A HANDBOOK, PUBLISHED IN THE OPEN

The Data Collection Handbook

Best practices for the data collection industry. Written to be easy to read, backed by data, and replicable by anyone.

David Martin Riveros

v1.0 · August 23, 2026 · davidmartinriveros.com/handbook

The Data Collection Handbook

Best practices for the data collection industry. Written to be easy to read, backed by data, and replicable by anyone.

By David Martin Riveros

v1.0, built August 23, 2026. All 14 chapters, complete. The always-current edition lives at https://www.davidmartinriveros.com/handbook/

About this book

I have spent years building systems that collect web data at scale, and most of what I know arrived the expensive way: through production incidents, wrong datasets, and lessons no book had written down. This handbook is my attempt to write them down, in the open, for the whole industry: the engineers and analysts who build collectors, and the founders and buyers who pay for the results.

Three rules govern every chapter. First, plain language: no jargon without a one-line definition. Second, evidence: each chapter runs one analytical experiment, with data and runnable code published alongside it, so you can replicate every result instead of taking my word. Third, the written analysis carries the full conclusion, so you get the finding even if you never run the code.

One scope note. This book covers collecting publicly visible data at fair load. It does not teach collecting from behind logins, defeating CAPTCHAs, or evading any specific vendor's detection product. Where anti-bot systems appear, the treatment is how detection works and why coherent, respectful traffic is the durable strategy. Those are principled boundaries, not oversights.

Contents

PART I. FOUNDATIONS

Chapter 1. The Data Collection Landscape

Picture two companies selling the same espresso machine online. On a Tuesday night, a marketplace seller undercuts them both. The first company knows by morning: a collection job read the public listings overnight, the change landed in a dashboard, and a category manager repriced before lunch. The second company finds out in the quarterly review, from a slide, after the lost sales have already gone somewhere else. Same product, same market, the same listing sitting in public view. One company treats the public web as raw material for decisions and collects it on purpose. The other is flying blind and calling it normal.

Closing that gap is the industry this book is about. The work has a plain definition: using software to fetch publicly visible web pages and app responses, at a load the source can comfortably serve, and turning what comes back into records. A record is one delivered row of data. One product with its price, one flight with its fare. Decisions run on this raw material, and the raw material about the world outside your own walls, what competitors charge, what is actually in stock where, sits mostly on the open web.

I have spent years building and running these systems as a founder, mostly for retail and travel pricing. One promise before we start: every term of art gets defined the first time it appears, the way I just did with record, because this book is written for the person who signs the data contract as much as for the person who writes the code. The other thing I can tell you from inside the field is that it is wider than the word "scraping" suggests. It is a supply chain: sources on one side, decisions on the other, and a set of access methods in between whose cost and reliability decide whether it makes economic sense.

Who collects, and for what

Pricing intelligence is the vertical I know best and the easiest to picture, because you just met it. Retailers, airlines, hotel groups, marketplace sellers: anyone whose prices must answer a market that moves daily. Take the espresso machine story, multiply it by a catalog of tens of thousands of products, and run it every night. The deliverable is a reflex the company did not have before.

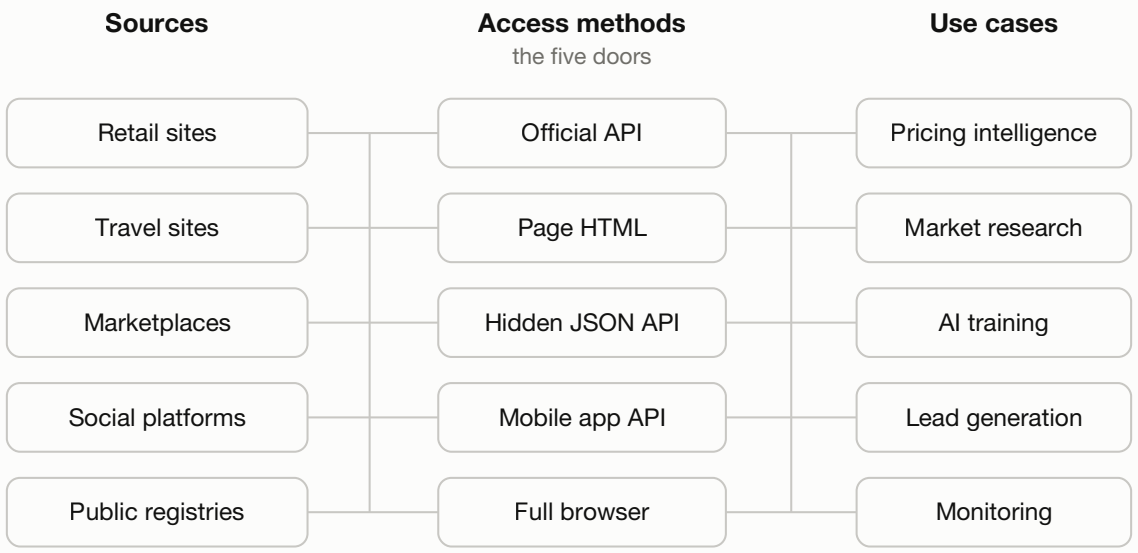
Market research buys breadth instead of speed. An analyst wants to know whether a category is growing, which brands are gaining share of shelf on the big marketplaces, how ratings and review counts moved after a launch. Those signals sit in public view, and in my experience, collected steadily across enough sources, they move earlier than official numbers do. Research firms collect the signals and sell the reading.

Then there are the teams assembling AI training data, the newest buyers and the hungriest ones. They collect public text and images at volumes the rest of the industry considers absurd. The practical question this vertical puts to a publisher is whether its public pages have become someone's training

set without anyone deciding it; the next chapter's census measures how the biggest sites are answering.

Lead generation runs on public registries and directories. A sales team that collects new company filings and fresh job postings stops buying stale lists and starts calling the company that began hiring, this week, for the exact problem its product solves.

Monitoring is the fifth vertical, the one that never makes conference slides. Brands watch resellers for broken pricing agreements. Compliance teams hunt counterfeit listings. Publishers track where their content reappears. Nothing is priced and nothing is forecast; the deliverable is an alert that fires when the web changes in a way somebody promised it would not.



The industry on one page. Sources on the left, the five doors in the middle, use cases on the right. Any source can be reached through any door; the door decides the cost and the fragility of the collection.

The five verticals share one supply chain. They differ in how fresh and how broad the records must be. Getting in is the next problem.

The five doors into any source

Every source I have worked with offers at most five ways in, and this book uses one canonical list for them, here and in every chapter that follows: the official API, the page HTML, the site's own hidden JSON API, the mobile app's API, and a full browser. I think of them as five doors into the same building, because the data behind them is identical. What changes from door to door is the price of walking through and how often it slams on you. An API, for the non-engineers reading: an interface a

site publishes so software can request data directly and get a structured answer instead of a page meant for human eyes.

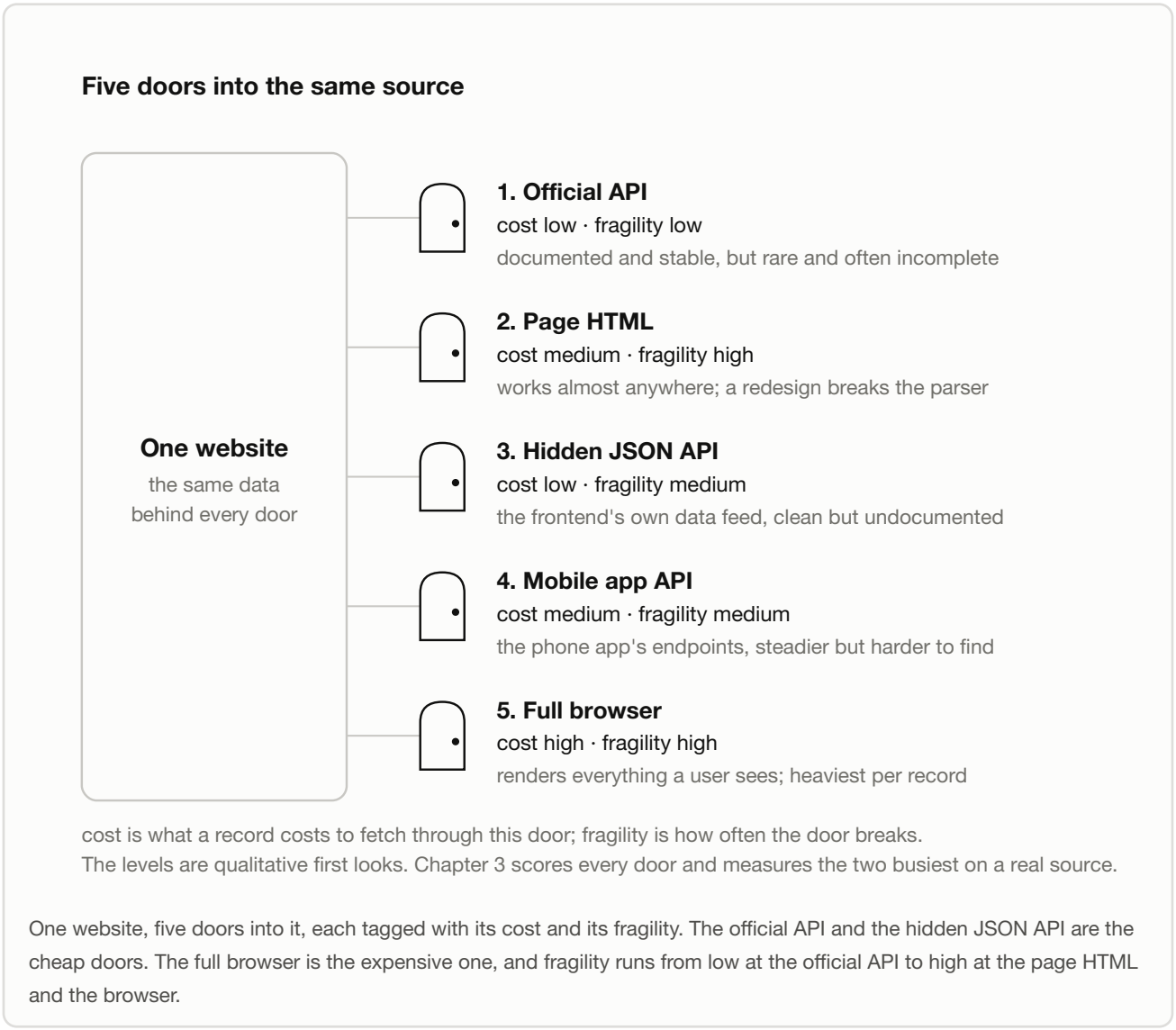
The official API is the front door with a doorbell. The site documents it, hands you a key, and promises a format. When it exists and covers the fields you need, take it and stop looking. It is the cheapest door to operate and the only one whose changes arrive with a deprecation notice, an advance warning that an old version will be shut off. The catch: most sources do not offer one, and many that do exist are rate capped or missing exactly the field you came for.

Everyone finds the page HTML door first: fetch the same page a browser would receive and parse the markup for the price node and the stock badge. It works almost anywhere, which is why the trade gets called scraping, and it is the most fragile item on this list. The site redesigns, your parser starts reading prices out of what is now a photo caption, and no notice arrives, because nobody knows you exist.

My favorite door to show people is the site's own hidden JSON API, because it hides in plain sight. A modern site is an app. The page your browser displays is assembled from data the site's frontend fetches from its own internal endpoints, and anyone who opens the browser's developer tools can watch the catalog arrive as clean, structured JSON before the visuals are painted on. Collect from that feed and you skip the markup entirely: the answers are lighter and cleaner, and a visual redesign cannot touch them. It is also undocumented and owes you nothing. It can move or change shape on any deploy, without notice, forever.

One layer further out sits the mobile app's API, the endpoints the phone app talks to. Finding them takes more work, you inspect the app's traffic instead of a web page, and some are locked behind signatures. In exchange they tend to change more slowly: a company can redeploy its website any afternoon, but breaking those endpoints would break installed apps it cannot force anyone to update.

The full browser is the door that always opens. Run a real browser with software hands, let the page render completely, read what a human would have seen. You pay for that certainty. It is the most expensive door per record by a wide margin, and since you still parse what got rendered, a redesign still breaks you. The costliest door and one of the most fragile at the same time. Sometimes it is the only one that works, and then it is the right choice, made knowingly.



One website, five doors into it, each tagged with its cost and its fragility. The official API and the hidden JSON API are the cheap doors. The full browser is the expensive one, and fragility runs from low at the official API to high at the page HTML and the browser.

That was a first look. Chapter 3 has the fifteen minute audit that finds every door a given source exposes, scores each on cost, fragility, and completeness, and orders them into a waterfall: cheap doors absorb the volume, expensive doors rescue the failures.

And one option stays off the list on purpose. You can skip every door and buy the records ready-made from a data vendor. Buying is a decision not to access at all rather than a sixth access method, and it deserves better than the reflexes it usually gets. Engineers default to build, because building is what they do. Executives default to buy, because vendors return their calls. Both are guesses that sound like decisions. How to buy well is Chapter 14's subject. What I can do here is replace the guess with arithmetic.

Build vs buy: running the numbers

Every chapter of this book makes the same deal with you: a real question gets settled by an experiment you can rerun, and the prose follows the experiment's output rather than my instinct. The question

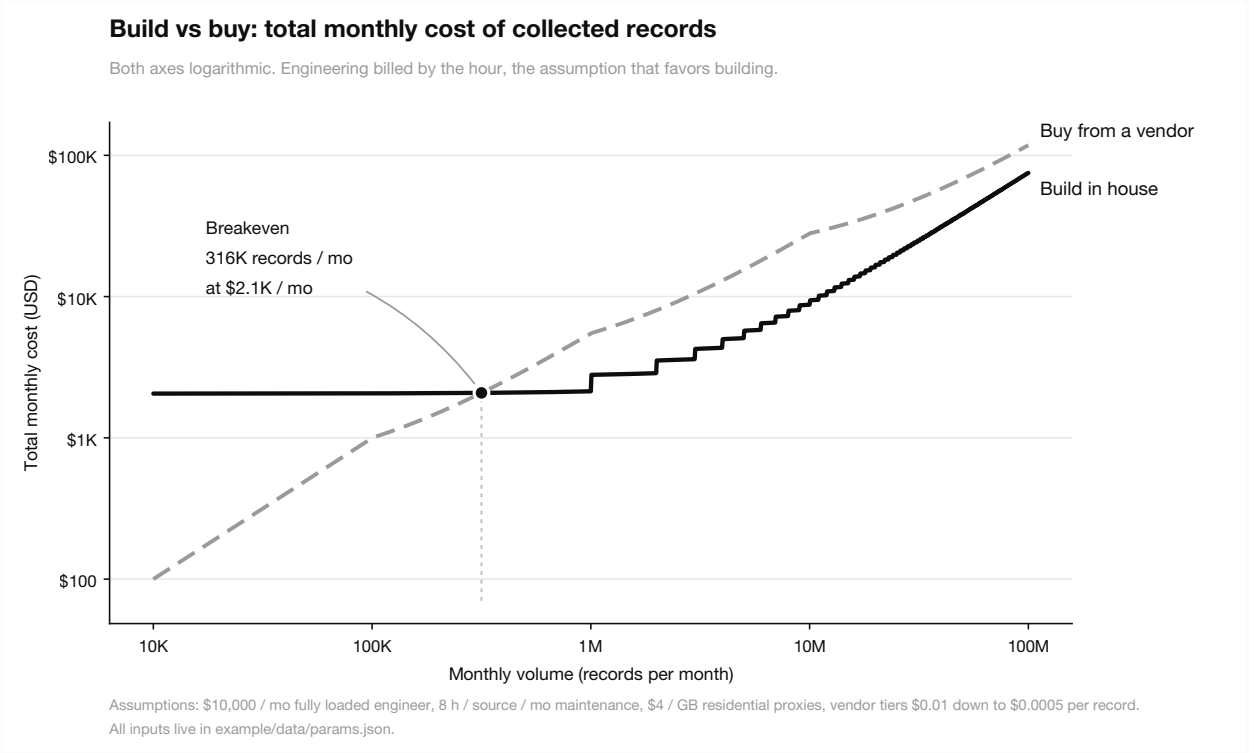
this time: at what monthly volume of collected records does building an in-house collection capability become cheaper than buying the same records from a vendor?

The experiment is a cost model in plain Python. One command runs it, no network needed, and every number in this section comes from the run shipped with this chapter in July 2026. It prices the same job twice.

Building in house adds four costs, and the first one dominates. Engineering time: the model charges 20 hours a month just to own a pipeline at all, deploys, storage, scheduling, monitoring, before a single source is added. Then 8 hours a month of maintenance per source, a source being one website or app you collect from. Then a 60 hour build for each new collector, the program that fetches and extracts one source, spread over a 24 month working life. Hours bill at 62.50 USD, from a fully loaded engineer month of 10,000 USD over 160 hours; fully loaded means what an employee really costs, salary plus taxes, benefits, equipment, and overhead. After the people come residential proxies, intermediary IP addresses from home connections, rented by the gigabyte so that your requests travel the way ordinary traffic does: 4 USD per GB, which under the model's traffic assumptions works out to 0.60 USD per 1,000 requests, about 0.000078 USD per record. Then compute at 10 USD per million requests, and 150 USD a month of fixed small infrastructure. The vendor side is simpler: one price per record on graduated tiers, 0.01 USD per record for the first 100K sliding down to 0.0005 USD beyond 100M, matching public list prices for ready-made web data. Graduated means each tranche bills at its own rate, like income tax brackets.

Every input above is an assumption, set at market typical values, and each one lives in the shipped params.json with a one line justification. Disagree with any of them, edit the file, rerun. The model sweeps demand from 10K to 100M records a month, assumes a typical source yields a million records a month so sources get added as volume grows, and finds where the two cost curves cross. That crossing is the breakeven.

Three volumes tell most of the story. At 100K records a month, building costs 2,064 USD and buying costs 1,000 USD; the vendor wins, comfortably. At 1M the sides have already traded places: 2,136 USD to build against 5,500 USD to buy. At 10M it is 8,756 USD against 28,000 USD. The crossing sits at 316,266 records a month, where both paths cost about 2,081 USD.



Total monthly cost against volume from 10K to 100M records a month. The vendor line wins below the crossing at 316,266 records a month. The in-house line wins above it and climbs in steps, one step for each source added.

Look inside the in-house bill at that crossing, because a budgeting habit dies here. Engineering is 92 percent of the cost. Proxies are 25 USD. Compute is under a dollar. Most of the cost is people, and almost none of it is the script. Teams budget a build by imagining the collector script and pricing the proxy bill, then meet the platform floor, the maintenance, and the fact that all of it is engineer time.

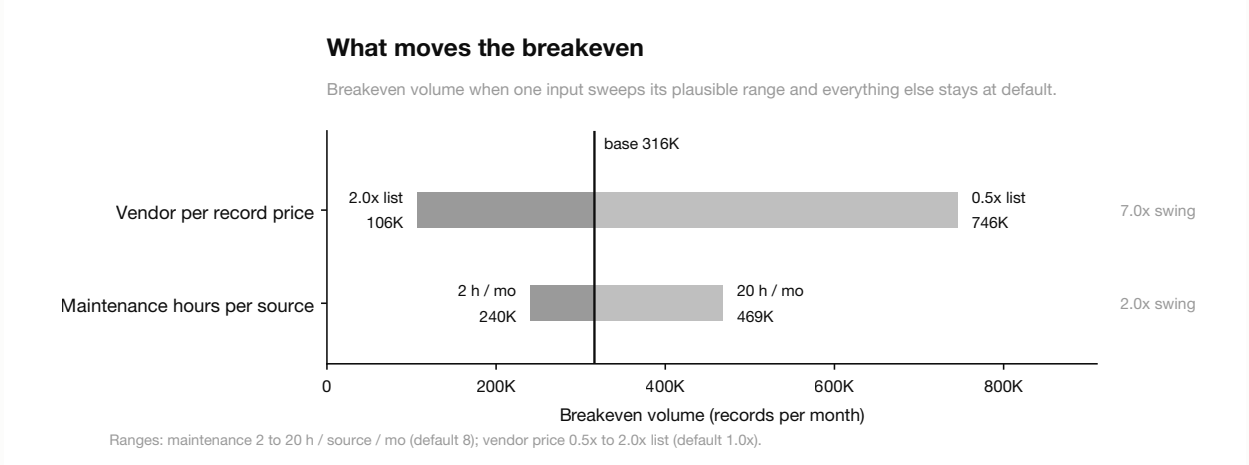
That number hands whoever approves the budget a screening test. When a build proposal reaches your desk, check what fraction of it is people. If the line items are mostly tools, proxy bandwidth, and cloud, with the hours vague, the proposal has not found its own cost yet, because nine tenths of the real bill is missing. Send it back with three questions: whose hours, how many per month per source, and at what fully loaded rate. If those three answers are not most of the total, the total is wrong.

The base case also flatters building in a way I want in the open: it bills engineering by the hour, as if you could hire a fifth of an engineer. Companies hire whole people. Rerun the model in whole heads and the breakeven jumps from 316,266 to 2,953,691 records a month, 9.3x higher, with both sides near 10,384 USD at the crossing. So the answer is a zone. Defensible inputs put the crossing anywhere from about 100K to about 3M records a month, and where it lands inside that zone is mostly an accounting question about how your company pays for people. The model also answers the staffing question that follows the decision: one strong engineer carries about 13 sources, roughly 13M records a month, before collection stops fitting inside someone's week and needs dedicated people.

Before anyone argues, do the multiplication in your own units, because nobody budgets in records. A pricing team's volume is products tracked, times competitor sites, times refreshes a month. Picture a

category manager watching 5,000 products across four competitor sites daily: 600,000 records a month, inside the zone, so the build question is real and turns on how you pay for people. Picture a brand team checking 200 listings weekly across three marketplaces: about 2,600 a month, nowhere near it, so buy the records or give an analyst an afternoon.

Then the sensitivity run, and this is where the model corrected me. Take the two most uncertain inputs and push each across its plausible range while holding the rest still. Sweep maintenance from 2 to 20 hours per source per month and the breakeven moves from 240,057 to 468,683 records a month, a 2.0x swing. Now price the vendor at double list and the crossing drops to 106,469; price it at half list, the low end of the model's range for quotes on the same specification, and the crossing climbs to 746,168. That is a 7.0x swing, and in absolute records the vendor price bar is 2.8 times wider than the maintenance bar.



