonstrate

WHAT YOU BUILT

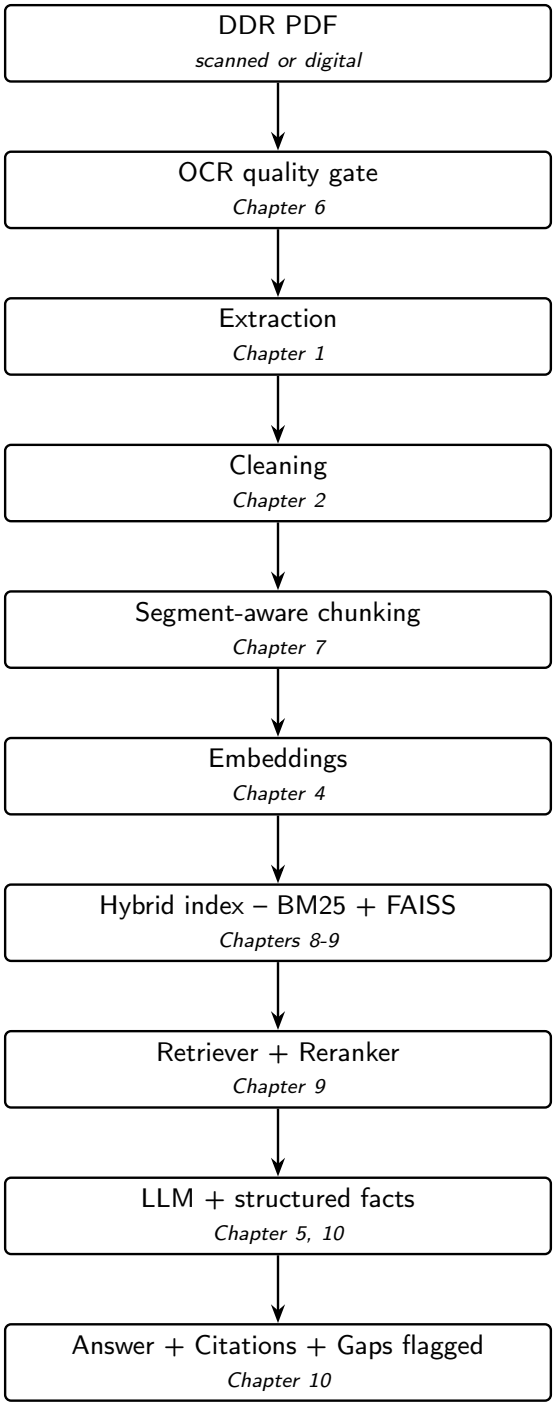
`torque_trend_check.py` — a structured sequence-detection check, and the first building block toward genuine cross-well intelligence: the same technique the companion pipeline's `causality_analyzer.py` runs at scale once more than one well's data exists.

12.12. What can you do now that you couldn't do before?

You can check, by hand and with real numbers, whether a structured trend and a narrative event actually moved together — and you know precisely what's still missing (more wells, re-tuned thresholds, a matching phase vocabulary) before that check becomes genuine cross-well operational intelligence.

12.13. The complete system

Twelve chapters ago, this was a single PDF and a five-line function. Here's everything you built, end to end — every arrow below is a chapter you completed, not an aspiration:



Two more pieces sit alongside this pipeline rather than inside it — they don't run on every query, but they're what make the rest of it trustworthy at scale:

```
Evaluation (Chapter 11) ---------- measures how well the pipeline above
                                   actually performs, against questions
                                   with known correct answers
```

```
Sequence detection (Chapter 12) -- mines what the pipeline has already
                                   indexed for patterns no single query
                                   would surface on its own
```

Every stage exists because an earlier, simpler version of this system failed at it — a scanned page with no text, a chunk that split a stuck-pipe sentence in half, a query that missed a report because it used the wrong three words, a hallucinated number where a table lookup belonged. None of the complexity here is complexity for its own sake.

12.14. Suggested next step

Where to go from here: You've now built, chapter by chapter, a working RAG system grounded entirely in real, public DDRs — from extracting a single PDF's text through checking a real numeric trend against a real narrative event. Appendix A shows how to set up the full companion pipeline and point it at your own DDR archive, whether that's more Utah FORGE data as it becomes available, or a different well entirely.

Part IV.