The tornado chart from the run, one bar per uncertain input. The vendor price bar spans 106,469 to 746,168 records a month. The maintenance bar spans 240,057 to 468,683. The wider bar is the stronger lever on the breakeven.

I wrote a hypothesis in the chapter brief before this model ran, and the book's rule is that it gets judged in public. It said: the breakeven sits much higher than most teams assume, and it is driven by the maintenance tax, not the initial build.

Two parts held, one of them with a condition attached. Maintenance does dominate the build, and not subtly: over a collector's 24 month life the model charges 192 hours of upkeep against the 60 hour build, 3.2 to 1. You pay the build once, while maintenance comes back every month like rent that the sites you collect from keep raising without asking. "Higher than most teams assume" held under the accounting that actually applies to most companies: hiring whole people pushes the crossing out to 2,953,691 records a month, far beyond any casual guess. Billed by the hour it lands at 316,266, higher than a first instinct but not dramatically so. The claim survives with that condition stated.

The third part missed. Of the model's two uncertain inputs, the vendor's per record price moves the breakeven more than the maintenance tax does, 7.0x against 2.0x. Maintenance rules the in-house side, exactly as predicted, but the breakeven moves further with the number on the other side of the

table. The practical order of operations inverts: get real vendor quotes first, then audit your own maintenance load. I would have told you the reverse, calmly and with authority, if I had not run the model. That miss is why every chapter here runs an experiment with the hypothesis written down beforehand, so the text cannot drift into whatever I already believed.

Cash that out now, because the fix is one email, sent three times. One quote is not enough to know the price: the model's range for quotes on the same specification runs from half of list to double it. Say a team needs 500,000 records a month: at list prices, billed by the hour, the crossing sits at 316,266 and building is on the table, while at half list it climbs to 746,168 and they should buy. The same team and the same need, flipped by paper they had not requested. So put one written specification, sources, fields, refresh rate, delivery format, in front of at least three vendors before any build gets approved.

These numbers have two edges. At the very top of the sweep, the vendor's best tier at 0.0005 USD per record undercuts the model's in-house marginal cost of 0.000736 USD per record, so if that tier price held, the vendor would retake the lead near 283M records a month; at extreme scale the question reopens rather than settles. And sometimes the answer is neither. Below the bottom of the zone, where a vendor's monthly minimums and setup fees that real contracts often carry, or your own platform floor, would dwarf the value of the data, the right tool is an analyst with a spreadsheet and an afternoon a week, or an official API's free tier, and no pipeline at all. The write-up shipped with the model lists further limits, including how the answer shifts when your sources yield fewer records than the model's million a month.

The maturity ladder

Nobody plans a data platform on day one. I have never seen a team that needed to. What happens instead is a ladder with four rungs, each one built out of the failure of the rung below, and knowing it in advance saves real money, because each rung fails in a predictable way.

It starts as a script. Somebody technical writes it in an afternoon, it fetches a modest pile of pages, and the numbers land in a spreadsheet that makes one meeting noticeably smarter. The script runs when its author remembers. Then the author takes a vacation, and the smartest chart in the company quietly freezes, which is how everyone discovers they had started depending on it.

So it becomes a scheduled collector: the same logic moved onto a server, running nightly, writing somewhere shared. This rung feels like victory. It is also where silent failure starts. The source redesigns, the parser begins returning empty fields, and the schedule keeps reporting green, because what is being checked is that the run happened, not that the data arrived. The gap between "it ran" and "it worked" is wide, and several later chapters live inside it.

The pipeline earns its third rung when somebody starts watching it: fill rates, today's volume compared against yesterday's, retries with growing pauses, an alert that wakes a person. Breakage stops being silent and becomes ordinary morning work. This rung breaks on the payroll arithmetic from the model above. Sources accumulate. Each one carries its 8 hours a month of rent. Somewhere

near the 13 sources one engineer can hold, collection stops being a side duty and becomes a team, with a budget and somebody accountable for data quality by name.

The top rung is a data product: the schema becomes a promise to the people who consume the data, delivery gets verified at the destination, and a quality gate catches a bad batch before anyone downstream is hurt by it. Records nobody inside the company has to think about. You have become, internally, the vendor this chapter's model priced, and their per record price was paying for this same ladder, spread across every customer they have.

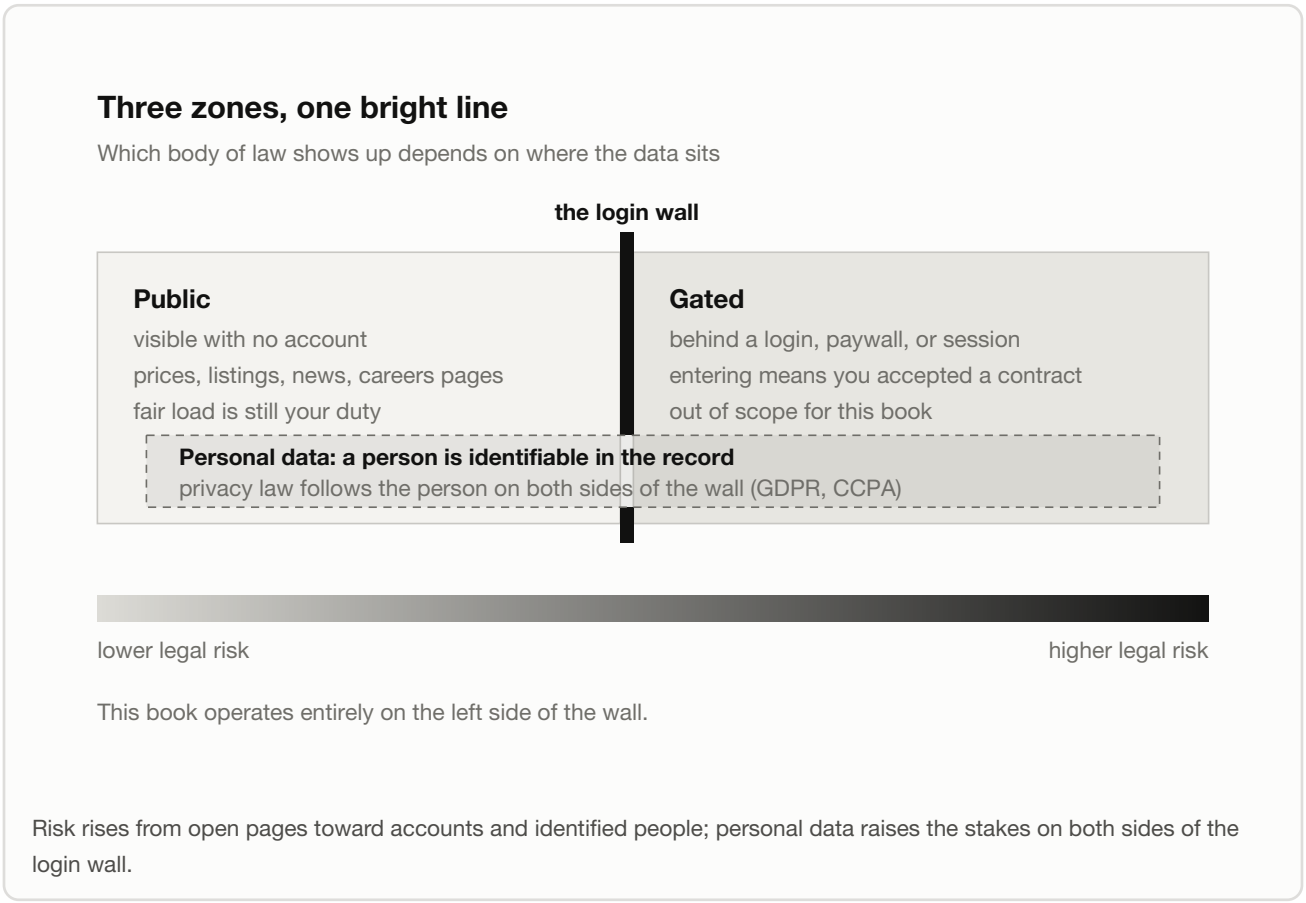
No cost model survives collecting data you had no right to touch. So before any of this, one question: what is fair to take from the public web, at what load, and under which rules, written and unwritten.

Chapter 2. Legality, Ethics, and Fair Use

Every founder in this industry hears the same question, usually within five minutes of explaining what the company does. Is this even legal? I have answered it on sales calls and at least once over a family dinner. The answer is conditional. It hangs on what you collect, how you collect it, and how much of it you take. I will walk those lines in the order a lawyer would draw them, then add the part lawyers rarely cover: how to be a decent guest on infrastructure that someone else pays for.

One thing before we start. I am not a lawyer, and none of this is legal advice, only how a practitioner works these lines after years of building collection systems that paying customers depend on. When a real dispute or a real contract is on the table, hire counsel. What I can give you is the map counsel will sketch in your first meeting, and the operating habits that keep the second meeting short. And a note on the title: fair use in these pages means using the open web fairly, load and manners. Copyright's formal fair use doctrine is a usage-rights question, which belongs to the buyer's side of the table in Chapter 14.

The map has three zones. Public pages, which anyone can see without an account. Gated content, which sits behind a login or a paywall. And personal data, which cuts across both, because a human being can be identifiable on either side of the wall. Most legal questions in this field start with which of those three you are in.



Risk rises from open pages toward accounts and identified people; personal data raises the stakes on both sides of the login wall.

The login wall is the brightest line

Public means visible to anyone with a browser and no account. A product page, a price, a news article outside the paywall, a company's careers listing. Gated means the site asked for something before showing the content: an account or a payment. That wall is the brightest legal line in this field, far more than the tool you fetch with.

The reason lives in a United States law from 1986 called the Computer Fraud and Abuse Act, the CFAA, written to punish breaking into computers. For two decades nobody knew for certain whether fetching a public web page that the site did not want you to fetch counted as "access without authorization" under that law. Being wrong meant a federal claim, potentially a criminal one, hanging over an ordinary HTTP request.

hiQ Labs v. LinkedIn is the case that tested it, and you should know the whole story, because half of it gets quoted everywhere and the other half gets forgotten.

hiQ was a small analytics company that scraped public LinkedIn profiles and sold employers predictions about which of their employees might quit. In 2017 LinkedIn sent hiQ a cease-and-desist letter demanding it stop, and began blocking its traffic. hiQ sued, and a federal district court in California ordered LinkedIn to stand down while the case ran. In 2019 the Ninth Circuit Court of Appeals upheld that order and said the words this industry has been quoting ever since: the CFAA's "without authorization" concept likely does not reach data that is public in the first place. The Supreme Court sent the case back in 2021 after Van Buren v. United States, its own decision reading the CFAA as a law about gates, up or down. In 2022 the Ninth Circuit reaffirmed. The CFAA is about gates, and where no gate is up the Ninth Circuit saw nothing for it to reach.

That is the half everyone quotes. The case did not end there. Back in the district court, the case turned in November 2022 on a much older kind of claim: contract. The judge found on partial summary judgment, a ruling without a full trial, that hiQ had breached LinkedIn's User Agreement. hiQ held LinkedIn accounts, which meant it had accepted that agreement, which bans scraping and fake profiles, and hiQ had done both; what remained for trial were hiQ's fallback defenses and the damages. It never got there. In December 2022 the parties settled. hiQ accepted a judgment of $500,000 and a permanent injunction barring it from ever touching LinkedIn again. By then the company was, in every practical sense, already gone; years of litigation had drained its funding and its customers long before the final order.

So when someone tells you that hiQ established that scraping public data is legal, they are wrong twice. The Ninth Circuit held something narrower: collecting public data is likely not federal hacking. And the company that won that point still lost on contract, and did not survive to enjoy its own precedent.

Three working rules fall out of that story, and they carry different weight. The heaviest: never collect through accounts. An account is a signed contract. Courts have been reluctant to enforce terms buried behind a footer link against people who never saw them, the Ninth Circuit said as much in Nguyen v.

Barnes & Noble in 2014, but nobody has that defense after clicking "I agree," and hiQ's case collapsed on exactly this point. Second, treat a cease-and-desist letter as a legal event rather than an inconvenience. The CFAA piece of that is narrower than folklore has it: in Facebook v. Power Ventures in 2016 the Ninth Circuit did find a company liable for pressing on after Facebook explicitly revoked its access, but the data there sat behind Facebook's logins, and the hiQ court later distinguished public pages on exactly that ground. What a cease-and-desist changes, on any data, is everything else. It revokes whatever welcome you could once assume. From that day forward the sender's contract and trespass claims are stronger, and the next decisions belong to counsel rather than the roadmap. Third, and least discussed: being right can be ruinously expensive. hiQ won its argument and lost its company. The cheapest legal strategy in this business is to collect in a way nobody feels moved to sue over. The rest of this chapter is how.

If you buy data rather than collect it, the hiQ ending is also your vetting script, because you just watched what legal cost does to a vendor's product before any verdict arrives. A vendor's legal exposure is your supply risk. Ask before you sign: do any accounts touch the collection path for my sources, has any of my sources sent a cease-and-desist, and what happens to my deliveries if one arrives mid-contract? A vendor that has planned for a dispute answers those without checking with anyone.

Personal data is different

Everything above was about property and contracts. Personal data answers to a third body of law, and that law follows the person even when the page is public.

A price is not personal, and neither is a product title. The moment a record identifies a human being, a name, an email address, a face, a profile handle, you have left the world of property arguments, and the fact that the data was public stops helping you. The strictest regime is the GDPR, the European Union's General Data Protection Regulation, applied since 2018. It covers any information relating to an identifiable person, and it reaches across borders: wherever your servers sit, monitoring people in Europe or offering them services brings you inside it, and collecting profiles at scale is monitoring. Publishing something does not waive its protection. Under the GDPR you need a documented lawful basis to collect personal data at all. People can demand to see what you hold and, under its Article 17, demand deletion. Article 14 goes further than most engineers expect: when you collect someone's data from a source other than the person, you are generally obliged to tell them. Fines run to 4% of worldwide revenue or 20 million euros, whichever is higher.

Clearview AI found out what enforcement looks like. It built a face-search product on billions of images scraped from public websites. In September 2024 the Dutch data protection authority fined it 30.5 million euros and ordered the collection stopped, one of several European regulators to reach the same conclusion about the same database. Every one of those images was public, and to the regulators that did not matter.

California's CCPA, the California Consumer Privacy Act, in force since 2020, takes a different line: it carves "publicly available information" out of its definition of personal information, so data lawfully published in government records, or made public by the person themselves, sits outside much of the law. Do not read that as a green light. The carve-out is narrower than it sounds, other US states have passed their own variants, and the moment your dataset includes people in Europe you are back under the GDPR.

My translation of all this into practice is short. When the job allows it, collect the field and leave the person out. If the deliverable is prices and catalog structure, strip names, handles, avatars, and emails at parse time; a field you did not store is one you never have to defend or delete. When the job genuinely is about people, recruiting intelligence, say, or review authorship, then price the compliance work into the engagement: a lawful basis written down and a deletion path that actually works, reviewed by someone who has seen the GDPR up close. I have watched teams file that under paperwork for later. Later is when the access request arrives.

What the top of the web actually writes in robots.txt

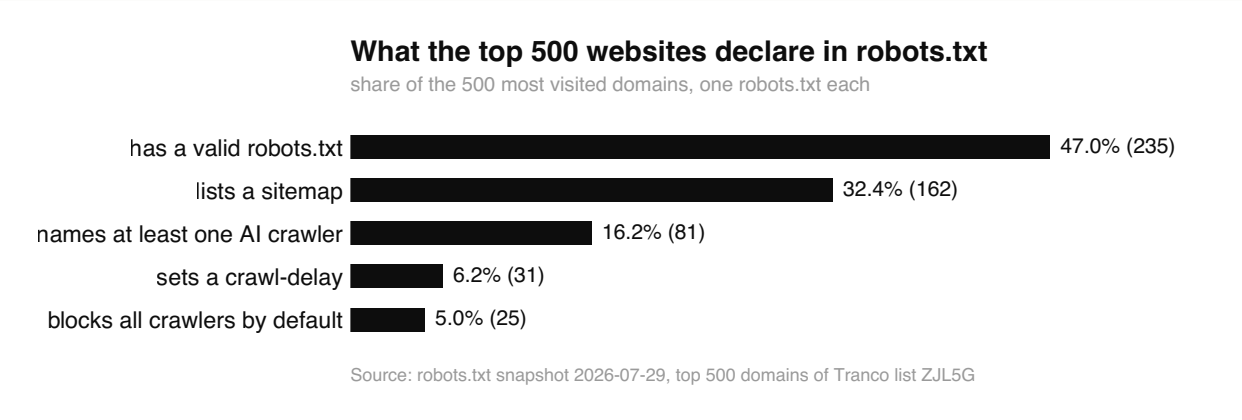
There is a file that answers half the ethics questions in this chapter before you ask them, and most people have never read one. robots.txt is a plain text file at a fixed address, example.com/robots.txt, where a site declares which automated visitors, called crawlers, may fetch which paths, and how fast. Martijn Koster proposed the convention in 1994; it stayed informal custom for twenty-eight years until the IETF wrote it down as RFC 9309 in 2022. For a working collector the RFC's main value is a citable standard: the name to give when a customer or their counsel asks which rules your crawler follows. Nothing in it can stop a request, and United States courts have not treated it as a binding contract. It is a preference, stated in public, in a format built for machines to read. I respect it anyway, on every collector, and not out of fear. It is the one place where a site tells you, in advance and unambiguously, what kind of guest it wants. Ignoring it buys you little and costs you the ability to say, later and truthfully, that you followed every preference the site published.

For years I repeated the industry folklore about these files, that hardly any site blocks everything and most just fence off a few paths, without ever counting. This chapter's experiment counts.

The question: what do the most visited domains on the web actually declare in robots.txt? In particular, how many block every crawler by default, and which crawlers get singled out by name? The method fits in three sentences. Take the top 500 domains from Tranco, a public research ranking of popular domains (list ZJL5G, so anyone can pull the identical sample). Fetch one file per domain, politely: a user agent that names us truthfully (the user agent is the label a program sends identifying itself), a ten second timeout, at most one retry over https with a single fallback to plain http, and every failure recorded as a result rather than retried into submission. The snapshot, dated 2026-07-29, ships with the chapter, so `python run.py` re-derives the census offline, and the remaining details in this section read straight out of the shipped manifest and snapshot files; the entire run moved 8.7 MB, less than one news homepage.

146 of the 500 domains gave no usable answer, and that needs saying before any percentage does, because the census never counts an unreachable site as evidence of anything. Most of that bucket is infrastructure rather than websites: akamai.net, cloudfront.net, root-servers.net, names that serve software rather than people and mostly came back as gateway errors, plus 24 domains that never answered at all. Tranco builds its ranking from several public popularity signals, browser visits among them but also raw DNS lookup volume, so its top 500 is full of such names, and a dead end there is the expected answer. Among the 354 domains that answered like websites, 235 publish a valid robots.txt, 66.4%. Across the full sample of 500, that is 47.0%. Another 83 answered but had no file to give, mostly plain 404s (44 of them) and 403s (32), and 36 more returned something at the robots address that was not a robots.txt at all.

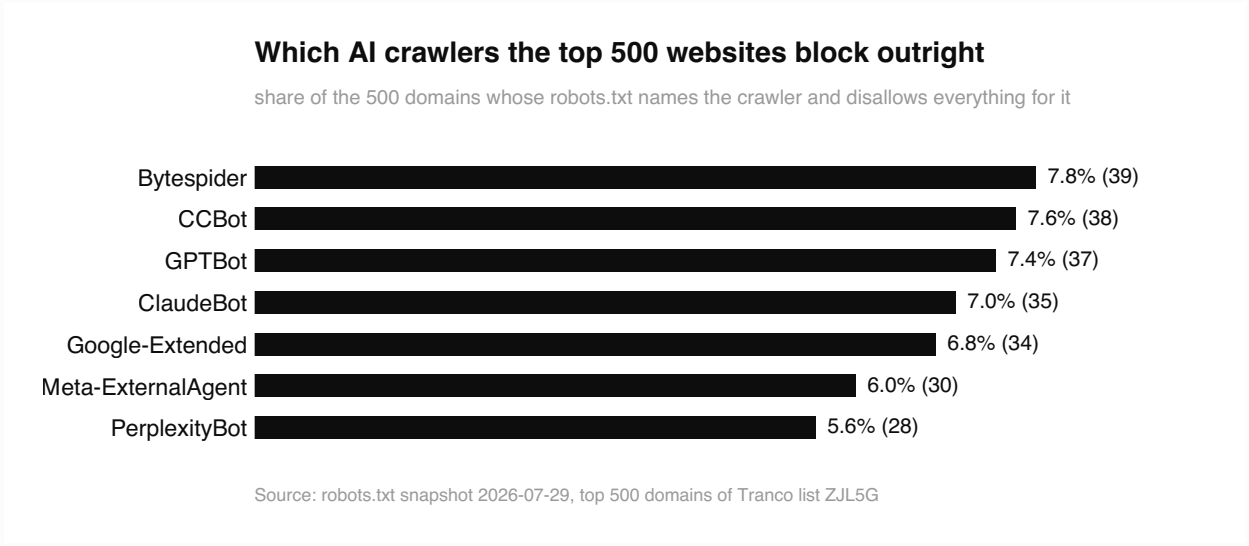
Walls turn out to be the exception. Only 25 of the 500 domains block all crawlers by default, 5.0% of the sample and 10.6% of the valid files, and 20 of those 25 immediately name specific crawlers and let them back in, so even most walls come with a guest list. Meanwhile 162 of the 500 use the file to point crawlers at a sitemap, which is nothing more than a machine-readable directory of the site's pages; that is 68.9% of all valid files, which means the most common thing the top of the web does with robots.txt is invite orderly crawling. A smaller group, 31 of 500, sets a crawl-delay, asking crawlers to pause between fetches, with a median of 4 seconds and a range of 0 to 20: github.com asks for 1 second, wikipedia.org for 5, europa.eu for 10.



Walls are the exception at the top of the web: 235 of 500 domains publish a valid robots.txt, 162 use it to hand crawlers a sitemap, and only 25 block everyone by default.

Before the run, I had written down a hypothesis: blanket blocking would be rare, most sites would express preferences rather than walls, and the recent wave of AI-crawler blocking might be far larger than the folklore admits. The data supports all three, and adds a twist I did not predict: the blockers are the biggest consumer platforms on earth. The 25 default-deny files belong to facebook.com, instagram.com, x.com, netflix.com, reddit.com, wsj.com, reuters.com, trustpilot.com, and their peers. So the top of the web is mostly open, and the few walls stand in front of its most valuable rooms.

The AI wave is real and measurable. 81 of the 500 domains, one valid file in three (34.5%), now name at least one of the seven AI crawlers the census tracks. Naming is usually a prelude to blocking: across those seven crawlers the snapshot holds 362 naming events, and 241 of them are full blocks, 66.6%. GPTBot, OpenAI's crawler, is the most named, by 60 sites, of which 37 block it fully. Bytespider, ByteDance's crawler, is the most blocked outright, by 39 of the 49 sites that name it; Meta-ExternalAgent runs at the highest rate of all, blocked by 30 of its 37 namers. CCBot, the Common Crawl crawler whose archive fed a generation of AI models, is blocked by 38 of 50. The sites doing the blocking include nytimes.com, amazon.com, cnn.com, ebay.com, and yahoo.com. When a top site bothers to write an AI crawler's name into robots.txt, it blocks it two times out of three.



Naming is usually a prelude to blocking: across the seven AI crawlers, 241 of the census's 362 naming events are full blocks.

The AI rows of these files are changing fast, so a collector that read robots.txt once at build time is working from a stale answer. Re-read it on a schedule, the way you would re-check a price.

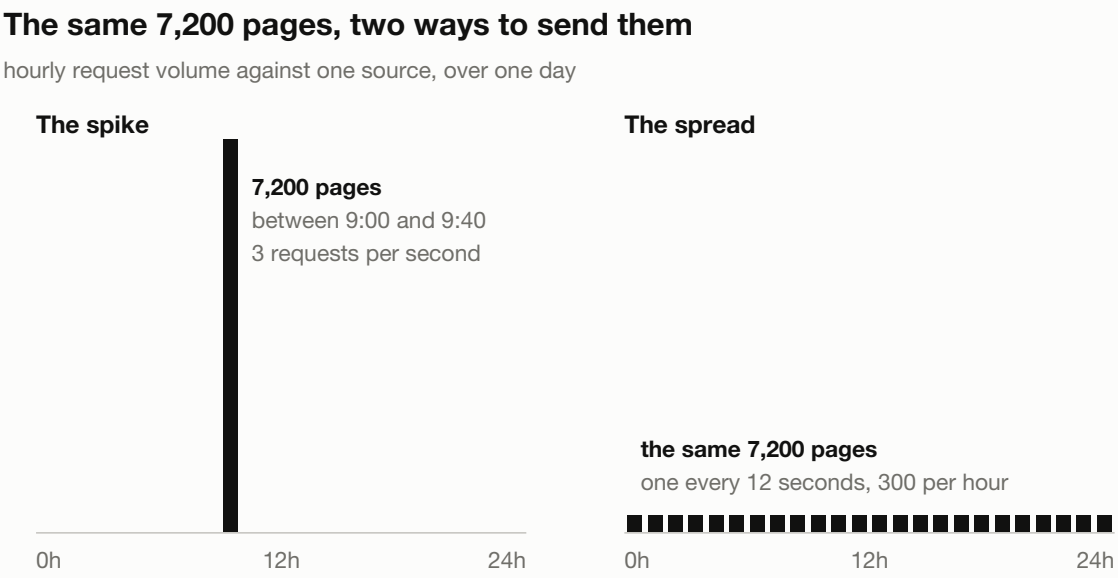
If you sell data that could train a model, or buy data that will, this wave changes what provenance is worth: the record of where each row came from and what the source was asking of crawlers at the time. The census above shows why: with a third of valid files now naming AI crawlers, the same page can welcome collection one quarter and refuse it the next. Keep the robots.txt you read, dated, alongside each batch it governed. It is a dated record of the preferences the site published on the day of collection, and it is the document a serious training-data buyer will ask for. A vendor who cannot produce it cannot show you what the source was asking on the day your rows were collected.

Fair load, or how not to become someone's incident

Fair load is how you behave where the law is silent, and the law is silent about most of what a collector does all day. Every request your collector sends consumes compute and bandwidth the site provisioned for its human visitors, plus, past a certain point, the attention of whoever is on call. Push

hard enough and the law stops being silent: a federal court in California treated crawler load on eBay's servers as trespass to chattels, interference with someone else's property, back in 2000, in eBay v. Bidder's Edge, and the hiQ consent judgment two decades later still carried the same tort. Load was what sites sued over long before anyone trained a model. But the deeper argument is commercial rather than legal. A collector that degrades its source gets noticed, then blocked, and a blocked collector delivers nothing.

In practice it comes down to four habits. Choose a request-rate ceiling on purpose and write it into the collector's configuration, instead of inheriting whatever your thread pool defaults to; where the site declared a crawl-delay, that number wins, and where it declared nothing, silence is not an invitation. Spread the volume: the same daily total can arrive as a spike that looks like an attack or as a background hum nobody notices. Picture 7,200 pages collected in a day. Sent between 9:00 and 9:40, that is 3 requests per second sustained for 40 minutes, exactly the shape a monitoring dashboard flags. Spread across the day, it is one request every 12 seconds, 300 pages per hour, a rounding error in a popular site's traffic. Schedule against the site's quiet hours instead of your own business hours. And never fetch what you already have: cache raw responses and reuse them (Chapter 7 makes storing them the default anyway), and when the site answers 429, the status code meaning "too many requests," or starts erroring, slow down automatically instead of pressing. Chapter 6 turns that back-off discipline into code.



The same 7,200 pages reach the server as a 40 minute spike or as one request every 12 seconds; only one of these gets a collector noticed and blocked.

The test I give engineers is one question: if the site's operations team could see your traffic labeled with your company's name, would anything about it embarrass you? If yes, fix it before someone else finds it.

A checklist you can hand to a customer

I keep this on one page because its real job is to be shown to other people: a customer's procurement team, or a prospect who opens the call with the legality question from the top of this chapter. Buying data, it reverses: send these eleven lines to every vendor you are comparing and ask for written confirmation of each, and where a vendor hesitates, line 2 on accounts or line 4 on personal data, is where your risk lives. It also works internally. Before a new collection goes live, every line gets a yes from a person willing to put their name next to it.

1. Everything we plan to fetch is visible without an account or a payment. No exceptions, no borrowed sessions.
2. We accepted no terms of service to reach the data, and no account of ours touches the collection path.
3. We have received no cease-and-desist letter and no block aimed at us specifically from this source. If either arrives, collection pauses and counsel decides what happens next, not the roadmap.
4. No field in the output identifies a person. If the job truly is about people, the lawful basis is written down, unneeded personal fields are dropped at parse time, and a deletion request has a tested path.
5. We read the site's robots.txt before the first fetch, our target paths respect it, and any declared crawl-delay is honored.
6. Our request rate has a deliberate ceiling, written in configuration, with the day's volume spread thin instead of bunched into spikes.
7. Nothing gets fetched twice; a raw response, once collected, is kept and reused.
8. On "too many requests" and server errors, the collector slows itself down without waiting for a human.
9. Our traffic identifies itself honestly and does not impersonate a human visitor to get around a site's defenses.
10. The customer's rights to use the data are covered by contract, because collecting a record lawfully does not by itself grant the right to resell it or train a model on it. That second question belongs to the buyer's side of the table, and Chapter 14 handles it there.
11. We could explain this collection, in plain language, to the site being collected, and we would be comfortable doing it.

If you can defend a yes on item 11, the other ten are usually already true.

With the lines drawn, the next problem is mechanical: any source worth collecting has several doors into it, the official API, the page itself, the site's own hidden data feeds, and a few others less obvious, and the cheapest one is rarely the one facing the street.

PART II. GETTING THE DATA

Chapter 3. Choosing the Access Method

A team I worked with was collecting a retail catalog the hard way. Every night a fleet of real browsers opened product pages one at a time, waited for the scripts to finish painting the layout, and read the price back out of the finished page. It worked most nights. It also cost more in machine time than the rest of their data platform put together. The morning it broke, I opened one of those same product pages myself with the browser's developer tools showing, and watched the page ask its own server for the catalog: one request, clean structured data coming back, every field they were scraping and several they did not know existed. That door had been there the whole time. In public. Standing open. Cheaper and faster than the one they were using.

Nobody on that team was careless. They did what almost everyone does, which is to look at a website and see a website. What is actually in front of you is a building with several doors, and the one facing the street is rarely the cheapest way in. Chapter 1 introduced the five: the official API, the page HTML, the site's own hidden JSON API, the mobile app's API, and a full browser. The work is to find every door a given source exposes, score them, and then arrange them so the cheap ones carry the volume and the expensive ones only rescue what falls through.

The fifteen minute audit

Before a line of collection code gets written for a new source, I spend fifteen minutes with the site open and the browser's developer tools showing. Developer tools are built into every modern browser, usually behind the F12 key or an Inspect menu item, and the tab that matters here is Network, which lists every request a page makes while it loads. Fifteen minutes is not a figure of speech. This is where the time goes.

Two minutes on the paperwork. Look for published API documentation, read robots.txt, and check for a sitemap or an RSS feed. Chapter 2 covered robots.txt as a statement of preferences you are going to honor. It is also reconnaissance: the census in that chapter found that 162 of the top 500 domains use the file to advertise a sitemap, which is a machine-readable directory of the site's pages and the fastest way to learn what exists without crawling to find out.

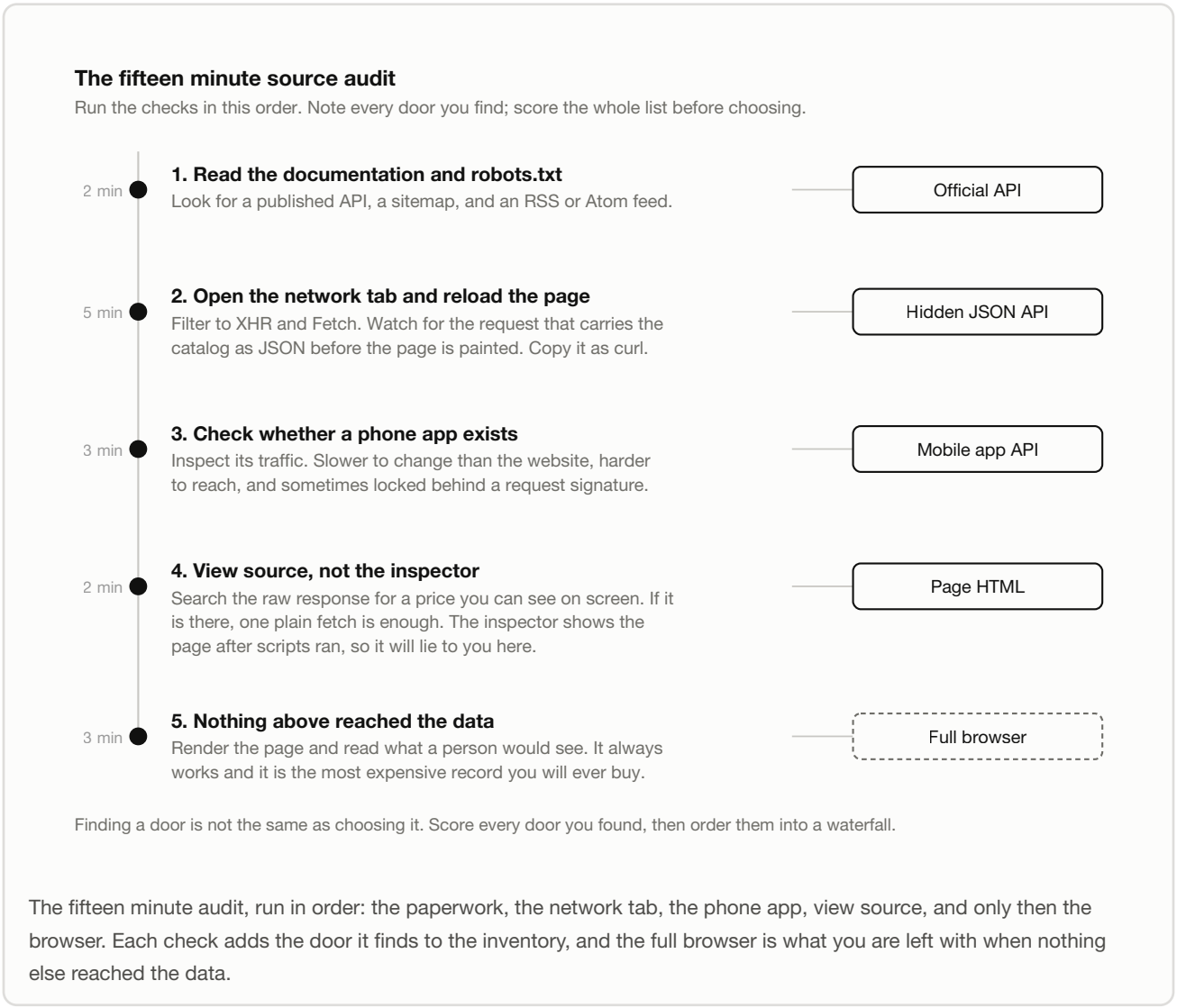
Five minutes in the network tab. Reload the page with the tab open and filter to XHR and Fetch, the two labels browsers use for requests that JavaScript makes in the background rather than for images and stylesheets. Then use the site like a person: click a category, turn a page, sort by price. Watch for the request whose response carries the values you came for. When you find it, right click and choose Copy as cURL, which hands you the exact request as a command you can paste into a terminal. If it answers there, outside the browser, you have found a candidate for the hidden JSON API. Copy as cURL carries the browser's cookies and tokens along with it, so the real test is whether the request still answers without that session; if it does, most of the job is done.

Three minutes on the phone. Does the company ship a mobile app? Its endpoints are usually a separate door, often steadier than the website, sometimes carrying fields the web version never shows. Reaching it means inspecting the app's traffic rather than a page, and some apps sign their requests in ways that make this a project rather than an afternoon. Note it and move on; you are taking an inventory, not building anything yet.

Two minutes in view source. Not the inspector. This distinction costs teams weeks, so it is worth being exact. The inspector shows the page as it stands after every script has run. View source shows the document as the server sent it, before any script ran, and the Network tab's document entry shows the exact bytes of the load you just watched, which is the purer version of the same check. Search the raw response for a price you can see on screen. If it is there, one plain fetch gets you the data and you never need to render anything. If the raw response is a near empty shell and the numbers only appear after the scripts run, the data is arriving separately, which is your cue to go back to the network tab and find it.

Three minutes writing it down. List the doors you found, note which fields each one exposes, and mark the ones that need real work to reach. That list is the input to every decision that follows.

It is also the cheapest control I have ever put on a collection budget. When a build estimate or a vendor proposal reaches me without a door named, I send it back with this list as the ask, because an unnamed door defaults in practice to the browser, the one that always works, and the waterfall arithmetic later in this chapter prices that default at nearly thirty times the blended alternative.



The fifteen minute audit, run in order: the paperwork, the network tab, the phone app, view source, and only then the browser. Each check adds the door it finds to the inventory, and the full browser is what you are left with when nothing else reached the data.

You leave the audit holding a list, and the list is not yet a choice. Two sources with identical doors can still deserve different answers, because the doors are not equally good at the thing you actually need.

Scoring what you found

Three axes decide it. Cost per record is what one delivered row costs to fetch through this door, counting bandwidth and machine time. Fragility is how often the door breaks on you, and more precisely, how often it breaks without telling you. Completeness is which fields the door exposes, which varies more than people expect: an official API can be missing the one number you came for, while the site's internal feed often carries stock levels and identifiers the visible page never prints. A fourth axis, effort to build, matters for planning but not for the choice, because it is paid once while the other three are paid on every record forever.

Scoring the five doors

Five marks is the most of that quality. Read cost, fragility and effort as costs to pay, and completeness as the only one you want more of.

	Cost per record less is better	Fragility less is better	Completeness more is better	Effort to build less is better
1. Official API documented, rare, often incomplete	●○○○○	●○○○○	●●●○○	●○○○○
2. Page HTML works anywhere, breaks on every redesign	●●●○○	●●●●●	●●●●○	●●○○○
3. Hidden JSON API the frontend's own feed, undocumented	●○○○○	●●●○○	●●●●●	●●●○○
4. Mobile app API steadier, harder to reach	●●●○○	●●●○○	●●●●○	●●●●●
5. Full browser always opens, costs the most	●●●●●	●●●●●	●●●●●	●●●○○

These marks are judgement from experience, and they are where the audit starts, not where it ends. This chapter measures two doors, HTML and JSON against each other on one real source, and two of the marks above do not survive contact with the data.

The five doors scored on cost per record, fragility, completeness, and effort to build. The official API and the hidden JSON API are the cheap doors, the full browser is the expensive one, and the page HTML is the most fragile.

Those marks are my judgement after years of running collectors, which makes them a starting point and not evidence. So the rest of this chapter tests two of them.

Two doors, measured on one source

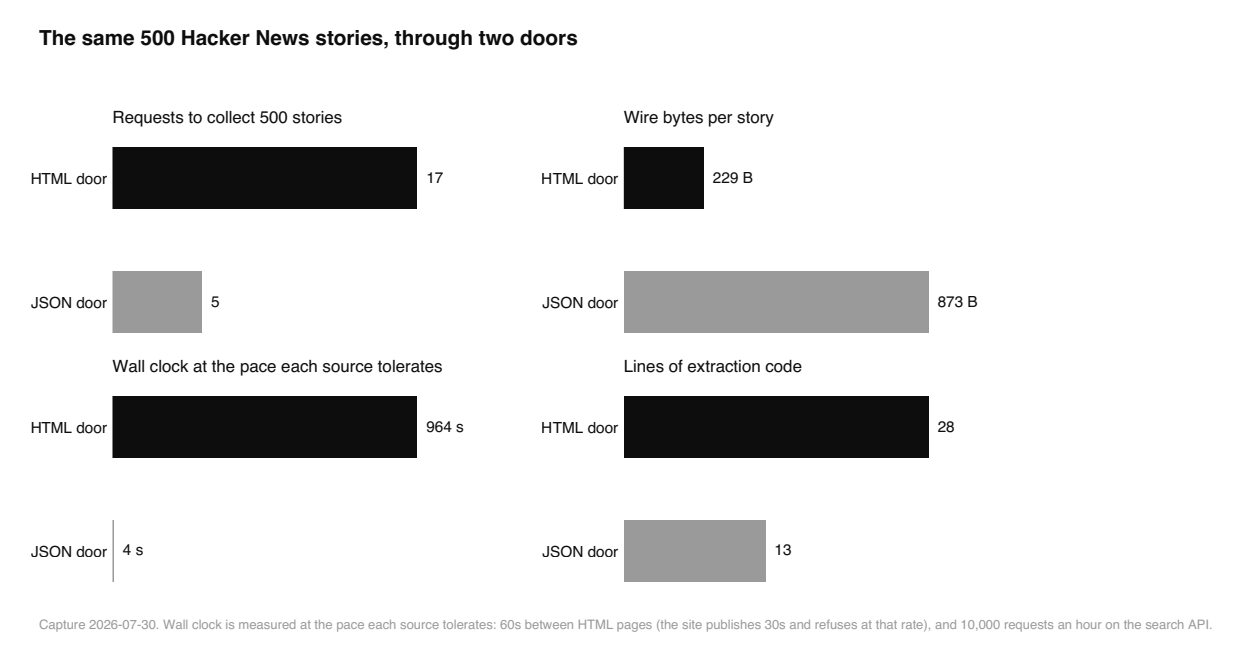
Hacker News publishes its listing of new stories two ways at once, which makes it a rare fair test bench. The HTML door is the page everyone knows, 30 stories at a time behind a More link. The JSON door is the site's documented search API, hosted on a separate domain, list shaped, fetched 100 stories per request in this run (it allows more), and returning nearly the same newest first walk. One note before the numbers: in this book's own vocabulary that makes it an official API rather than an undocumented frontend feed, and I chose it because it is list shaped the way a hidden feed would be. The redesign lesson below transfers to any JSON door; the pace and latency figures belong to this API's own limits. Hacker News also offers an item by item API that would need one request per story; using it here would rig the test against JSON, so the experiment does not. The job: collect the newest 500 stories through each door, extract the same seven fields from both, and record what each door charged. One page at a time, no concurrency, a truthful User-Agent, and every number below from the capture of July 30, 2026 that ships with this chapter.

The first attempt did not survive. Chapter 2's rule is that a declared crawl-delay wins, and Hacker News declares 30 seconds, so the collector waited exactly 30 seconds between pages. On the sixth page the site answered with status 429, too many requests, and a one word body: Sorry. I checked the obvious suspect first, the shared exit address, and cleared it; the same URL answered fine from the

same address after a rest. Then I probed again at the same pace on a fresh walk and was refused on the fifth page, after 120 stories. Both refusals at the site's own published pace. Doubled to 60 seconds, the walk completed all 17 pages without complaint. The rate a site publishes and the rate it enforces are two different numbers, and you only learn the second one by being refused. The probe that reproduces this ships with the example.

With the pace settled, the doors separate fast. The JSON door needed 5 requests where the HTML door needed 17, and at the pace each source tolerates, that difference compounds into the whole story: 4.3 seconds against 16.1 minutes for the same 500 stories, 223 times faster. One asymmetry inside that comparison: the HTML pace was found by being refused, while the JSON pace is the API's documented ceiling, never probed to a refusal. Extraction took 13 lines of code against 28, resting on 8 structural assumptions against 12; the twelve HTML couplings are things a restyle can move, while the eight JSON couplings are field names that move only when the schema does.

Multiply that speed out before you promise anyone freshness. Pages needed times the pace the source tolerates is your delivery window, and the pace that counts is the enforced one the probe found, not the published one. Through this HTML door at 60 seconds a page, 50,000 records is about 27 hours of walking, which breaks a daily delivery promise before a line of code is written. Through the JSON door the same volume fits in minutes. I run this multiplication before a refresh commitment goes into any contract, mine or a vendor's, because a promise the window cannot hold fails on schedule, every day.



Four panels from the run: 17 requests against 5, 229 wire bytes per story against 873, 964 seconds against 4.3 at tolerated pace, 28 lines of extraction code against 13.