Appendices

Appendix A: Environment Setup

Part 0 covers everything you need to install Python, create a virtual environment, and run the book's chapter scripts — start there if you haven't already. This appendix covers a few things Part 0 doesn't: the sample dataset's layout, rendering this book itself (not just running its code), the companion DDR Intelligence Platform pipeline referenced in Chapters 6–12, a data-handling note for your own archives, and troubleshooting for a few less common errors.

If you'd like a guide specific to the editor or notebook environment you chose in Part 0, see Appendices A1–A5 instead.

1. The sample dataset

Chapters 1–5 use ten real, curated Daily Drilling Reports from the public Utah FORGE archive (well FORGE 16A(78)-32) — already committed as PDFs in `datasets/sample_ddrs/`, since this is public data with no confidentiality concerns. Part II's "at scale" chapters use the full 76-report archive, committed at `datasets/forge_archive/`. Nothing needs to be generated or downloaded to start: cloning the repository (Part 0, Section 0.6) gives you the full dataset.

If you want to reproduce either curated subset from your own copy of the full public archive, or extend it with more reports, see `code/chapter_01/build_sample_archive.py` and `code/chapter_08/build_full_archive.py`.

2. Rendering the book with Quarto

Running the chapter scripts (Part 0) doesn't require anything beyond Python. Rebuilding this book itself — the HTML and PDF you're reading — is a separate step that needs Quarto 1.7 or later:

```
quarto --version
```

If that's not recognised, install Quarto from quarto.org for your operating system.

With Quarto installed and your virtual environment active:

```
quarto render
```

Output is written to `_book/`. For live-editing a single chapter while you write:

```
quarto preview chapters/chapter_01.qmd
```

Or, if you prefer conda/mamba instead of the `.venv` approach from Part 0:

```
conda env create -f environment.yml
conda activate ddr-rag-book
```

3. The companion pipeline (for Part II)

Chapters 6–12 reference **DDR_UTAH_FORGE**, a real, working DDR intelligence pipeline built specifically against this book's public archive — same architecture as the reference platform this project grew from (per-document extraction, hybrid retrieval, structured facts, causality analysis), adapted for Utah FORGE's real filenames and data. It's a separate repository — clone it alongside this book's repository, not inside it:

```
cd ../..                    # up and out of this book's own repository
git clone https://github.com/djimrastephane/DDR_UTAH_FORGE.git
cd DDR_UTAH_FORGE
python -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt
```

Its own `data/` folder is gitignored — the repository is code only. Copy this book's `datasets/forge_archive/` into `DDR_UTAH_FORGE/data/raw/` (or point it at your own copy of the public Utah FORGE archive) and follow its `README.md` Quickstart to run the full pipeline and Streamlit dashboard.

> **You don't need it to follow along**
>
> Part II chapters are written so every technique works fully standalone: the simplified code in `code/chapter_06/` through `code/chapter_12/` in *this* book's repository requires no access to the companion pipeline. Where a chapter cites a specific file or a specific result from `DDR_UTAH_FORGE` (for example, the real 1,428-chunk global index in Chapter 8, or the real report-gap dates in Chapter 10), that result was checked against the pipeline's actual output before being written into the book, so you can trust the numbers even without running the pipeline yourself.

4. A note on data handling

This book's own data (`datasets/sample_ddrs/`, `datasets/forge_archive/`) is public and safe to commit, share, and publish — that's true throughout this repository. The moment you point this book's code at your own organisation's DDRs instead, that changes:

- Never commit real, confidential DDR PDFs, extracted text, or derived artefacts (embeddings, indexes) to a public repository.

- If you're experimenting on a shared machine, confirm your organisation's data classification policy covers running third-party Python packages (embedding models, OCR engines) against confidential well data.
- Add your own archive's path to `.gitignore` before dropping any confidential PDFs into a local `datasets/` folder for testing.

5. Troubleshooting

Part 0, Section 0.11 covers the common setup errors (Python not found, virtual environment not active, and so on). A few more specific to rendering the book or running Part II's heavier packages:

Symptom	Likely cause
`quarto render` fails on a code cell	Activate `.venv` before rendering — Quarto uses whichever Python is on `PATH`. Confirm with `which python` and `python -c "import pdfplumber"`.
`ModuleNotFoundError: sentence_transformers` (Chapter 4+)	Re-run `pip install -r requirements.txt`; this is a heavier dependency and sometimes needs a separate `pip install torch` first on some platforms.
FAISS import error on Apple Silicon (Chapter 8+)	Ensure you installed `faiss-cpu`, not `faiss-gpu` — the latter has no macOS wheel.
`read_ddr.py` reports no text extracted	Confirm you're pointing at a file in `datasets/sample_ddrs/` or `datasets/forge_archive/` — both are committed directly, so this shouldn't happen unless the path is wrong.