Now the parts my own brief got wrong. I wrote the hypothesis down before the run, so it gets judged by the result. I predicted the JSON door would answer several times faster per request. Backwards:

the HTML door's median response was 176 milliseconds against 440, 2.5 times faster, a plain page cache beating a search engine doing real work per query. The 223x above is rate limits and page sizes, not server speed. I also predicted the JSON would be an order of magnitude lighter. Backwards again, on the wire: the HTML door moved 229 bytes per story and the JSON door 873, because Hacker News compresses its pages to a sixth of their size while the API answered uncompressed in this capture, compression requested and not applied, and pads every hit with search metadata. Only after decompression is the JSON lighter, 1.7 times, nowhere near an order of magnitude. If your proxies bill by the gigabyte, the door that is easier to parse can be the one that costs four times more bandwidth. I have priced collections for years and I would have gotten both of these wrong from the armchair.

The third prediction was that a layout redesign cannot break the JSON door, and the experiment backs it with a number I did not expect to be so lopsided. The test breaks both parsers on purpose: six restyle mutations, renamed CSS classes and moved attributes, the changes a redesign makes; then four data model mutations, renamed fields, the changes a schema revision makes. The restyle broke the HTML parser five ways out of six and the JSON parser zero, and the mechanism is visible by counting: of the 21 class names in the page, exactly one string coincides with anything in the JSON payload. There is nothing for a redesign to reach.

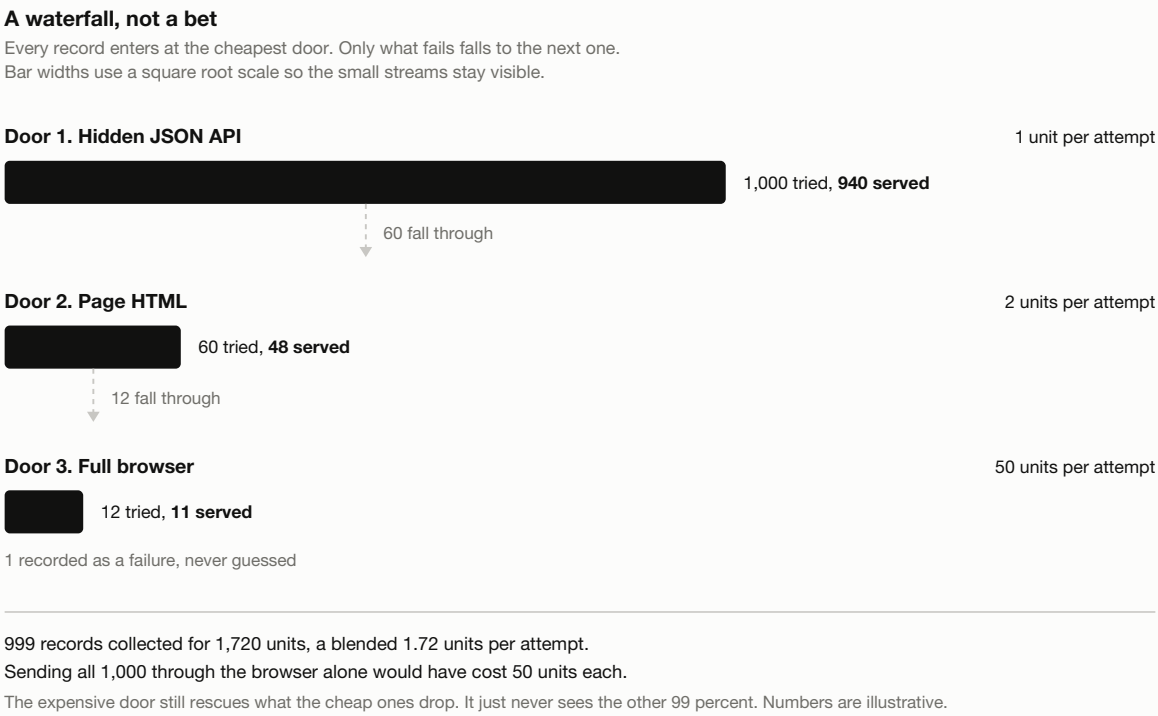
Then the same test cut the other way. Under the four data model mutations the JSON parser failed all four, and every failure landed in the quietest possible mode: records kept flowing with the renamed field silently empty. No crash, no zero row day, a green pipeline delivering rows with a hole in them. The HTML parser, tested against four markup changes of its own, failed two the same silent way. So the ledger reads: the JSON door breaks far less often, and when it does break, the failure is easier to miss. Completeness told a similar both ways story in miniature. Every field arrived 100 percent through both doors except one: the JSON reported no external link for the 14 self posts, 97.2 percent fill, while the HTML door papered over the same fact with an internal link. And of 498 stories both doors returned, titles, links, and authors matched at 100 percent, while points and comment counts disagreed on a handful, 99.0 and 99.6 percent, only because the two walks read live counters minutes apart. None of this decides itself without a monitor, and you have not built one yet; Chapter 8's per-field fill check is its first piece and Chapter 10 is the rest. What it decides right now is that no door exempts you from checking what arrived.

That silent partial is also the reason I now write one clause into every data contract, on either side of the table: acceptance is measured field by field, never by row count. A batch can arrive at full row count with its price column quietly empty, which is exactly how all four JSON failures above landed, and a buyer who accepts on row count has agreed to pay for holes. So the specification names each field and the fill rate it must hold against the last accepted batch. Picture a repricing team feeding on a competitor price column that silently went empty; the pipeline stays green and the reprices chase a ghost.

The waterfall

Every door fails sometimes. The cheap one hits a rate limit, the JSON feed changes shape on a routine deploy, the page you needed renders differently for one region. A collector built on a single door has that door's worst day as its own.

So keep several doors and put them in order. Send every record at the cheapest door that can serve it, and let only the failures fall to the next one down. This is a waterfall, and it is the most useful piece of collector architecture I know.



A waterfall of three doors. A thousand records enter at the cheapest door, which serves 940. Sixty fall through to the page HTML door, which serves 48. Twelve reach the full browser, which serves 11. The expensive door touches about one percent of the volume.

The arithmetic is why it matters. Put a thousand records through a waterfall where an attempt at the JSON door costs one unit, at the HTML door two, and at the browser fifty, with failed attempts charged like successful ones, because the source bills the attempt, not the outcome. If the cheap door serves 940 of them, the middle door serves 48 of the remaining 60, and the browser rescues 11 of the last 12, you have collected 999 records for 1,720 units, a blended 1.72 units per record. Sending all thousand through the browser would have cost fifty units each, about twenty-nine times as much, for one extra record. The expensive door still earns its place; it just never sees the other ninety nine percent of the volume.

One habit makes the waterfall measurable: record which door served each record, in the record. It costs one field, and it answers questions that are otherwise guesswork. When the share of records

served by the cheap door starts sliding week over week, something changed at the source and you now have the date it started. When your proxy bill jumps, you can see whether volume grew or whether traffic migrated down the waterfall to the expensive doors. Without that field, a collector that has silently fallen back to rendering every page in a browser looks exactly like a collector that is working, right up until the invoice.

The doors also tend to fail for different reasons, which is the other reason a waterfall beats a single bet. A site redesign takes out the HTML door and leaves the JSON feed untouched. An API deprecation does the reverse. A shared outage or a block on your address takes every door at once, and no waterfall hedges that; it pays for itself on the far more common days when only one door closes. A rate limit hits whichever door is doing the volume, which is usually the cheap one, and the fallback absorbs the overflow while you fix it. Build the fallback even though it will rarely run. It is there so that the day the cheap door closes is an ordinary day.

When the browser is the right answer

The full browser is last in the waterfall, and it does belong there. There is a real minority of sources where nothing else reaches the data: the values are assembled by scripts from several requests, or they only appear after an interaction that is hard to reproduce by hand, or the shape of the internal feed changes often enough that maintaining a parser for it costs more than rendering the page. In those cases, render it. Using the browser there is not a failure of skill; it is an expensive tool used on purpose, on the fraction of the work that needs it.

What I argue against is reaching for it first, which is what that retail team had done. The browser has a gravitational pull because it always works: no reverse engineering, no hunting for endpoints, just open the page and read it. That certainty is genuinely worth paying for on the hard cases. Paying for it on every record, when the site's own frontend is fetching clean structured data that you could have asked for directly, is how a collection budget disappears into machine time with nothing to show for it.

There is one more argument for the browser that gets made and I do not accept: that rendering pages the way a person would is a way to look less like a collector. That is a detection question, and Chapter 5 takes it seriously on its own terms. It does not decide which door you collect through.

The experiment kept returning to one finding. Neither door was refused for being the wrong shape or the wrong software. One of them was refused for arriving too often from one address, at a pace the site itself had published as acceptable. The door you knock on is only half of what a source sees. The other half is where you are knocking from.

Chapter 4. Network Identity: Proxies and IPs

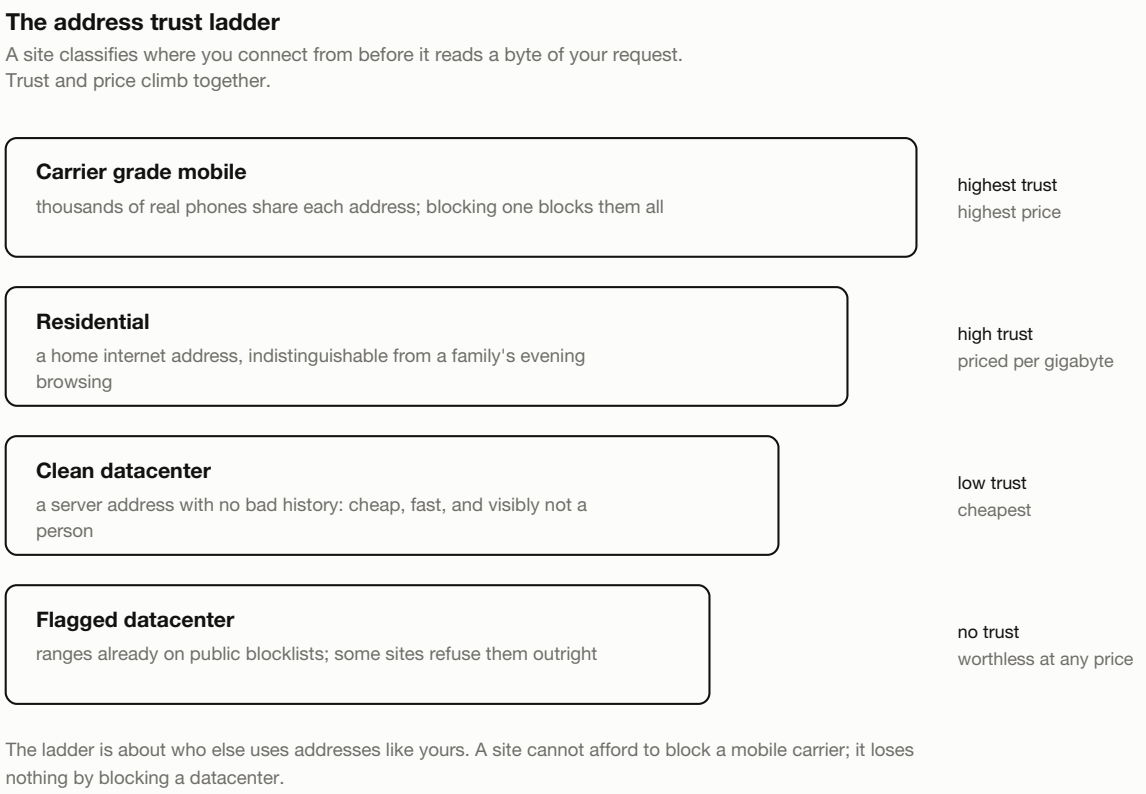
A site decides how much to trust you before it reads a single thing you asked for. The request has not been parsed, the path does not matter yet, no cookie has been checked. The first and cheapest signal the server has is the address the connection came from, and that address already carries a story: who owns it, what kind of network it lives on, and whether traffic from its neighborhood has behaved. Chapter 3 chose the door. This one is about the address you knock from, because to the site the same request from two different addresses is two different events.

The last chapter ended here. A collector honoring Hacker News's published pace was refused anyway, for too many requests arriving from one address too quickly; the door and the software had nothing to do with it. Fix that and you have fixed the most common reason good collectors fail, which is usually an address that ran out of welcome rather than any clever detection.

Your address is a reputation

Every address on the internet belongs to a block that is registered to someone, and that registration is public. Anyone, a site included, can look up an address and learn in milliseconds whether it belongs to a data center full of servers, a home internet provider, or a mobile phone carrier. Those three answers carry very different trust, and the reason is who else is standing near you.

A proxy, for the record: an intermediary that forwards your request so the site sees the proxy's address instead of yours. It is how a collector controls the address it appears to come from, which is the one thing about itself the site judges first.



The trust ladder: flagged data center ranges at the bottom, clean data center above them, home residential higher still, and carrier grade mobile at the top. Trust and price climb together, because they measure the same thing: how much collateral damage a site takes by blocking addresses like yours.

Start at the bottom. A data center address is a server's address, and a server does not shop for espresso machines. Sites know the big cloud providers' address ranges the way you know your neighbors' cars, so traffic from them is visibly automated before it does anything suspicious. Some of those ranges are already on public blocklists from past abuse, which makes them worse than useless: refused on sight. Clean data center addresses, ones with no bad history, still announce themselves as data center addresses. They are cheap and fast and openly machines, which is fine for sites that do not care and fatal for sites that do.

Climb to residential and the calculation inverts. A residential address is a home internet connection, the same kind your own router has, rented through a provider that routes your request out through a real household's line. To the site's address lookup it reads as a household's connection, because it is one, borrowed by the second. That does not make your traffic invisible: address-intelligence services flag known proxy pools, some residential inventory is really provider address space hosted in data centers, and the request pattern is still a machine's. What you are paying for is the collateral, because blocking that address risks blocking a real customer. These are priced by the gigabyte, because bandwidth through someone's home is the scarce thing.

At the top sits mobile. A mobile address belongs to a phone carrier, and carriers do something that makes their addresses radioactive to block: they share one address among many subscribers at once,

tens to a few hundred, a design called carrier grade NAT. Block that address and you have blocked real customers along with it. Sites still challenge and rate-limit mobile traffic constantly; what they hesitate to do is hard-block the address, which is why mobile is the most trusted and the most expensive rung, kept for the sources that fight hardest.

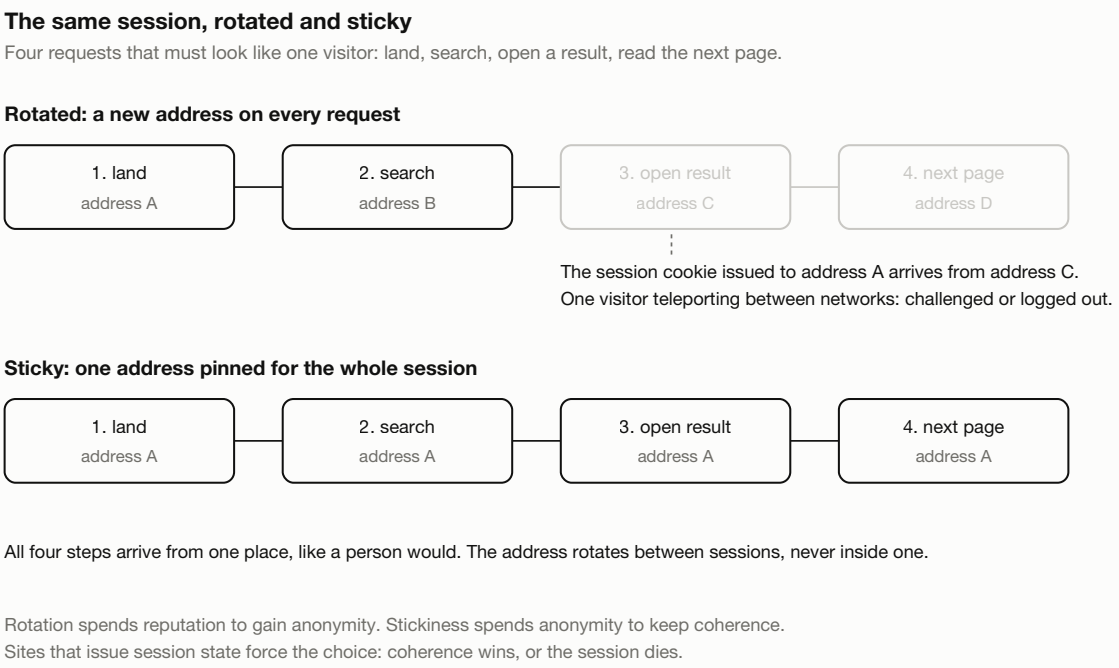
The ladder ranks collateral damage, not quality, and reaching for the top rung by reflex is how collection budgets die. A site loses little by blocking a data center and cannot afford to block a carrier, and everything you pay climbing the ladder is buying that asymmetry. Which rung a job needs is set by how hard its sources look, not by how much you want to be safe. Most catalog and pricing work runs fine on clean data center addresses. You move up only when a specific source starts refusing them, and Chapter 3's audit is where you find out whether it does.

And move up per source, never per platform. The expensive mistake I keep meeting is a portfolio where one hard source forced residential, and then every easy source in the fleet rode the same pool, paying the hard source's rate for traffic a data center address would have carried. Route each source at the lowest rung it accepts and the bill takes the shape of Chapter 3's waterfall: the pricey addresses touch only the traffic that needs them. One routing table, checked once a quarter, keeps the proxy line tracking difficulty instead of habit.

Rotation and stickiness

Once you have addresses, you decide how to spend them, and there are two opposite ways. Rotation means using a different address for each request, spreading your traffic so no single address carries enough of it to look unusual. Stickiness means pinning one address to one line of work and keeping it there. The mistake that breaks more collectors than any detection system is using the first when the job needs the second.

The deciding question is whether the work has memory. Many collection tasks do not: fetch this product, fetch that one, and the two requests share nothing, so it does not matter that they came from different addresses. Rotation is free coverage there, and you should use it. But the moment a task spans several requests that the site expects to come from one visitor, a search that sets a session, a result you open, the next page of listings, memory appears, and rotation turns from an asset into a tell.



The same four step session, run two ways. Rotated, a new address on every request, the site sees the session it handed to the first address come back from a third, and challenges the impossible visitor. Sticky, one address for all four steps, it looks like one person and completes.

Picture the session that fails. Your collector lands on a site, which sets a session cookie, a small token the site issues to recognize the same visitor across requests. The collector searches, opens a result, asks for the next page. Run that with rotation on and each of those four requests leaves from a different address. From the site's chair, one visitor was just handed a session token in California and used it well under a second later from Frankfurt and again from Singapore. No human moves like that. Most sites do not tie a session to one address, because real phones hop between Wi-Fi and cellular all day, but continent-scale travel inside a second is the impossible journey that gets a session challenged or reset, and the collector, which was being polite in every other respect, looks exactly like the thing detection is built to catch. Stickiness fixes it by keeping all four steps on one address, so the session's travel history is the boring straight line a real person produces.

So the rule has two halves that people collapse into one and get wrong. Rotate between independent units of work, and stay sticky within any unit that carries session state. Rotation spreads the traffic and breaks the session; stickiness holds the session and concentrates the traffic. Sites that hand out session state have already chosen for you: stay coherent, or the session dies. Providers sell stickiness as a timed lease on an address, hold this one for ten minutes, and matching that lease to how long your session actually runs is a real part of the design, not a checkbox.

Mismatch that lease and you pay for the work twice. A lease that expires mid session hands the site an impossible visitor, the session dies, and the collector starts over: new lease, new session, the same

pages fetched again, every byte of the retry billed at the same per gigabyte rate as the first attempt. On a residential pool that failure can be invisible in the error log, because every individual request succeeded. So before buying sticky sessions I time the real session, landing to last page, and buy the lease one comfortable step longer. It is the cheapest number in the proxy contract to get right.

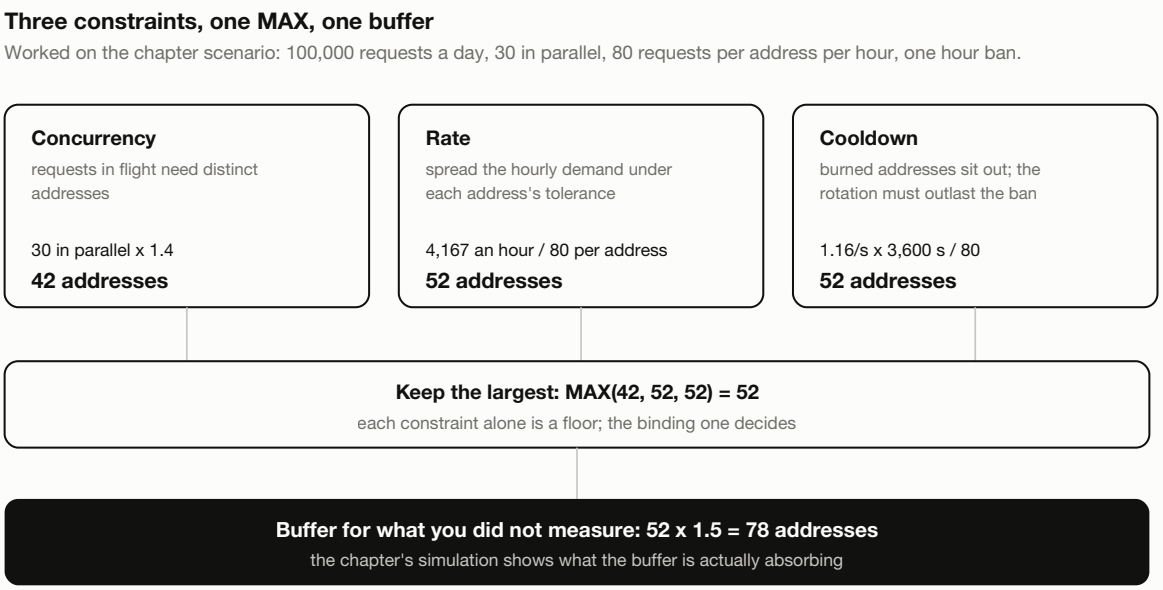
When the country is part of the data

One more thing an address decides, and it is easy to miss because it does not look like a blocking problem. The country an address sits in can change the data itself. A price shown to a shopper in Germany is not the price shown in Brazil; currency, availability, tax, and the assortment on the shelf all shift with where the visitor appears to be. If your collector's addresses live in the wrong country, the site answers honestly and you collect the wrong truth. Nothing errors. The numbers are just not the ones your customer needs, which is the most expensive kind of wrong because nobody notices until someone downstream acts on it. When the deliverable is prices for a market, the address has to stand in that market, and that requirement sizes and shapes the pool before cost does.

Sizing the pool without guessing

Teams either do arithmetic here or pull a number from the air, and guessing is expensive in both directions. Too few addresses and the collector spends its day refused. Too many and you are renting reputation you never use. The good news is that a pool size falls out of three constraints you can each estimate, and the answer is the largest of them with a safety margin, never their sum.

The first constraint is concurrency. If your collector runs 30 requests at the same instant, those 30 need enough distinct addresses that no single one is carrying a suspicious pile of simultaneous connections. The second is the rate limit: each address tolerates only so many requests in an hour before the site starts refusing it, so your hourly volume divided by that per address budget sets a floor on how many addresses you need in rotation. The third is cooldown. When an address does get refused, it is not out for a moment, it is out for a ban, often an hour, and while it sits in the penalty box the remaining addresses have to carry the whole load. A fast rotation through addresses that keep getting burned needs a deep enough bench to cover everyone in timeout.



The three constraints worked on one scenario: concurrency gives 42 addresses, the rate limit gives 52, the cooldown gives 52. Keep the largest, 52, then apply a 1.5 safety buffer for 78. Each constraint is a floor; the binding one decides, and the buffer covers what the three numbers left out.

Take a concrete job: 100,000 requests a day, 30 in parallel, each address good for about 80 requests an hour, a one hour ban when one is burned. Concurrency asks for 42 addresses, the 30 parallel lines with 1.4 times headroom so no address carries two at once. The rate limit asks for 52, and the cooldown, with a one hour ban against a per hour budget, lands on the same 52, because those two floors only separate when a ban outlasts the rate window. So the binding constraint is 52 and a 1.5 buffer lifts the working answer to 78. That buffer covers three facts: the inputs are estimates, real addresses do not all tolerate the same load, and the day you undersize is the day traffic spikes. What the buffer is really buying is what the experiment tests, rather than taking it on trust.

There is a budget rule hiding in this formula. When a collector starts getting refused, the requests that land on my desk are usually purchases: a higher rung, a bigger pool, a new vendor. I make the first response arithmetic instead. Re-derive the pool size from the three constraints, re-probe the pace the source actually enforces, and only then price an upgrade. Most refusals I have debugged were an address short or a pace too hot, both fixable in an afternoon for nothing. A rung upgrade approved on a bad diagnosis becomes permanent, because nobody ever walks spending back down the ladder.

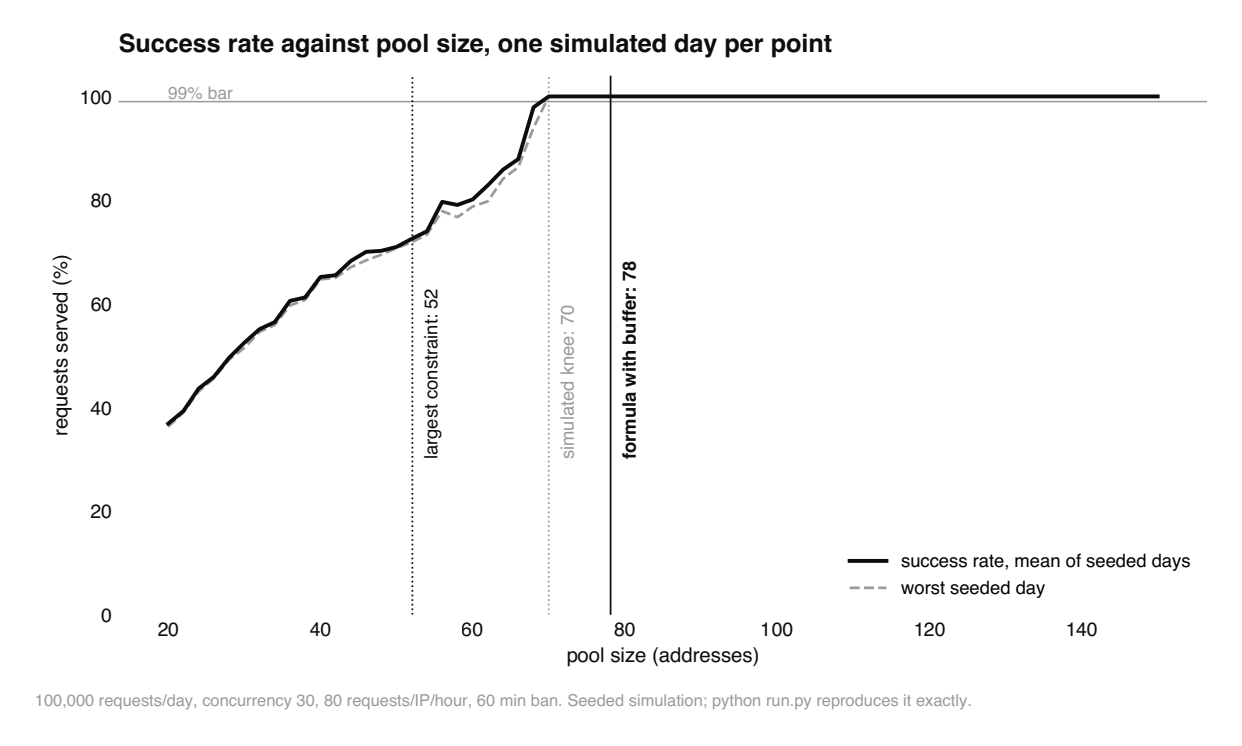
What undersizing actually costs

I have watched people treat pool size as a dial where a bit less money means a bit less coverage, shave the pool by a fifth to save on the proxy bill, and expect to lose a fifth of the data. The brief for this chapter carried that instinct as its hypothesis, written down before the run: the formula and a

simulation would agree on the pool size, and below the right size, success would collapse fast rather than slope down gently.

The experiment is a simulation of one collection day. Requests arrive at random moments through the day, never more than the set concurrency at once. Each address carries its own tolerance, drawn a little above or below the average so the model is not pretending every address is identical, and the collector does the ordinary thing: it rotates evenly, stops using an address the instant a request on it is refused, and takes it back when the ban lifts. It never knows any address's true budget in advance. It learns, the way a real collector learns, by being refused. Then the simulation sweeps pool size from far too small to comfortably large, runs several seeded days at each size so the number is not a fluke, and records what share of the 100,000 requests got through. It is fully offline and seeded, so `python run.py` reproduces every figure here exactly.

The finding is the shape of the curve. At a pool of 40, the collector served 65 percent of its requests. At 56, four fifths. At 66, still under 88. Then between 68 and 70 it jumped from 98 percent to a clean 100 and stayed there for every larger pool. It does not ramp. It has a cliff, and the top of the cliff is a knee you can point at. The simulation's knee landed at 70 addresses. The formula, from the other direction and knowing none of the simulation's internals, said 78. Those two numbers agreeing, one from closed-form arithmetic and one from a seeded day of simulated refusals, is the chapter's main result: the formula lands just past the knee, which is exactly where you want a safety buffer to put you, on the flat ground above the cliff rather than on its edge.



Requests served against pool size. The curve climbs, then jumps to 100 percent at the knee near 70 and stays flat. The formula's answer of 78 sits just past the knee on the flat, which is what the buffer is for. Below the knee the loss is not gentle.

The next numbers kill the dial idea. Shave the pool by a hair, from the knee of 70 down to 66, four addresses out of seventy, and you do not lose four addresses' worth of throughput. The collector falls from a clean 100 percent to under 88, twelve thousand of the day's hundred thousand requests gone for a six percent trim, more than double the loss a straight line would have predicted. Keep cutting and it keeps hurting: under 80 percent by the time the pool is down to 56. The mechanism is a feedback loop, and once you see it you stop trusting linear intuition near a rate limit forever. Too few addresses means each one works harder, working harder trips more bans, every banned address dumps its share onto the survivors, and the survivors, now busier, trip their own bans faster. Right below the knee the system drops off a step. That is why the buffer exists, and why the safe move is to sit above the knee where a bad day costs you nothing, not to trim toward it to save a few dollars and meet that step on the worst possible afternoon.

Carry these numbers off with two limits attached. The knee sits where it does for this job's exact inputs; change the per address tolerance or the ban length and it slides, which is why the formula takes them as inputs rather than baking in a constant. Part of what the 1.5 buffer is covering is the spread between the weakest address and the average one, since the knee is set by the addresses that tolerate least; a pool with a wider spread needs a wider buffer, and the parameters file is where you test your own. And a simulation is a clean-room argument. It proves the shape of the failure, the cliff and the feedback loop, not the precise percentage your real source will show, because a real detector is messier than a per hour counter. What transfers is the lesson rather than the decimals: near a rate

limit, undersizing is a trapdoor you reach by chasing a discount, and the arithmetic that finds the knee is cheaper than the afternoon you would spend underneath it.

The addresses are handled. But an address is only the first thing a site reads about you, and a clean address running a collector that in every other respect screams robot will still get caught.

Chapter 5. Looking Human: Fingerprints and Anti-Bot Systems

The first time a site blocked a collector I had spent a week perfecting, I took it personally. I had matched every header a real browser sends, in the right order, down to the ones nobody thinks about. The site refused me anyway, instantly, on the first request. That is the day I learned that a site starts judging you before you say anything, from the shape of the handshake your software makes to open the connection. I had labored over the headers. They were the wrong layer, and the tell was underneath them.

Chapter 4 covered where you connect from. Once the connection opens, a request can be refused for how it was made rather than for anything it asked, and that is this chapter. The same note as Chapter 2 applies: this is about collecting public data at fair load. It explains how detection works at the level of principle, so you can send traffic that is what it claims to be. It is no recipe for defeating any particular company's product, and the durable strategy turns out to be coherence rather than evasion, which is a happier thing to have to teach.

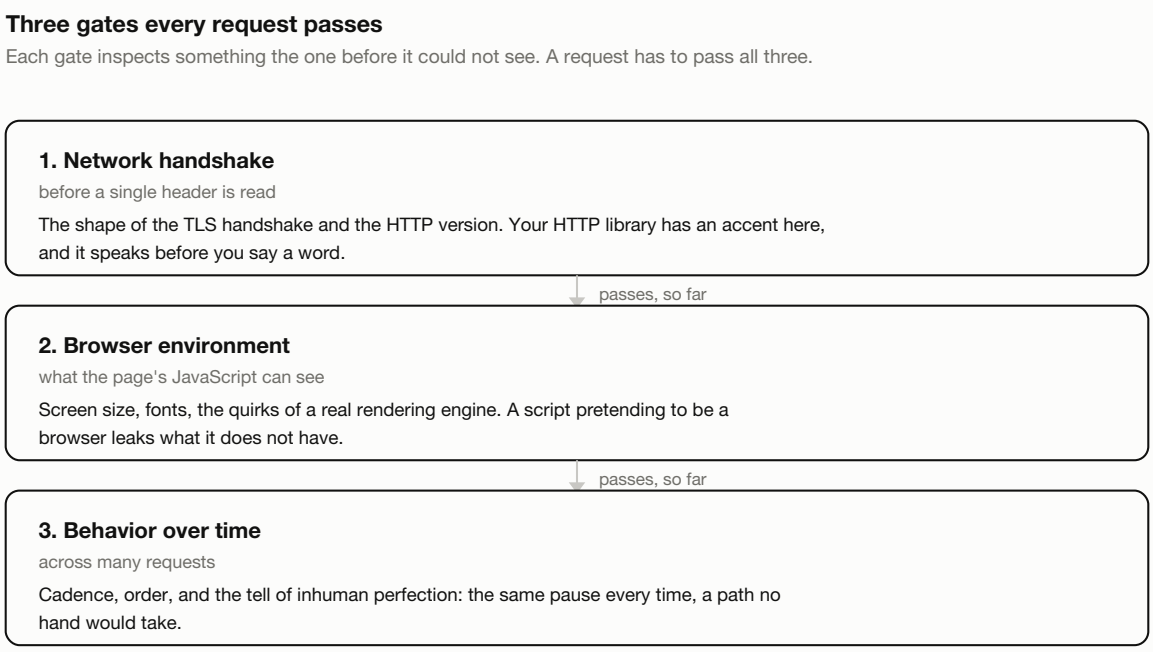
Why sites do this at all

Put yourself on the other side for a moment, because the whole field makes more sense from there. You run a site. Your servers are sized for the humans you expect. One morning a collector that did not size itself points a thousand requests a minute at you, your pages slow down for real customers, and your on-call engineer's phone goes off. You cannot tell, from any single request, whether it is a shopper or a script. So you start looking for the difference, and detection is born from the ordinary need to keep a service standing, not from malice toward collection.

That framing decides the posture from here. Do not set out to defeat the site's defenses; that fight escalates forever, and you lose it the moment your traffic degrades their service. Be traffic they have no reason to stop: coherent, believable, and light. Detection is looking for things that do not add up. Give it nothing that does.

Three gates, not one

What people call anti-bot is really three inspections stacked on top of each other, and they read completely different things. A site may lean on any one of them, and most collectors that fail were caught at the one they were not thinking about.



A request passing three gates in order: the network handshake, the browser environment, then behavior over time. Each gate sees something the one before it could not, and each can stop the request on its own.

The first gate is the network handshake, and it is the one that surprises people, so it gets the experiment later in this chapter. Before any web request is sent, your software and the server perform a short negotiation to set up the encrypted connection, called a TLS handshake, TLS being the protocol behind the padlock in a browser. That negotiation has a shape: which encryption options are offered, in what order, with which settings. Different software makes that negotiation differently, and the difference is stable enough to fingerprint. Your HTTP library has an accent, and it speaks before you have said a word.

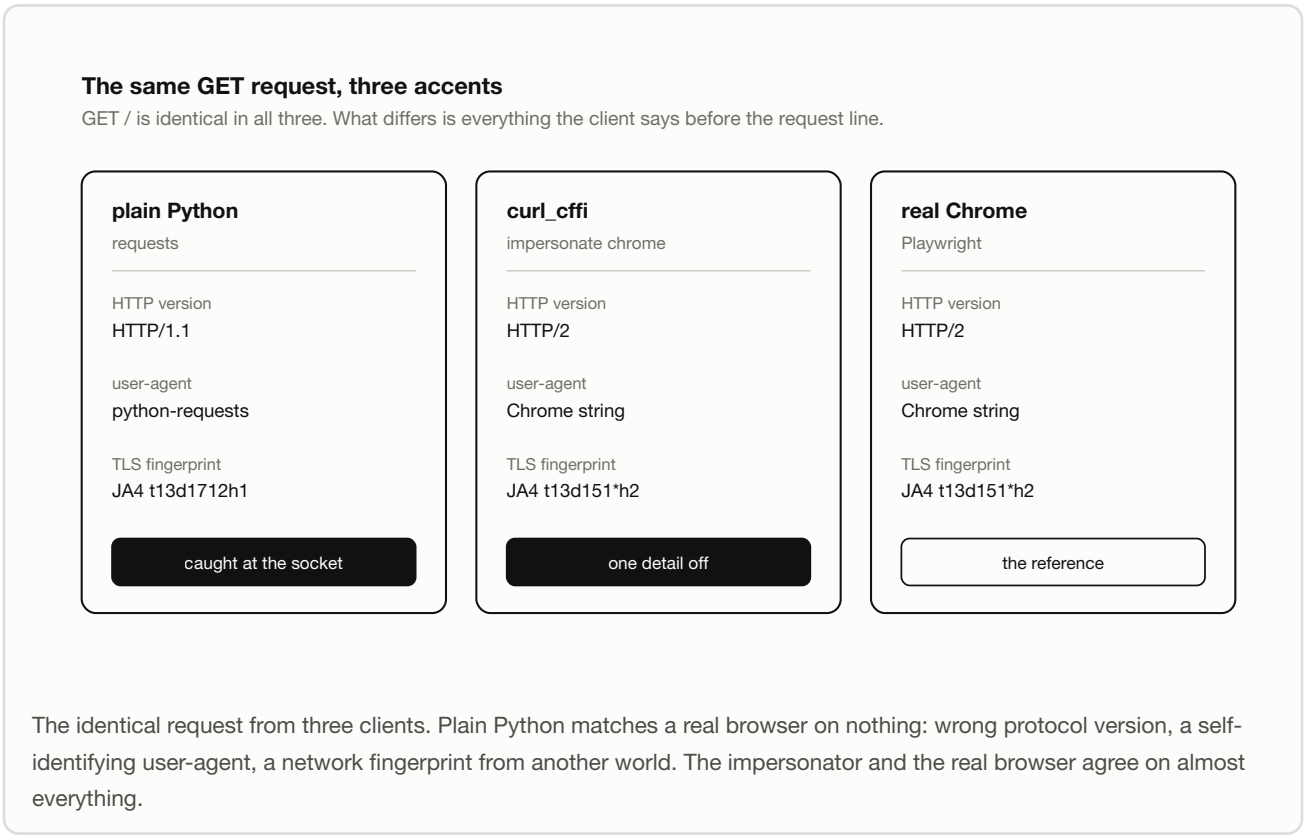
The second gate is the browser environment. If the site serves a page with JavaScript, small programs that run inside the browser, that code can look around the room it is running in: the screen size, the fonts it can detect, the exact quirks of how this engine draws and calculates. A real browser answers those questions one way. A script pretending to be a browser, or a stripped-down automated browser, answers them wrong or cannot answer at all, and the gaps are the tell.

The third gate is behavior, and it needs more than one request to see anything. It watches the rhythm and the route: how long between requests, in what order pages get visited, whether the timing has the small irregularity of a hand or the metronome regularity of a loop. One request has no rhythm to read; a thousand do.

What the socket already knows

I wanted to see the first gate with my own eyes rather than describe it from memory, so this chapter's experiment is a mirror held up to three clients. There are public services that echo back exactly what they saw of your connection, the handshake shape and all, and I sent the same simple request to one of them from three different clients: a plain Python HTTP library, an impersonating library built to mimic a real browser's handshake, and a real Chrome browser driven by automation. Same request, same destination. The only variable is who is knocking. Everything below comes from that capture, taken on the last day of July 2026, and it ships with the chapter so `python run.py` reproduces it.

The plain Python client gave itself away at every level at once, which is the point. It spoke an older version of the web's transport protocol, HTTP/1.1, while both the impersonator and the real browser spoke a newer one, HTTP/2. Its handshake offered a different set of encryption options, in a different arrangement, producing a network fingerprint with no resemblance to a browser's. And it announced itself, in plain text, as `python-requests`. A site does not need clever detection for this. Two of those three tells, the protocol version and the shape of the handshake, are set before the request's headers are read at all. The user-agent is the third, read the moment a site parses those headers. My week of perfect headers would have been read, if it was read at all, only after the connection had already been judged by the client that opened it.



The identical request from three clients. Plain Python matches a real browser on nothing: wrong protocol version, a self-identifying user-agent, a network fingerprint from another world. The impersonator and the real browser agree on almost everything.

Then the impersonating client, and here the data did something better than prove the obvious. The library exists to wear a real browser's handshake, and it wore it well. It spoke HTTP/2, presented the same list of encryption options as the real Chrome, and, tested across three connections, it varied part

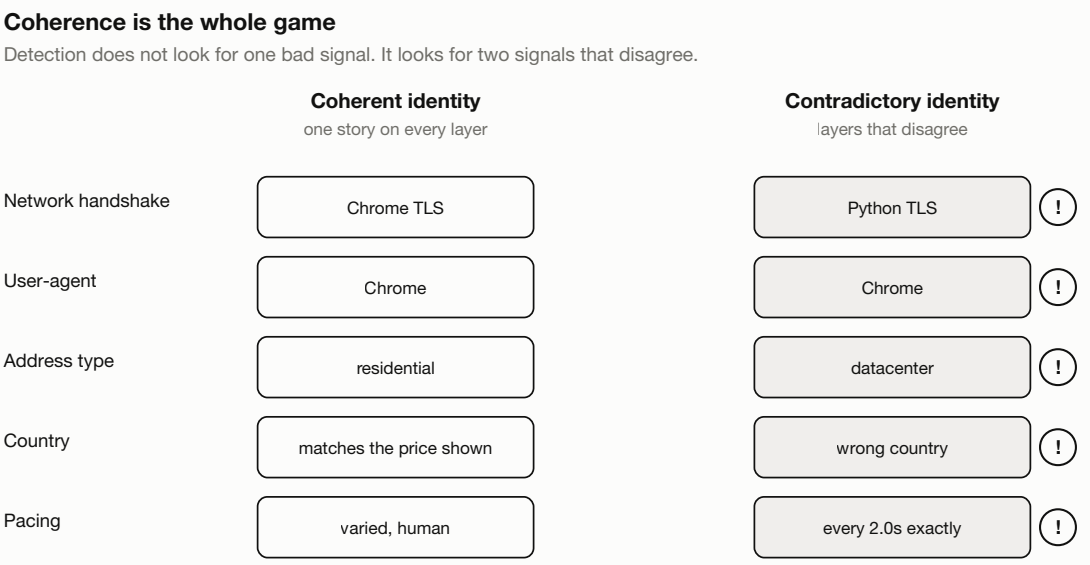
of its handshake each time exactly the way modern Chrome does. That last point matters more than it looks. Today's Chrome deliberately shuffles a piece of its handshake on every connection, so the older style of fingerprint, called JA3, the one that hashes the handshake's exact byte order, changes from one Chrome connection to the next. I had half-expected the impersonator to betray itself by being too consistent, by faking a single frozen fingerprint while real Chrome shimmered. It did not. It shuffled too. The naive tell I was ready to write about was not there, so I followed the data instead.

But the data did leave one seam, and finding it is the real lesson. The newer, sturdier fingerprint, JA4, is built to see through that shuffling: it sorts the shuffled parts before hashing, so the per-connection reshuffle stops changing it the way it changes JA3. Read the three clients that way and a gap appears that the shuffle was hiding. The impersonator matched real Chrome's encryption options exactly, the two share an identical cipher fingerprint, but the part of the fingerprint that summarizes the handshake's extensions still differed from genuine Chrome's. And the real browser did something the impersonator did not: across three connections its sorted fingerprint itself shifted slightly, flickering between two neighboring values as Chrome included one more or one fewer element in its handshake, while the impersonator held to a single fixed value. Genuine software varied here in a way the copy did not. The copy was excellent, and still not identical. At this depth the first gate does not check for a byte-perfect match to one browser. It asks whether the handshake belongs to the family it claims, and a residue like this says it does not.