Appendix A1: Using Jupyter Notebook

This appendix assumes you've completed Part 0 — Python installed, `.venv` created and active, packages installed via `pip install -r requirements.txt`. Everything below is specific to running this book's code inside Jupyter Notebook instead of from a plain terminal.

Install the notebook

`requirements.txt` already installed Jupyter for you in Part 0, so you can likely skip straight to "Start the notebook" below. If you skipped that step, or only want Jupyter on its own:

```
pip install notebook
```

Expected output: a `Successfully installed ...` line mentioning `notebook`.

Start the notebook

With your virtual environment active (you should see `(.venv)` in your terminal prompt) and from inside the `ddr-rag-book` project folder:

```
jupyter notebook
```

What this does: starts a local notebook server and opens a new browser tab showing the contents of your project folder.

If nothing opens automatically, the terminal will print a URL starting with `http://localhost:8888/...` — copy and paste that into your browser.

Open a chapter notebook

Every chapter of Part I has an interactive companion notebook. In the browser tab Jupyter opened, click into the `notebooks/` folder, then open `chapter_01_explore.ipynb` (or whichever chapter you're on).

Run a cell

A notebook is made of cells — small blocks of code you run one at a time. Click into a cell, then press **Shift+Enter** to run it and move to the next one. The output (printed text, or an error) appears directly below the cell.

Expected output: for Chapter 1's notebook, running the extraction cell prints the real report text described in that chapter.

Stop the notebook server

When you're done, go back to the terminal window where you ran `jupyter notebook` and press **Ctrl+C**, then confirm if asked. Closing the browser tab alone does not stop the server — the terminal is where it's actually running.

Appendix A2: Using VS Code

This appendix assumes you've completed Part 0 — Python installed, `.venv` created and active, packages installed. Everything below is specific to working in VS Code instead of a plain terminal.

Install VS Code

Download it from code.visualstudio.com and run the installer for your operating system — no special options needed.

Install the Python extension

VS Code doesn't understand Python out of the box; a small add-on teaches it to. Open VS Code, click the square-icon **Extensions** tab in the left sidebar, search for **Python** (published by Microsoft), and click **Install**.

Open the project folder

Use **File → Open Folder...** and select the `ddr-rag-book` folder you created in Part 0. VS Code will use this as its workspace — every file in the sidebar is now part of this project.

Select the interpreter

> Engineering Translation: Interpreter
>
> An **interpreter** is simply which copy of Python VS Code should run your code with. You want it pointed at the `.venv` toolbox from Part 0, not some other Python installation on your machine.

Press **Cmd+Shift+P** (Mac) or **Ctrl+Shift+P** (Windows/Linux) to open the command palette, type `Python: Select Interpreter`, and choose the one showing a path ending in `.venv/bin/python` (Mac/Linux) or `.venv\Scripts\python.exe` (Windows). If you don't see it listed, click "Enter interpreter path" and browse to that file directly.

Open a terminal

Use **Terminal → New Terminal** from the top menu. VS Code opens a terminal already positioned inside your project folder — and if you selected the right interpreter above, it should already show `(.venv)` at the start of the prompt.

Run a script

Either:

- Type the command directly into the terminal you just opened, e.g.:

```
python code/chapter_01/read_ddr.py
 ↪  datasets/sample_ddrs/FORGE-16A-78-32_Drilling_038_2020-11-26.pdf
```

- Or open the `.py` file in the editor and click the **Run** ( ) button in the top-right corner — VS Code runs it in an integrated terminal for you.

Both produce identical output — use whichever feels more natural.

Appendix A3: Using PyCharm Community

This appendix assumes you've completed Part 0 — Python installed, `.venv` created and active, packages installed. Everything below is specific to working in PyCharm Community (the free edition — this book needs nothing from the paid Professional edition).

Install PyCharm Community

Download it from jetbrains.com/pycharm/download — select the **Community** edition, which is free, and run the installer.

Open the project

From PyCharm's welcome screen, choose **Open** and select the `ddr-rag-book` folder you created in Part 0. PyCharm treats this folder as one project.