I want to be careful about what this proves, because the temptation is to over-read it. It does not prove the impersonator gets caught in practice; in my experience most sites do not inspect at this depth, and against most of the web a good impersonating client sails through the first gate. What it proves is narrower and more useful: the discriminating information is present at the socket, before headers, before cookies, before behavior. The plain client is separable from a browser in the opening handshake, decisively. The impersonator is separable too, if the inspector looks closely enough, by a residue it did not know to hide. The first gate is a real inspection with real signal in it, and a user-agent string does not satisfy it. Whatever story you intend to tell has to be true at the handshake, or it is not true.

Coherence is the actual principle

The mistake that follows people through this field for years is treating detection as a checklist of tricks to satisfy: fix the user-agent, add the headers, match the fingerprint, and imagine that stacking enough correct details eventually adds up to invisible. It does not, because detection cares less about whether each detail is correct than about whether the details agree. The contradiction between them is its loudest signal.



Two identities across five layers. The coherent one tells a single story on every layer. The contradictory one pairs a browser user-agent with a Python handshake, a datacenter address with a claim to be residential, a wrong country, and metronome timing. Every contradiction is a flag.

A request tells a story on every layer at once. The handshake says what software you are. The user-agent says what software you claim to be. The address says what kind of connection you are on. The country says where you are. The timing says whether a hand or a loop is driving. Detection lines those up and looks for the pair that disagrees. A browser user-agent riding a Python handshake is the classic disagreement: the request announces itself as Chrome and then shakes hands like a script, and the contradiction is louder than either signal alone would have been. The handshake alone already gave the client away. Wearing a Chrome user-agent on top did not hide that. It added a claim the socket beneath refused to back, and two signals that cannot both be true are easier to catch than one that is merely unusual.

This is why coherence beats a bag of tricks, and it is good news for someone collecting legitimately. Assembling a perfect fake from parts is the wrong work. The work is to be one consistent thing all the way down. If you present as a browser, be a browser at the socket too, which is what the impersonating libraries and real automated browsers from the experiment are for. If you present as residential traffic from Germany, connect from a residential address in Germany, the Chapter 4 decision reaching up into this one. If you present as a person reading pages, do not read them on a metronome. The layers stop contradicting each other when they all tell the same true story, which is also much less work to maintain than a web of matched lies.

None of this means you have to look like a person, or like a browser at all. Coherence is only the layers agreeing. A client that is openly a script, never dressed up as Chrome, kept light and asking for public data plainly, tells a true story too, and usually the easiest one to keep. Standing out as a script is not

the same as being blocked. Save a real browser for pages that genuinely need one, since that path costs far more to run and keep working.

Anti-patterns that fingerprint you instantly

A few specific contradictions show up so often they are worth naming, because each one is a single detail that undoes everything else.

The impossible value is the first. A real site's own interface only ever sends certain values, because a human clicking it can only produce certain values. When a collector sends a request asking for nine million results per page because it wanted everything in one shot, it has sent a number no human interface would ever generate, and that single field marks it more reliably than any handshake. The fix is discipline: send values the real interface sends, derived from what the site actually offered, never a round sentinel you invented for convenience.

The metronome is the second. Nothing that is really a person reading pages produces perfectly even gaps. A human's timing is lumpy, distracted, irregular. A collector that fetches every 2.0 seconds exactly is not blending into human traffic, it is signing every request with a ruler. Vary the timing, and vary it because you actually should be slower, not just to look random.

The crowd of one is the third, and it is the Chapter 4 lesson again. One address presenting a hundred different browser identities in an hour is a contradiction in space the way the metronome is a contradiction in time: no single home connection is a hundred different people at once. Identities and addresses have to move together, or the mismatch between them becomes the signal.

When you are blocked anyway

Even coherent, respectful collection gets blocked sometimes, and how you treat that block decides whether your data is trustworthy. The one rule that matters: a block is not data. When a request comes back challenged, throttled, or refused, you have learned nothing about the thing you were collecting, only that this attempt did not land. A block does not always look like one. Sometimes it comes back as an ordinary success that carries a challenge page or an empty shell where the data should be, and that is still not data. The unforgivable error, the one that poisons datasets, is to record the block as a result, to let a challenge page become a price of zero or a product marked not found. Chapter 8 is about catching that. For now, hold the line that a block is an ambiguous non-answer to be retried or escalated under the back-off discipline Chapter 6 builds, never a fact to be written down.

Be coherent so you are rarely blocked. Be respectful so a block is rare and recoverable. And when a block does come, never let it be mistaken for the truth you were sent to collect.

Chapter 6. Building Resilient Collectors

The worst data bug I have ever shipped did not crash anything. A collector I ran was pulling prices from a retailer, and one afternoon the retailer started challenging our requests. The collector did what it had been told to do with a failed request: it moved on and wrote down what it got. What it got was a challenge page with no price on it, and the code, finding no price, recorded zero. For most of a day we delivered a feed claiming a whole catalog of products had dropped to nothing. Nobody's program crashed. Every job reported success. The dashboard was green the entire time it was wrong.

That day taught me the discipline this chapter is built around, and it is a way of working more than a clever technique. A collector has to succeed when it can, it has to know when it did not, and it must never, under any pressure, let a failure wear the costume of a fact. Chapters 3 through 5 got you collecting. This one is about the other 15 percent, the share the experiment assumes, the requests that fail, and why designing for that minority rather than the easy majority is what separates a collector you can bill from one that lies to you.

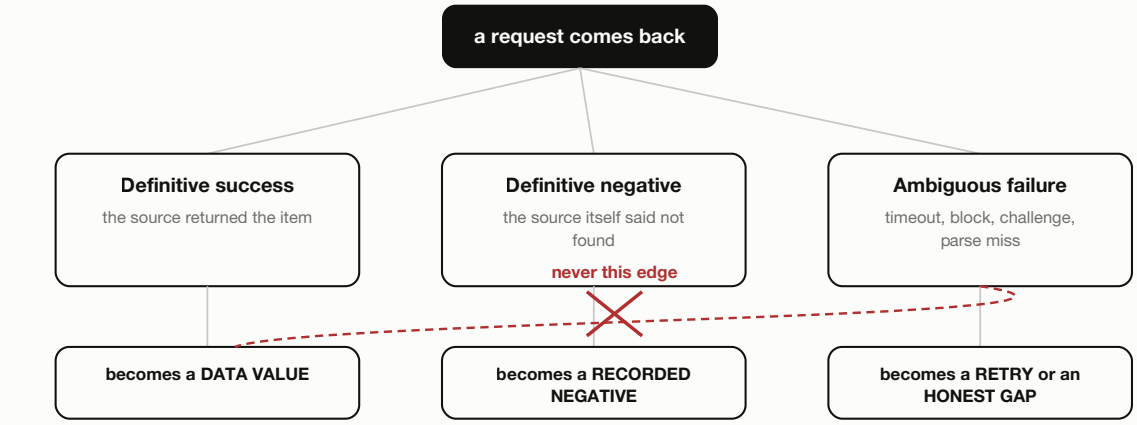
Failure is the normal case

New collectors are written for the happy path, because the happy path is what you test on. You point it at a page, it works, you ship it. Then it meets production, where a real source times out, rate limits you, serves a challenge, returns half a page, or simply has a bad minute. None of that is exotic. It is Tuesday. A collector that treats failure as an exception to be surprised by will be surprised several times an hour, forever.

So flip the design around. Assume every request can fail, and ask a different question: not whether a request failed, but what you are allowed to do with the failure. That question has one dangerous answer. Ruling it out is the job.

The three buckets

Every outcome a request can have sorts into three buckets, and keeping them separate is the habit I care about most in this work.



A request comes back and sorts into one of three buckets. A definitive success becomes a data value. A definitive negative, the source itself saying not found, becomes a recorded negative. An ambiguous failure, a timeout or block or challenge or parse miss, may only become a retry or an honest gap. The edge from an ambiguous failure to a data value is crossed out in red.

The first bucket is definitive success. The source answered, the answer is the thing you asked for, and it becomes a data value. Simple.

The second bucket is the one people forget exists, and forgetting it causes its own bugs. It is the definitive negative: the source itself, clearly and on purpose, told you the item is not there. A product page that returns a real not found, not a challenge and not an error, an actual answer that the thing does not exist. That is information, and it is allowed to become a recorded negative, a row that says this item was gone on this date. One rule guards this bucket: only an unmistakable answer from the source earns it.

The third bucket is everything else, and among failures it is by far the largest: the ambiguous failure. A timeout. A block. A challenge page. A rate limit. A response that arrived but did not parse. All of them share one property, and it decides what you may do with them: they tell you nothing about the item. The attempt did not tell you the price is zero or that the product is gone. It only told you that this attempt did not land. The only things an ambiguous failure may become are a retry or an honest gap, a row that says we do not know, recorded as missing rather than invented.

The forbidden edge is the red line in the diagram, and it is the bug from the top of this chapter. An ambiguous failure may never become a data value. A block does not mean a price of zero, a timeout does not mean the product vanished, and a challenge page does not mean out of stock. The moment

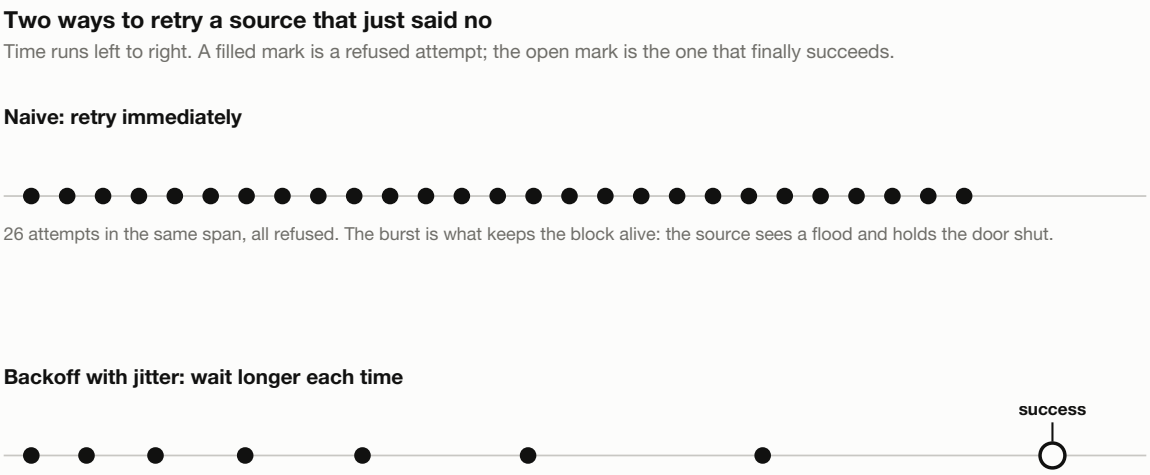
your code lets any failure fall through into a real value, it has manufactured data, and manufactured data is worse than no data in a way that took me years to feel in my bones: it is billed, it is confident, and nothing downstream can tell it from the truth. A missing row announces itself. A fabricated row hides, gets delivered, gets acted on, and shows up as a decision someone made on a number you invented. Chapter 8 catches these after the fact. Never creating them in the first place is cheaper, and that is the job here.

Retries, and how they go wrong

Once a failure can only become a retry or a gap, the next question is how to retry. The naive answer is: it failed, try again right now, a few times, then give up. It feels responsive. It does real damage, and the experiment below measures how much.

The problem is that the most common reason for a cluster of failures is that the source has started pushing back, and immediate retries push into the push-back. A collector running many requests at once, which is every real collector, responds to a struggling source by instantly sending all its failed requests again, which is a burst on top of the concurrency that was already testing the limit, and the burst is what keeps the block standing. In the experiment below the workers cross the source's limit on their own; what the naive retries add is the habit of spending the whole attempt budget against a door that has already closed. It is a collector fighting itself.

The fix is old and boring and works: exponential backoff with jitter. Backoff means you wait longer after each successive failure, half a second, then one, then two, then four, giving a struggling source room to recover instead of a fresh flood. Jitter means you add a random wobble to each wait, so that a thousand requests that failed together do not all retry at the same instant and re-form the very burst you were trying to avoid.



Two retry timelines against a source that has started refusing. Naive retry sends a dense burst of attempts, all refused, and the burst keeps the block alive. Backoff with jitter spaces the attempts further apart each time, the gaps let the block expire, and the attempt after it succeeds.

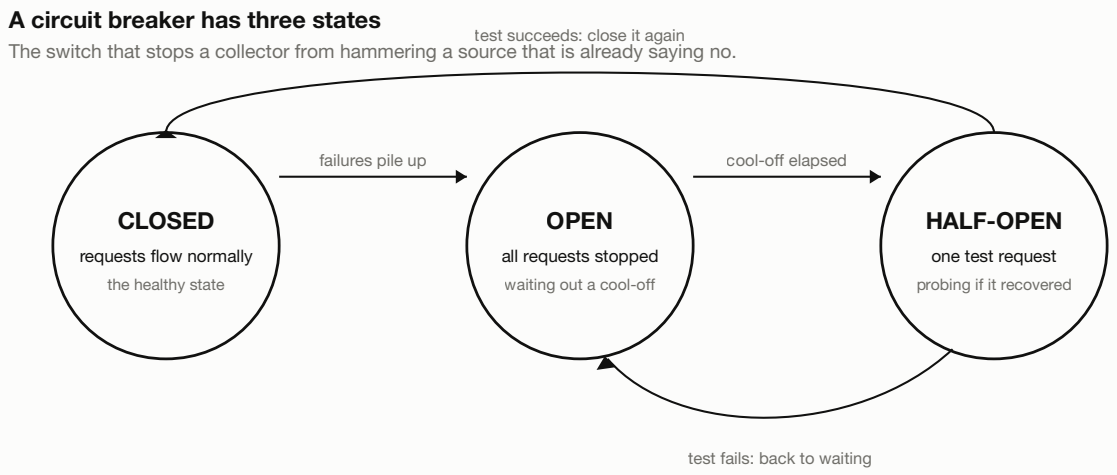
Timeouts are budgets, not settings

A timeout is how long you wait before deciding a request has failed, and the mistake is having only one, buried deep. If the only timeout is on the individual network request, a collector can still hang far past any deadline: an item that retries ten times, each retry waiting the full per-request timeout, can burn minutes on a single item, and a job made of such items runs as long as items times retries allow, a bound nobody planned for. I have watched a job that was supposed to take an hour still going the next morning, every individual request dutifully under its timeout, the whole making no progress.

The discipline is to budget time at every level. A per-request budget: this single fetch gets N seconds. A per-item budget: all the retries for one item together get M seconds, after which the item is recorded as a gap and the collector moves on. And a per-job budget: the whole run gets a wall-clock ceiling, after which it stops, delivers what it has, and reports the rest as not collected. The per-item budget is the one people miss, and it is the one that turns an infinite hang into a bounded gap. A budget you did not set is a budget set to infinity.

Circuit breakers

Backoff spaces out one item's retries. A circuit breaker handles the case where the whole source has gone bad, and it is the piece that makes a collector stop hurting itself. The idea is a switch with three positions.



A circuit breaker has three states. Closed lets requests flow and is the normal, healthy state. When too large a share of recent attempts fails, it trips to open, which stops all requests for a cool-off. After the cool-off it goes half-open and sends a single test request: if that succeeds it closes again, if it fails it re-opens and waits more.

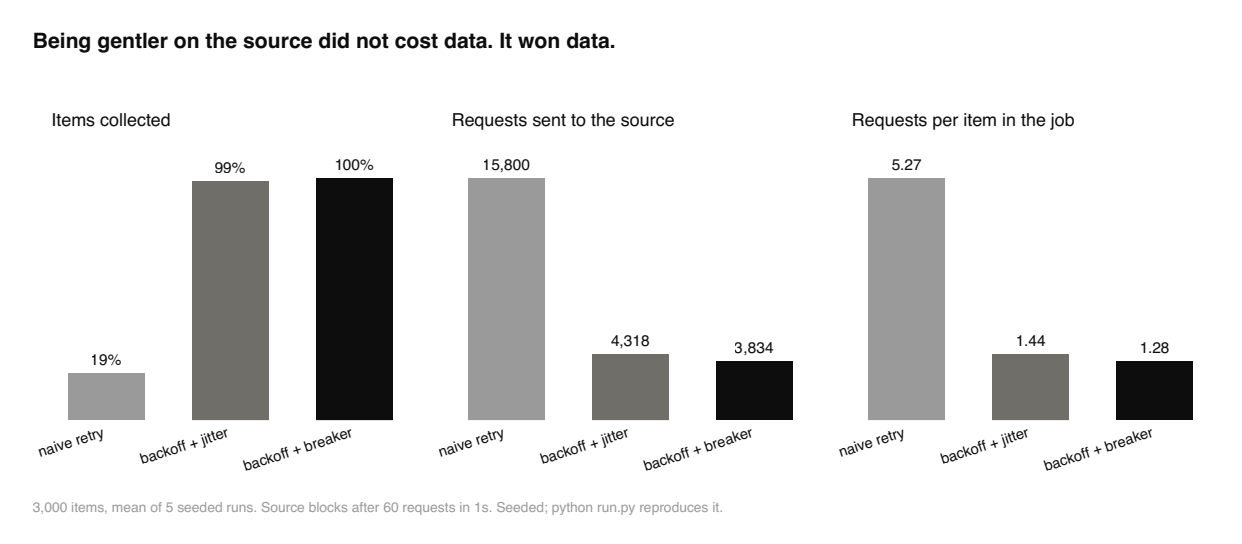
Normally the breaker is closed and requests flow. When failures pile up past a threshold, say more than half of recent attempts failing, the breaker opens, and while it is open the collector sends nothing at all to that source for a cool-off period. This is the move that feels wrong and saves the job: in the face of a source that is refusing everything, the most productive thing a collector can do is stop, because every request it sends into a block fails, and on many sources also re-trips the block the moment it lifts. After the cool-off the breaker goes half-open and sends one lone test request. If that succeeds, the source has recovered and the breaker closes back to normal. If it fails, the breaker re-opens and waits again. The breaker is how a collector notices, without a human, that it should back off the whole source rather than keep grinding item by item.

Three policies against one bad source

I wanted to know whether all this politeness costs data, because there is a real intuition that says the aggressive collector, hammering away, must at least collect more of what it can before it gets blocked. So this chapter's experiment pits three policies against the same unreliable source: naive immediate retry, backoff with jitter, and backoff plus a circuit breaker. The source behaves like a real one, a share of requests fail transiently, and sustained hammering trips a block that lasts a while and fails everything underneath it. Twenty workers run at once, the way a real collector does, because it is the concurrency that lets naive retries stampede. The job is 3,000 items, run five times per policy with different random seeds so the numbers are not a fluke. Everything is simulated and seeded, so `python run.py` reproduces every figure below.

The result was not close, and it did not split the way the aggressive intuition predicts. Naive retry collected 19 percent of the job. Not 19 percent less, 19 percent total: it tripped the block early, then

spent the rest of the run hammering a blocked source with immediate retries, most of them failing into the very wall its own bursts kept standing. Backoff with jitter collected 99 percent. Backoff plus a breaker collected 100 percent. The gentle policies did not trade data for manners. They collected five times as much.



Three panels from the run. Items collected: naive 19 percent, backoff 99 percent, breaker 100 percent. Requests sent to the source: naive 15,800, backoff 4,318, breaker 3,834. Requests per item in the job: naive 5.27, backoff 1.44, breaker 1.28.

Then look at what each policy did to the source. The request counts turn the result from surprising into obvious. Naive retry sent 15,800 requests to collect its miserable 19 percent: 5.27 requests for every item in the job, and about 27 for every item it actually got, most of them wasted against the block. The breaker sent 3,834, about a quarter as many, 1.28 per item with the whole job collected, so for it the two ratios are the same number. The aggressive collector put four times the load on the source and came home with a fifth of the data. It did not trade politeness for yield. It lost on both at once, because against a source that hard-blocks bursts, the load it inflicted is what caused the failures it suffered. A source that only slows you down, rather than shutting the door, narrows that gap without reversing it. So I stopped thinking of resilience and politeness as a trade-off. In this experiment they turned out to be the same property.

One cost to name, because the experiment shows it. The breaker took longer in wall-clock time, because it deliberately waits out its cool-offs instead of hammering: in these runs it spent about seven minutes where naive spent under four. But naive's four fast minutes produced a fifth of the data and a mountain of load, so its speed is the speed of failing quickly. Backoff sat in between, nearly all the data at nearly the lower load without the breaker's patience. Slower is not better in itself. The breaker spends time to buy yield and gentleness, and time is usually the cheapest of the three to spend.

Idempotency, so you can always just re-run

One more property makes all of the above safe to operate: idempotency. A collector is idempotent when running it again with the same input does no damage: no double counts, no double bills, no duplicate rows. The practical version is that each record has a natural key, the identity of the thing, the product code and the date, say, and writing a record means upserting on that key, updating the row if it exists rather than blindly appending a new one. Get this right and recovery becomes trivial: a job died halfway, so you run it again, and the items it already has are simply rewritten under the same keys, with fresher values if the source moved, while the missing ones fill in. Get it wrong and every retry is a risk, and operators start being afraid to re-run the thing. That fear is its own failure: a collector nobody dares restart ends up left broken.

The resilience checklist

I keep this on one page and run every collector against it before it goes near production.

1. Every outcome is classified into definitive success, definitive negative, or ambiguous failure, and the three are handled by separate code paths.
2. No ambiguous failure can produce a data value. A block, timeout, challenge, or parse miss becomes a retry or an honest gap, never a number.
3. A definitive negative is only recorded when the source unmistakably said not found, never inferred from an error.
4. Retries use exponential backoff with jitter, never immediate repetition.
5. Time is budgeted per request, per item, and per job, and the per-item budget exists.
6. A circuit breaker stops all traffic to a source that is refusing, and probes before resuming.
7. On a rate-limit or error signal, the collector slows itself without waiting for a human.
8. The collector is idempotent: safe to re-run on the same input, with records upserted on a natural key.
9. A run reports what it collected and what it could not, and the two never blur.

If I could keep only one line it would be item 2, because it is the one whose absence you cannot see. A collector that breaks most of the others fails where you can see it. A collector that breaks item 2 keeps running, keeps reporting success, and feeds false values into everything downstream; items 3, 8, and 9 guard against the related failures. Everything from here on assumes the data you collected is real. Part III checks that assumption again at the other end, where the data is delivered, and it can only hold if no failure was ever allowed to pose as a fact.

Chapter 7. Extraction: From Page to Record

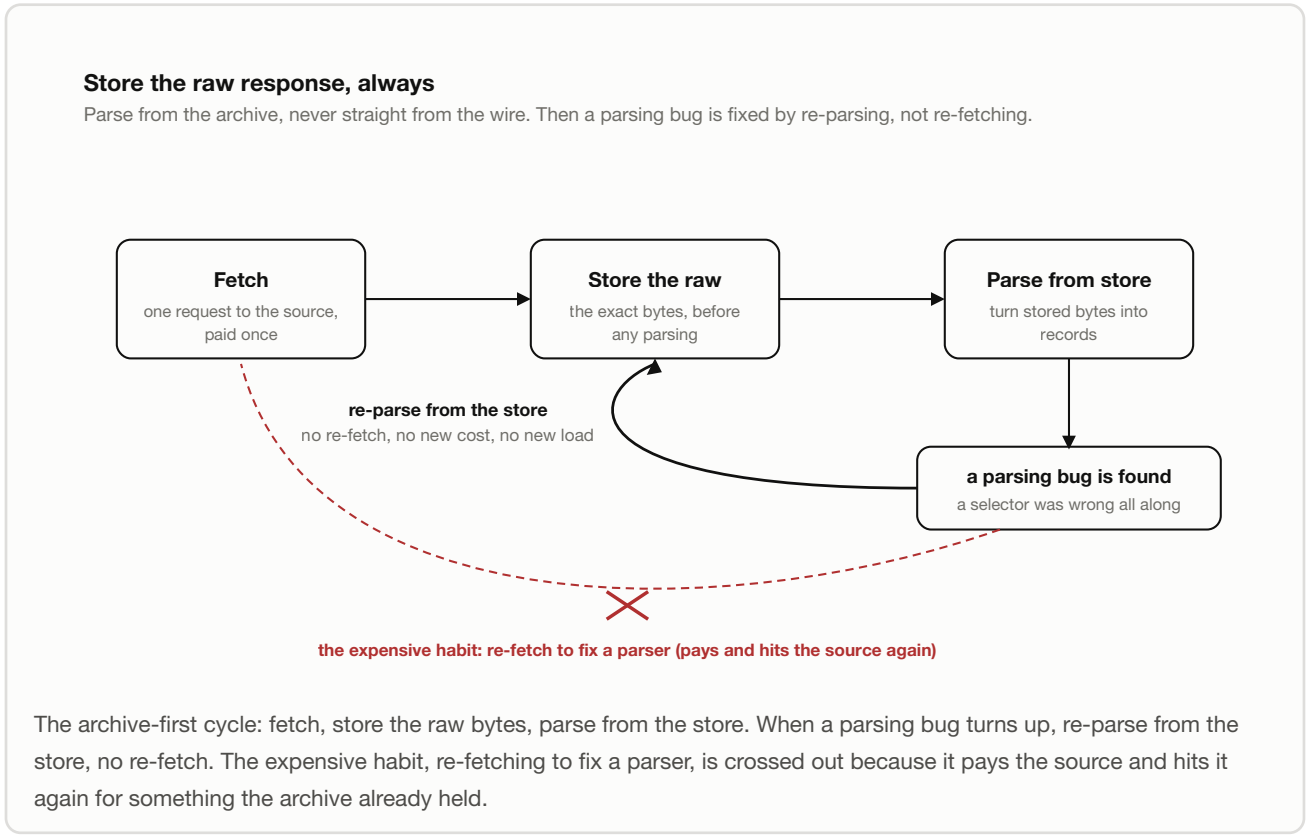
The bytes are not the data. I wish someone had told me that early, because for a long time I thought the hard part of collection was getting the page, and that once the page was in hand the record fell out of it. It does not. Between the response you fetched and the row your customer reads sits a step where you decide, field by field, what the page means. That step is extraction, and it is where the record actually gets made. A fetch that works and an extractor that is subtly wrong do not cancel out; together they produce a confident, well-delivered, wrong number.

Chapters 3 through 6 were about getting bytes off the internet without lying and without breaking. Turning a page into a record is the step everyone skips in the telling, the one between the bytes and the validated row. A cheap habit saves more grief here than anything else; there is a place to look before you write a single selector; and there is a failure mode that is a cousin of Chapter 6's forbidden edge. Those three are how extraction stops being the part that breaks every other week.

Store the raw response, always

The habit that has saved me the most grief costs almost nothing: store the raw response before you parse it. Fetch the page, write the exact bytes to storage, and then parse from the stored copy, not from the wire. It sounds like bookkeeping. A parsing bug against stored bytes costs an hour; the same bug after you threw the bytes away costs you the data.

Think about what a parsing bug is. Your extractor has been reading a field wrong, maybe for a week, maybe since the day it shipped. You discover it. Now, what do you have to work with? If you parsed straight from the wire and threw the bytes away, the answer is nothing: to fix the past you would have to re-fetch every page, which means paying for the requests again, putting that load on the source again, and hoping the pages even still say what they said last week, which for prices and stock they do not. The history is simply gone. But if you stored the raw responses, the fix is a re-parse. Correct the extractor, run it back over the archive you already have, and the past repairs itself with no new request, no new cost, and no new load on anyone.



The archive-first cycle: fetch, store the raw bytes, parse from the store. When a parsing bug turns up, re-parse from the store, no re-fetch. The expensive habit, re-fetching to fix a parser, is crossed out because it pays the source and hits it again for something the archive already held.

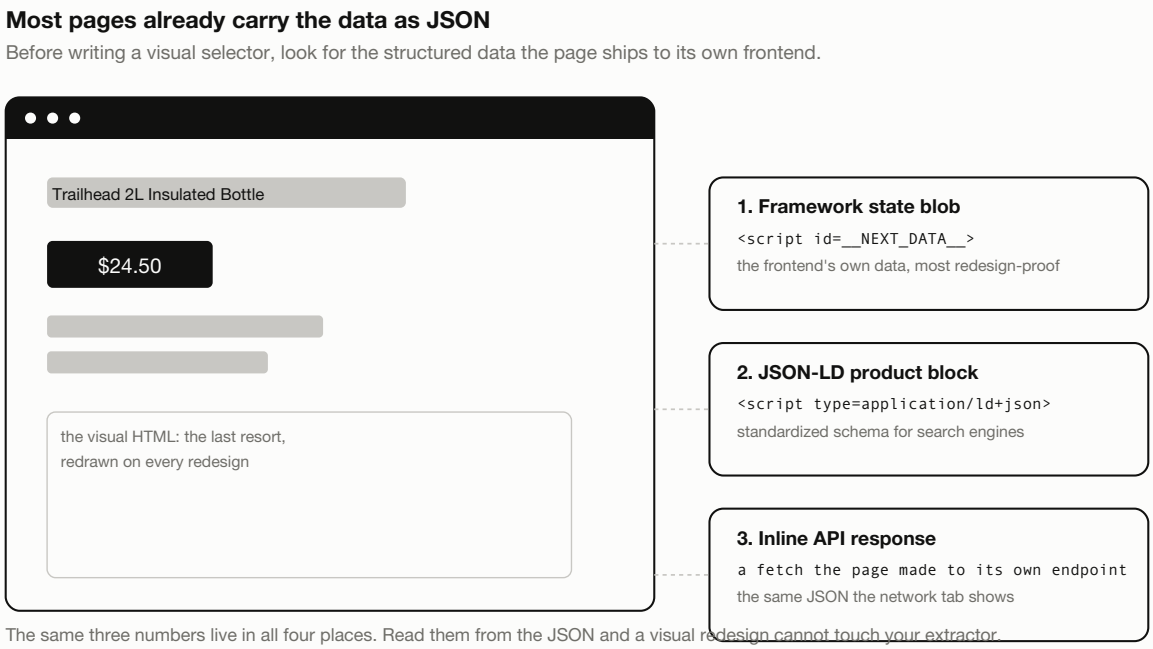
Storage is the cheapest thing in this book. A compressed HTML page is tens of kilobytes; a year of a large collection is a rounding error against what the collection itself costs to run. What that trivial spend buys is enormous: every parsing fix applies retroactively, you can audit what the source served on any past date, and the morning a site redesigns you can diff today's raw response against yesterday's and see what moved. I have debugged a customer's angry question about a number from three weeks ago by opening the stored response from that day and reading it with them on the call. Without the archive that conversation is a shrug. With it, it is a fact. Treat storing the raw as non-negotiable, the way Chapter 2's checklist treats never fetching what you already have.

The data is usually already in the page

The place to look before writing any selector is the habit that separates people who have done this for years from people who are about to have a hard month. A modern web page is a program that a site hands your browser, and that program very often arrives carrying its data as structured JSON, right there in the page, or fetches it moments later. Chapter 3 met this as the hidden JSON door. Here it comes back one level in: even when you are working with the rendered HTML, the clean structured data is usually embedded inside that HTML, and you should reach for it first.

There are three common forms, and you learn to know their shapes by sight. The first is the framework state blob: modern sites built on common frontend frameworks embed the whole page's data in a script tag, often under a name like the one this chapter's example uses, so the code that draws the page has something to draw from. The second is JSON-LD, a block of standardized product

data that sites publish for search engines, sitting in a script tag that announces itself as structured data. The third is an inline API response, the same JSON the site's own frontend fetched, sometimes baked directly into the page. All three hand you names, prices, and identifiers under named keys, not as text you have to dig out of visual markup; the framework blob's values even arrive typed, while JSON-LD's price is still a string you parse, just one that announces itself as a price.

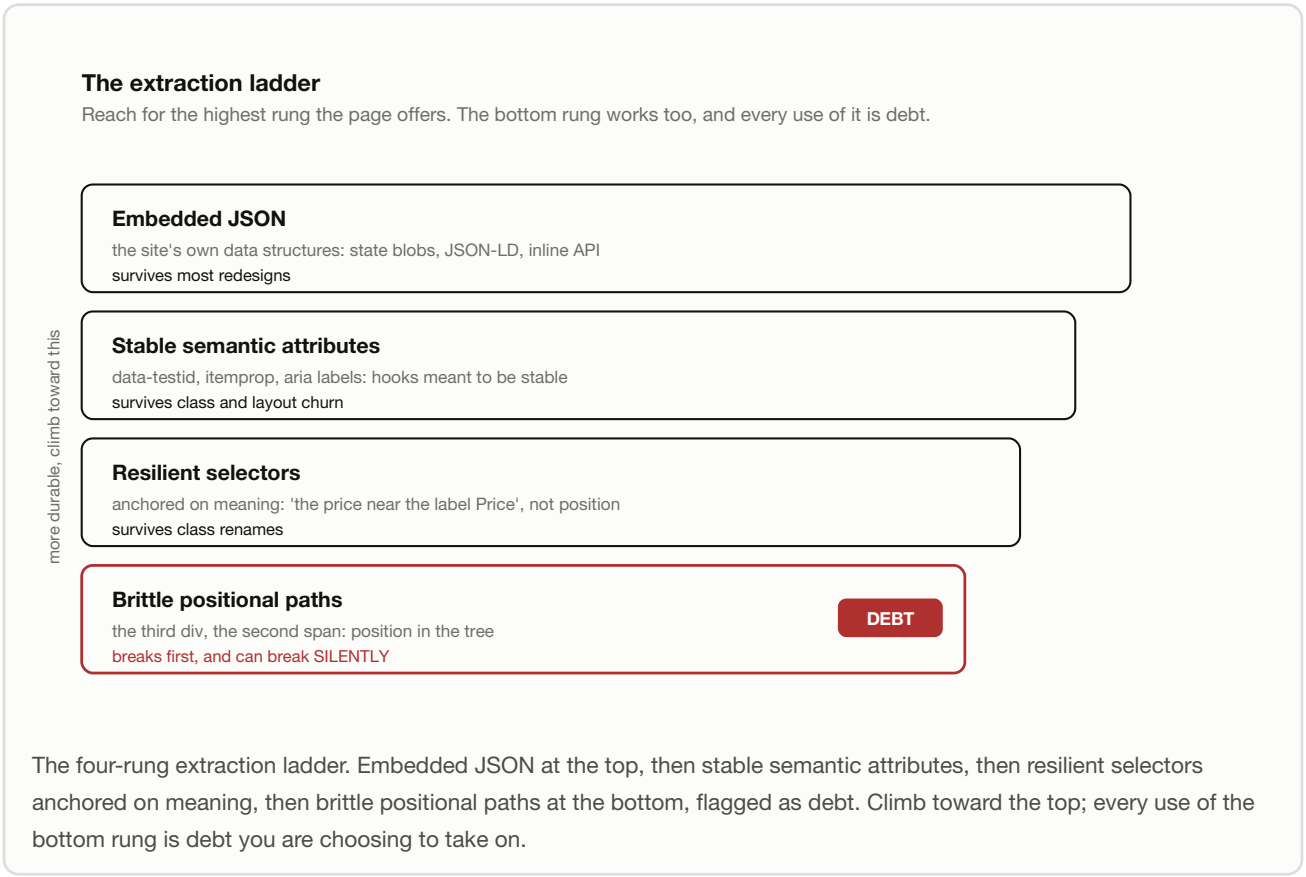


One product page with its three hidden structured sources marked: a framework state blob and a JSON-LD block in the head, and the inline data the frontend fetched. The visual HTML, drawn from all of them, is the last resort. The same numbers live in every one of these places.

I do not reach for these sources to keep the extractor tidy. I reach for them because that is how it survives a redesign. The visual HTML, the divs and spans a designer arranges, is the most redesigned thing on the internet; it changes shape every time someone tweaks the layout. The embedded data structures change far less often, for two different reasons that are worth keeping apart. The framework blob and the inline API response are the site's own plumbing, and breaking them breaks the site's own frontend. JSON-LD is a different animal: markup published for search engines, kept stable because search rankings depend on it, and for the same reason it can lag the page, carrying a list price or a stale one. When you read a price out of a framework blob instead of out of a styled span, a visual redesign sails right past your extractor, because you were reading the same source the visuals themselves read from.

The extraction ladder

Put those choices in order and you get a ladder, most durable at the top, and the rule is simple: climb to the highest rung the page actually offers.



The four-rung extraction ladder. Embedded JSON at the top, then stable semantic attributes, then resilient selectors anchored on meaning, then brittle positional paths at the bottom, flagged as debt. Climb toward the top; every use of the bottom rung is debt you are choosing to take on.

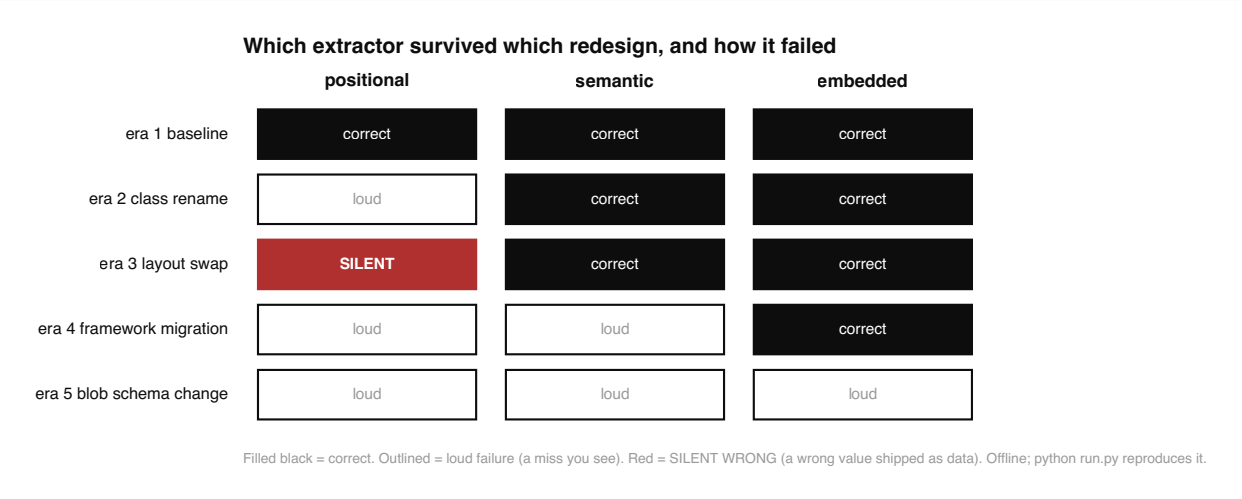
The top rung is the embedded JSON just described: the site's own data, the most redesign-proof source there is. Below it are stable semantic attributes, the hooks a site deliberately puts in its markup to be stable, things like a test identifier or a standardized data attribute; they exist so that code can grab an element without caring how it looks, which is what you want. Below that are resilient selectors: when you must read the visuals, anchor on meaning rather than position, find the price by its relationship to a label that says Price, not by its coordinates in the tree. And at the bottom is the brittle positional path, reaching for the third div, the second span, a field by where it happens to sit. That bottom rung works on the day you write it and is pure debt from then on. The experiment below is about what that debt costs when it comes due.

How extractors fail, not just whether

I have argued that the top of the ladder survives redesigns and the bottom does not. What I wanted to watch, rather than assert, is how each one fails, because failures come in two flavors that are not alike.

So the example ships the same product page across five eras: a baseline, then four redesigns modeled on the ones I see in the wild, a class rename, a layout swap, a framework migration, and a change to the shape of the embedded data itself. One fictional product, one true set of values, the same page dressed five ways. Three extractors are written against the baseline the way developers actually write them on day one, the brittle one carrying a generic price-span fallback, the defensive habit meant to survive small changes, and then run unchanged against all five eras. Each result is scored against the known truth as one of three things: correct, a loud failure, meaning it returned nothing and raised an

error you would see, or a silent wrong, meaning it returned a complete, ordinary-looking record with a value that is false. Everything is offline; `python run.py` reproduces the whole matrix.



The survival matrix from the run. Rows are the five eras, columns the three extractors. Black is correct, an outline is a loud failure you can see, and one red cell is a silent wrong: a false value shipped as a clean record. That one red cell is why this chapter exists.

The tallies came out about as the ladder predicts, and I will give them plainly. The embedded-JSON extractor got four of the five eras right and survived the most. The semantic-attribute extractor got three. The brittle positional path got one, the baseline it was born on, and broke on everything after. That is a tidy argument for reading the JSON, and not a very surprising one.

But look at the one red cell. On the layout-swap era, the brittle extractor did not crash. The redesign had put a struck-through original price into the markup ahead of the current one, both now plain price spans, and the positional extractor, told to grab the price span, grabbed the first one it found. It returned a perfect, complete, confident record for the Trailhead bottle at 32.00 dollars. The real price was 24.50. Nothing errored. Nothing looked wrong. That record would have been delivered, billed, and used to make a pricing decision against a competitor's number that was never real. It is Chapter 6's forbidden edge in another form, a wrong value recorded as a success, and it is worse than a crash for the same reason. A crash announces itself and gets fixed by lunch. A silent wrong hides inside a valid-looking feed and surfaces, if it ever does, as someone downstream asking why a decision went bad.

The next part corrected a lazy version of this argument I have made before. The embedded-JSON extractor was not magic, and it did not survive everything. On the fifth era, where the shape of the embedded data itself changed, it failed too. But how it failed is the lesson: it failed loudly. Because it was written to validate the shape it expected, a price where a price should be, it failed to find that shape and raised, returning no record rather than a wrong one. Zero silent failures across all five eras, for both the JSON and the semantic extractors. Only the brittle path lied. So do not walk away thinking that reading the JSON makes you safe, or that JSON fails loudly by nature; Chapter 3

watched a JSON parser without shape checks fail all four of its data-model mutations silently. Two habits did the work here: climbing the ladder is what survived the visual redesigns, and validating the shape before trusting a value is what turned the one failure that got through into a loud one. You need both.

Normalize on the way in

One more job belongs to extraction, done at the moment a field enters, and skipping it pushes a mess downstream that is much harder to clean once it has spread. The same fact arrives in many forms: a price as one thousand two hundred ninety nine and change might come as a European-formatted string with the comma and dot swapped, a currency as a symbol here and a code there, a date in three regional orderings, text wrapped in stray whitespace or odd encodings. Normalize each one as it comes in: a price becomes a number plus an explicit currency code, a date becomes one standard format, whitespace and encoding get cleaned at the gate. The rule of thumb I hold to is that everything past the extractor should see one consistent shape, so that no consumer downstream has to guess whether this row's price is in the same units as that row's.

Failing loudly is the point

This is the same rule as Chapter 6, and it is what the monitors in Chapter 10 will watch for. When an extractor cannot find what it came for, it must fail loudly: no empty record, no zero or blank in the field, no moving on. It must raise, so the miss lands in Chapter 6's third bucket as an ambiguous failure and becomes a gap, never a value; and because the raw response is already stored, the repair is a parser fix and a re-parse, not another fetch of a page that will fail the same way. The experiment made the stakes concrete: the extractors that raised on a schema they did not recognize protected the dataset, while the one that dutifully returned whatever it found poisoned it. An empty record that pretends to be full is the same lie as a block recorded as a price, born one step later in the pipeline.

The fetch got you bytes. The extractor decides what those bytes mean, and every field a customer sees is one of its decisions.

PART III. TRUSTING THE DATA

Chapter 8. Data Quality and Validation

A bottle that cost twenty-four dollars once went out in one of our deliveries at two thousand four hundred. About a third of the prices in that file were wrong the same way, shifted by a factor of a hundred, because a currency field upstream had changed shape. The client called it a failed delivery. By every measure the pipeline kept, it had not failed: the file arrived on time, in the right format, at the right place, with the right number of rows. The pipeline had no opinion about whether a price was believable, only about whether a row showed up. To the machine, a delivered file and a correct file were the same file. The client could tell them apart.

Delivered and correct are two different claims, and most collection systems only ever prove the first. Part II got the data in the door. This chapter is the second claim, that the data is actually right, proved the only way that counts: with a validator you have tested against errors you planted yourself, so that we validate our data stops being a feeling and becomes a number.