Select the existing .venv

PyCharm may offer to create its own virtual environment — decline that and point it at the `.venv` you already created in Part 0 instead, so you don't end up with two separate toolboxes:

1. Go to **PyCharm → Settings** (Mac) or **File → Settings** (Windows/Linux).
2. Navigate to **Project: ddr-rag-book → Python Interpreter**.
3. Click the gear icon → **Add → Existing environment**.
4. Browse to `.venv/bin/python` (Mac/Linux) or `.venv\Scripts\python.exe` (Windows) inside your project folder, and select it.

Open a terminal

PyCharm has a built-in terminal along the bottom of the window — click the **Terminal** tab. It opens already positioned inside your project folder, and should show `(.venv)` in the prompt once the interpreter above is set correctly.

Run a script

Two options, same result:

- Type the command into PyCharm's terminal tab:

```
python code/chapter_01/read_ddr.py
 ↪  datasets/sample_ddrs/FORGE-16A-78-32_Drilling_038_2020-11-26.pdf
```

- Or open the `.py` file in the editor and click the green **Run** ( ) arrow next to the file, or right-click inside the file and choose **Run**.

Run a Python file directly

Right-clicking any `.py` file in PyCharm's project sidebar and choosing **Run ‘’** does the same thing as typing `python <path/to/file.py>` in the terminal — pick whichever is more comfortable as you go through the book.

Appendix A4: Using Positron

This appendix assumes you've completed Part 0 — Python installed, `.venv` created and active, packages installed. Positron isn't required to follow this book — it's simply one of several environments that work well, particularly if you like moving between notebooks and plain scripts in the same window. Everything below is specific to working in Positron instead of a plain terminal.

Install Positron

Download it from positron.posit.co and run the installer for your operating system.

Open the project folder

Use **File → Open Folder...** and select the `ddr-rag-book` folder from Part 0. Positron uses this as your workspace root.

Select the Python environment

Positron shows an interpreter picker in the bottom status bar. Click it and select the `.venv` you created in Part 0 — the same environment every other tool in this appendix set points at.

Open a terminal

Use **Terminal → New Terminal** from the top menu, or the terminal icon in the bottom panel. If the right interpreter is selected above, this terminal should already show `(.venv)` in its prompt.

Run a script

```
python code/chapter_01/read_ddr.py
 ↪  datasets/sample_ddrs/FORGE-16A-78-32_Drilling_038_2020-11-26.pdf
```

Or open the `.py` file and use Positron's **Run** button in the editor toolbar.

Run notebooks

Positron runs Jupyter notebooks (`.ipynb`) and Quarto documents (`.qmd`) natively — open `notebooks/chapter_01_explore.ipynb` directly, and run each cell with **Shift+Enter**, the same shortcut as Jupyter Notebook (Appendix A1). For a `.qmd` file — this book's own chapters are written as `.qmd` — each ```` ```{python} ```` block behaves like a notebook cell, and the **Render** button in the editor toolbar builds the whole document, code and all.

Appendix A5: Terminal-Only Workflow

This appendix assumes you've completed Part 0. If you'd rather not install any editor at all, everything in this book runs from the terminal alone — you'll just open and edit files with whatever plain text editor came with your system (Notepad on Windows, TextEdit on Mac, `nano` or `vim` on Linux) instead of a dedicated code editor.

Navigate to the project

```
cd ddr-rag-book
```

Confirm you're in the right place with `pwd` (Mac/Linux) or `cd` alone (Windows) — it should end in `ddr-rag-book`.

Activate the environment

Mac/Linux:

```
source .venv/bin/activate
```

Windows:

```
.venv\Scripts\activate
```

Expected output: your prompt now starts with `(.venv)`. Do this every time you open a new terminal window to work on this book — activation doesn't persist between sessions.

Install packages

Only needed once, or again if `requirements.txt` changes:

```
pip install -r requirements.txt
```

Run scripts

Every chapter's script runs the same way:

```
python code/chapter_01/read_ddr.py
 ↪  datasets/sample_ddrs/FORGE-16A-78-32_Drilling_038_2020-11-26.pdf
```

Swap in whichever chapter and arguments that chapter's text shows you.

Render the book

If you want to build the book itself (this document, as HTML or PDF) rather than just running the chapter scripts, Quarto needs to be installed separately — see Appendix A, Section 2 for the render command and output location. This step is entirely optional: nothing about following the chapters and running their code requires rendering the book yourself.

Appendix B: Oilfield Abbreviation Glossary

This glossary is a bigger version of the same equipment lookup table introduced in Chapter 2 — it supports Chapter 2's abbreviation expansion engine and its challenge exercise. It is broader than the `ABBREVIATIONS` dict built in Chapter 2 itself — that chapter's dictionary is deliberately scoped to terms verified present in this book's real Utah FORGE archive. This appendix adds enough additional common oilfield terms to give the expander real coverage on a first pass against a different archive. It is still not exhaustive — abbreviation usage varies by operator, rig contractor, and basin.

Terms marked **(FORGE)** are confirmed present in this book's real sample archive; the rest are common industry terms not found in this particular dataset but likely to appear in others.

Abbreviation	Expansion
AFE	authorization for expenditure (FORGE)
BHA	bottom hole assembly (FORGE)
BOP	blowout preventer (FORGE)
CBL	cement bond log
CIRC	circulate
COND	condition
CSG	casing
DFIT	diagnostic fracture injection test (FORGE)
DFS	days from spud (FORGE)
DHSV	downhole safety valve
DLS	dogleg severity (FORGE)
DOL	days on location (FORGE)
DP	drill pipe (FORGE)
ECD	equivalent circulating density (FORGE)
EMW	equivalent mud weight
FIT	formation integrity test
FR	from
GPM	gallons per minute (FORGE)
HWDP	heavy weight drill pipe (FORGE)
KOP	kick-off point
LOT	leak-off test
MD	measured depth (FORGE)
MIRU	move in, rig up
MW	mud weight (FORGE)
MWD	measurement while drilling (FORGE)
NMDC	non-magnetic drill collar (FORGE)
NPT	non-productive time (FORGE)
OBSD	observed

Abbreviation	Expansion
PBTD	plugged back total depth (FORGE)
PDC	polycrystalline diamond compact (bit type) (FORGE)
PJSM	pre-job safety meeting (FORGE)
POOH	pull out of hole
PPG	pounds per gallon
PU	pick up (pipe)
PUW	pick up weight
RIH	run in hole
RKB	rotary kelly bushing (FORGE)
ROP	rate of penetration (FORGE)
RPM	revolutions per minute (FORGE)
SICP	shut-in casing pressure
SIDPP	shut-in drill pipe pressure
SLM	slow (rate), or short for "slim hole" depending on context — verify against operator style guide
SPP	stand pipe pressure (FORGE)
STDS	stands (of drill pipe) (FORGE)
STK	stroke length (pump spec) (FORGE)
TD	total depth (FORGE)
TIH	trip in hole
TOOH	trip out of hole
TVD	true vertical depth (FORGE)
UBHO	universal bottom hole orientation sub (FORGE)
UBI	ultrasonic borehole imager (wireline log) (FORGE)
WOB	weight on bit (FORGE)
WOC	wait on cement
WOW	wait on weather
XO	crossover (sub)

> ⚠ **Abbreviations are not universal**
>
> The same three letters can mean different things across operators and basins — build your abbreviation dictionary from *your* archive's actual usage before trusting it against your own reports, and treat any ambiguous abbreviation (like `SLM` above) as a signal to check context rather than expand automatically.

References

Dawson, Colin, and Harry Verkuil. 2014. "From a Daily Drilling Report to a Data and Performance Management Tool." *SPE Intelligent Energy International Conference and Exhibition*.

Ji, Ziwei, Nayeon Lee, Rita Frieske, et al. 2023. "Survey of Hallucination in Natural Language Generation." In *ACM Computing Surveys*, No. 12, vol. 55. https://doi.org/10.1145/3571730.

Johnson, Jeff, Matthijs Douze, and Hervé Jégou. 2017. *Billion-Scale Similarity Search with GPUs*. https://arxiv.org/abs/1702.08734.

Lewis, Patrick, Ethan Perez, Aleksandra Piktus, et al. 2020. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." *Advances in Neural Information Processing Systems* 33: 9459–74.

Matthes, Eric. 2023. *Python Crash Course: A Hands-on, Project-Based Introduction to Programming*. 3rd ed. No Starch Press.

Reimers, Nils, and Iryna Gurevych. 2019. "Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks." *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*.

Robertson, Stephen, and Hugo Zaragoza. 2009. "The Probabilistic Relevance Framework: BM25 and Beyond." *Foundations and Trends in Information Retrieval* 3: 333–89.