Quality has five dimensions

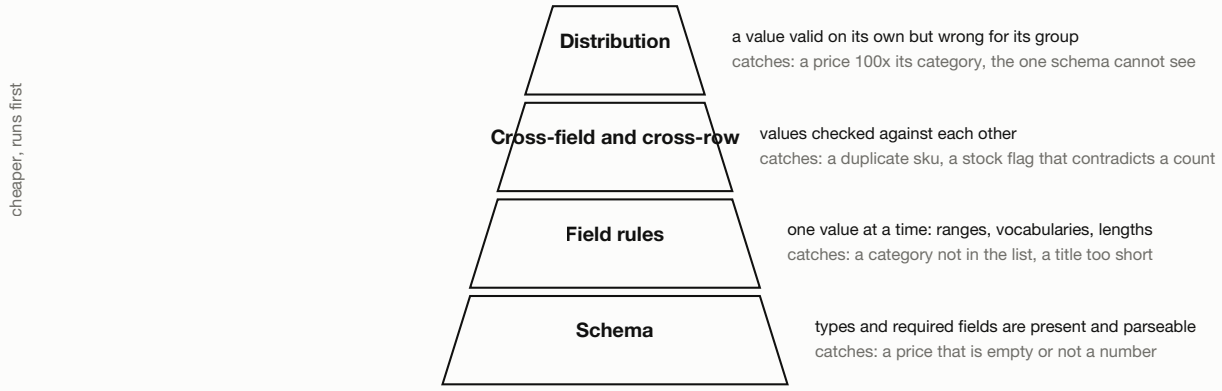
Before you can check quality you have to say what you mean by it. I use five dimensions, a teaching cut rather than a standard, and a dataset can ace four and fail the fifth into uselessness. Completeness: are the fields actually populated, or are they quietly empty. Validity: does each value have the right type and form, a price that is a number, a date that is a date. Consistency: do values agree with each other, does the in-stock flag match the stock count, does the total match the parts. Freshness: is the data from when it claims to be, or did a stale run get redelivered. And plausibility: is the value believable for what it is, a price above zero and below the sky, a rating inside its scale. My hundred-fold price error passed completeness, validity, consistency, and freshness. It failed only plausibility. Almost nobody checks that dimension, because it is the only one that requires you to know something about the world the data describes.

The validation pyramid

Turn those dimensions into checks and they stack into a pyramid, cheapest at the base. Each layer catches something the layer below it cannot see.

The validation pyramid

Four layers, cheapest at the base. Each catches something the layer below it is blind to.



The validation pyramid, four layers. Schema at the base checks types and required fields. Field rules check one value at a time. Cross-field and cross-row rules check values against each other. Distribution checks at the top catch a value that is individually valid but wrong for its group. Each layer is blind to what the one above it catches.

The base is schema validation: is every required field present, and does each one parse as the type it should be. This is cheap, fast, and where most teams stop, and stopping here is the mistake the experiment below measures. One step up are field-level rules, checking one value at a time against what it is allowed to be: a rating between one and five, a currency from a known set, a category that exists in your vocabulary, a title longer than a plausible minimum. Above that sit cross-field and cross-row rules, which check values against each other rather than in isolation: a product identifier that must be unique across the dataset, a stock flag that must not contradict a stock count. And at the top, the layer that would have caught my price disaster, is distribution validation: checking a value not against a fixed rule but against its own peers, so that a price multiplied by a hundred, now sitting far above everything else in its category, gets flagged because it looks fine on its own and absurd next to its peers. Schema would wave that twenty-four-hundred-dollar bottle straight through, because twenty-four hundred is a perfectly good number. A per-category price ceiling one layer up would catch many such cases; in this chapter's experiment, only the distribution layer did.

Fill rate is the master signal

If you build only one quality check, build this one, because it is the cheapest and the most sensitive early warning there is. Fill rate is the share of rows where a given field arrived with a real value, tracked per field, over time. It sounds trivial. It is the best detector of the most common silent failure

in collection: a selector or a parser breaks, and a field that was always there stops arriving, while everything else about the run looks normal.

One field's fill rate over thirty daily runs. It holds near 99 percent until day 18, when a selector silently breaks, and then it drops off a cliff to about 20 percent and stays there. Every run after the break still reported success; the rows kept coming, they just no longer carried a price.

Watch what the picture shows, because it is exactly the shape of a real incident. For more than two weeks the price field is populated on ninety-nine percent of rows. Then on day eighteen something upstream shifts, the extractor that used to find the price finds nothing, and from that day forward the field is populated on one row in five. Nothing else changed. The job still ran, still finished, still delivered its usual row count, still reported success, because Chapter 6's discipline held and the missing prices were recorded as gaps rather than invented zeros. But the value the customer is paying for fell off a cliff, and the only thing that saw it was the fill rate. A per-field fill-rate monitor turns that cliff into a page to a human on day eighteen. Without it, the first person to notice is the customer, three weeks later, asking why most of their prices are blank. Chapter 10 is about monitoring like this. Here, fill rate is where you start, because it costs almost nothing.

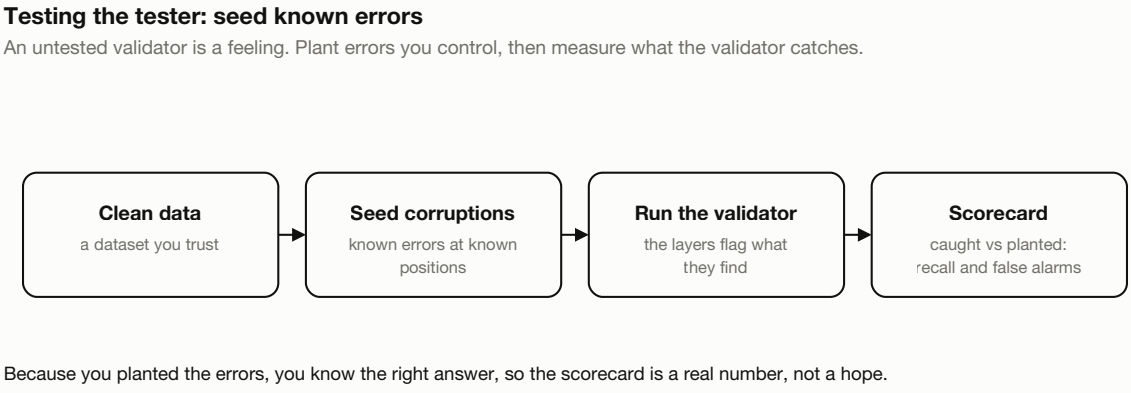
Plausibility is where domain knowledge lives

The upper layers of the pyramid are where you encode what you actually know about the data, as executable rules. A price is above zero and below some ceiling that makes sense for the category. A date falls inside a sane range, not in 1970 and not next century. A discount is between zero and a hundred percent. A category is one of the values you know exist. Each of these is a small piece of domain knowledge, the kind that lives in an experienced analyst's head, written down as a check a machine runs on every row. Keep writing them down as you learn them. Every plausibility rule came

from some past bad batch, and writing it down is what stops the same batch shipping silently a second time.

Seed errors and count

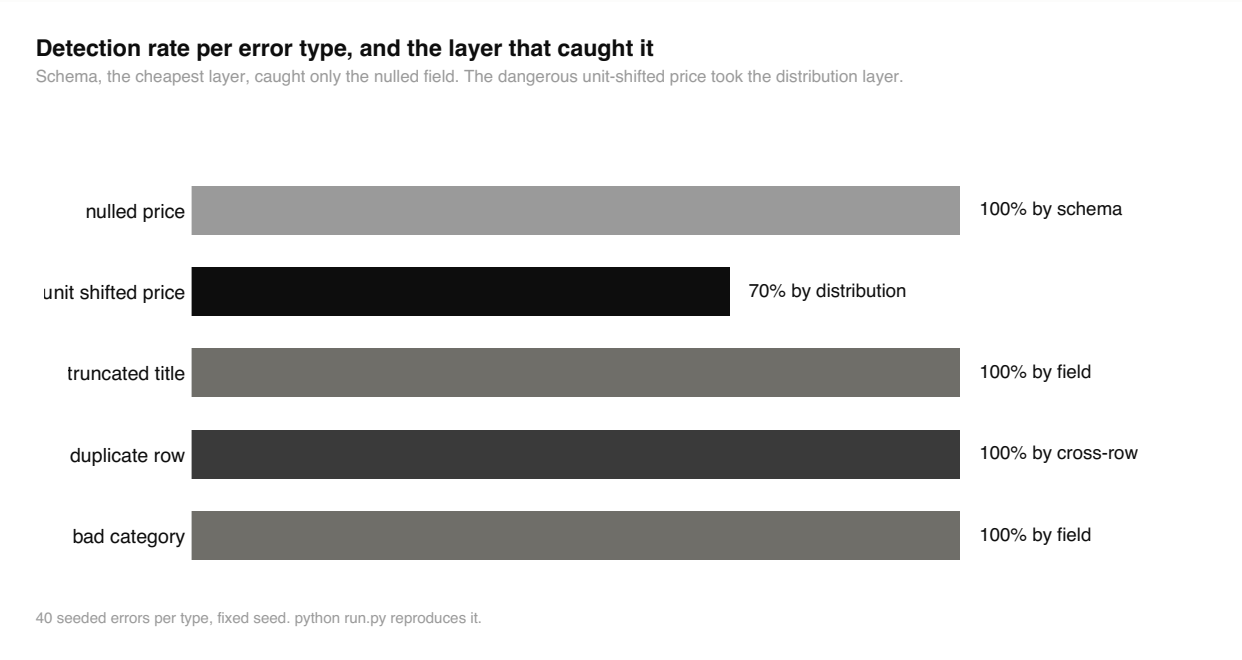
You have a validator. How good is it, as a number rather than a feeling? The only way to answer is to hand it errors you planted yourself, at positions you recorded, and count how many it catches. This is error seeding, and it is how a validator becomes a control instead of a comforting story.



The error-seeding loop. Start from clean data, inject known corruptions at known positions, run the validator, and compare what it flagged against the truth you planted, to get a precision and recall scorecard. Because you planted the errors, you know the right answer.

So the example does exactly that. It ships a synthetic but realistic catalog of ten thousand products, seeds five kinds of error that I have watched happen in production, forty of each at known rows, and runs a four-layer validator over the result. Five error types: a nulled-out price, a price unit-shifted by a factor of a hundred, a truncated title, a duplicated row, and a category value outside the known vocabulary. Then it scores each error type by how many the validator caught, and, just as important, notes which layer of the pyramid did the catching. Everything is seeded from a fixed seed, so `python run.py` reproduces the whole scorecard.

The run made the case better than I had. The schema layer, the cheap base of the pyramid where so many teams stop, implemented here as the usual type and presence checks, caught exactly one of the five error types: the nulled price, which failed to parse as a number. That is all. It waved through the truncated title, the bad category, the duplicate, and, most dangerously, the hundred-fold price error, because every one of those is, to a schema check, a perfectly valid-looking value. One error type in five. The layers above it, the field rules, the cross-row check, and the distribution check, caught the other four between them. If you had only schema validation, which is the real state of a great many pipelines I have seen, you would have shipped four of these five error types without a whisper.



The scorecard from the run. Detection rate per error type, each bar labeled with the layer that caught it. Schema caught only the nulled price. The unit-shifted price, the dangerous one, was caught only by the distribution layer, and only seven times in ten.

Two dents in the result, because a scorecard that reports only its wins is the dishonesty I am arguing against. First, the distribution layer caught the hundred-fold price error, the one that mattered most, but it caught only seventy percent of them. The other thirty percent slipped through for a plainer reason: the check measures how far a log price sits from its category's center in units of that category's own spread, and in the widest categories a hundred-fold jump did not always reach the cutoff the run used. Loosen the cutoff and you catch them, at the price of more false alarms; distribution checks are a dial, not a wall. Seventy percent caught is far better than the zero percent schema managed, and it is not a hundred, and I would be lying if I told you it was.

Second, the title-length rule caught every single truncated title, all forty, which sounds like a triumph until you count what else it flagged. It also raised forty-three alarms on titles that were not seeded errors at all: real, complete, two-word product titles like a short brand name and a three-letter noun, that happened to fall under the minimum length the rule enforced. On inspection, every one of those forty-three was a false alarm, a legitimately short title the blunt rule could not tell from a truncation. That gives the title rule a precision of forty-eight percent: fewer than half the rows it flagged were actually wrong. That forty-eight percent was invisible until I seeded errors and counted, and that is why seeding is worth the trouble. A validator that flags things feels like it is working. Only when you know the true answer can you see that it was crying wolf more often than not, and decide whether to live with the noise, tune the threshold, or make the rule smarter. The follow-up is a real engineering choice, and you cannot even have the conversation without the number.

That is why you test the tester. The seeding turned three vague good feelings, we check our schema, we validate titles, we catch outliers, into three facts: schema catches one problem in five, the outlier check misses three price errors in ten, and the title rule is wrong more than half the times it complains. None of those facts is visible from reading the validator's code or watching it run on data whose errors you do not already know. They only appear when you plant the errors yourself.

The quality report ships with the data

All of this rolls up into one artifact I now attach to every delivery: a one-page quality report that travels with the data itself. Per-field fill rates for this batch against the last one. The count of rows flagged by each validation layer, and what the flags were. The freshness of the underlying collection. A short note on anything the validator caught and anything it is known not to catch. It costs almost nothing to generate, because the validator already computed every number in it, and it changes the entire conversation with whoever consumes the data. Instead of them discovering a problem in production and calling it a failure, they see the shape of the batch before they act on it, and the rare bad batch gets caught at the door by a person who was handed the evidence. I can stand behind a dataset that ships with its quality report. A bare file is a hope with a filename.

That is as far as row checks go, and they have a blind spot. Every check here operates on rows that exist. A validator can tell you the prices you collected are believable, the categories valid, the titles complete. It has nothing to say about the products you never collected at all, because absence leaves no row to test. You have proven that what you have is correct. Whether you have all of it is a different question, and the full list is the one thing you cannot see.

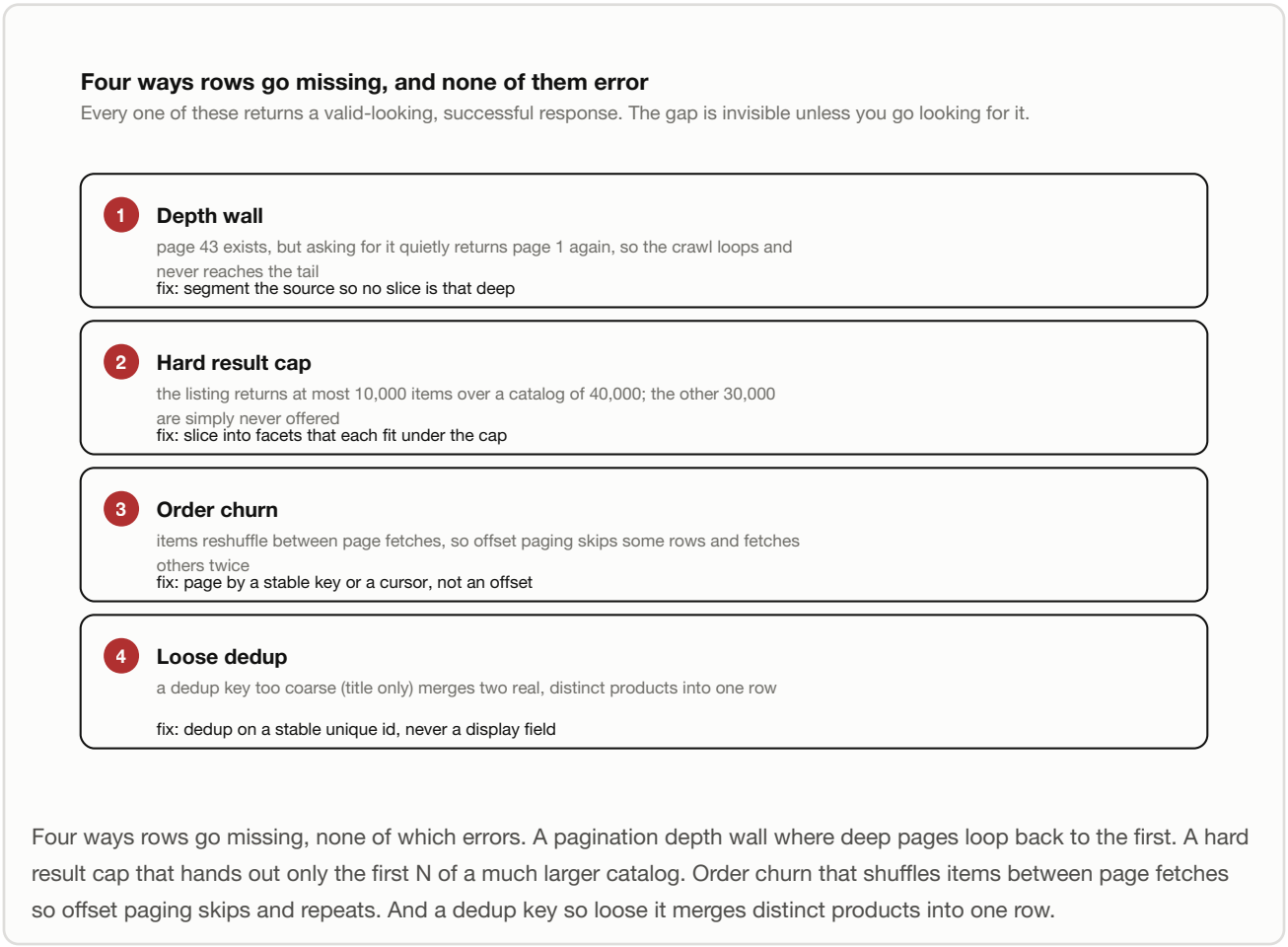
Chapter 9. Coverage: Collecting Everything

Everything so far has been about the rows you collected. This chapter is about the rows you did not, and it is harder, because a missing row leaves no trace. A wrong price sits in the data where you can find it and argue with it. A product that was never collected is simply absent, and absence is invisible. No validator from the last chapter can flag a row that is not there. The file looks complete, the schema passes, the counts look big, and a third of the catalog is gone.

The worst coverage failures look like successes. The scrape finished, every request returned a normal response, the row count looked healthy, and nobody saw the hole. The question hiding in that hole is "did we get all of it?", and the rest of this chapter is how to answer it with a method and an estimate instead of a shrug.

Where rows go missing

Rows rarely vanish dramatically. They leak, and four failure modes account for most of the leak, all sharing one property: the collector terminates cleanly and reports success while missing data. There is a fifth that hides even better, a scope narrowed before you began, a default filter or a region setting that shrank the universe you thought you were collecting, and the coverage contract later in this chapter is where you pin the scope down so that one cannot bite. Learn to see the four, because your monitoring will not.



Four ways rows go missing, none of which errors. A pagination depth wall where deep pages loop back to the first. A hard result cap that hands out only the first N of a much larger catalog. Order churn that shuffles items between page fetches so offset paging skips and repeats. And a dedup key so loose it merges distinct products into one row.

The first is the depth wall. A site serves the first several pages normally and then quietly stops: ask for page 43 and it hands back page 1 again, or an empty page, or the last page it is willing to serve. Your crawler, paging happily, loops or halts and never reaches the tail, and every one of those requests returned a valid page. The fix is to never trust a page count you were given, and to probe the wall on purpose, one careful request a little past where you expect the end, checking whether its contents are actually new. Probe gently: a request for page one million forces the source to scan its whole index, which is neither fair load nor a good way to stay unblocked.

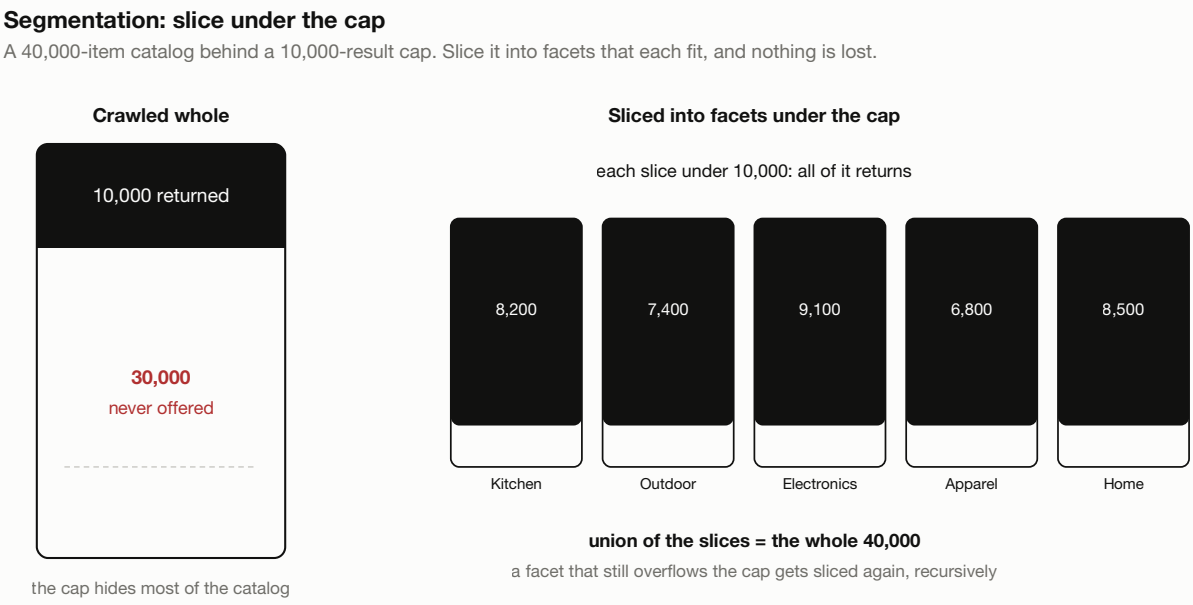
The second is the hard result cap, and it is so common it is nearly a law of the web. A listing announces a million results and the backend will only ever hand out the first N, almost always a round number. The reason is concrete and worth knowing. The default result window in Elasticsearch, the search engine behind a great many storefronts, is exactly 10,000, so deep pagination past the ten-thousandth result is refused outright, and many deployments also stop reporting an exact total past that window, which is why "10,000+" is so common a sight. Other backends draw the line elsewhere and just as hard. eBay's finding interface caps at 100 results per page and 100 pages, exactly 10,000 per query. The GitHub search interface stops at 1,000. A common maps interface returns at most 60 places per query, however many thousands exist. If the number of rows a query actually returned lands on a suspiciously round number, treat it as the cap, not as the size of the catalog.

The third is order churn. If items reshuffle between your page fetches, and on a busy catalog they do, then paging by position skips some rows and fetches others twice. The duplicate rate is your only visible symptom, which is why duplicates are worth watching rather than silently discarding. The fix is to page by a stable key or a cursor rather than a numeric offset. When the site offers only page numbers, buy back the same safety by overlapping your pages, striding less than a page at a time, and deduping the overlap on a stable key.

The fourth hides inside a habit that feels like hygiene: deduplication. A dedup key that is too loose, matching on a display title rather than a stable identifier, quietly merges two distinct products into one row and calls it cleaning. The drop counter reads like tidiness and is actually a coverage-loss counter. Dedup on a real unique identifier, never on a field a human reads.

Segmentation: slicing under the cap

The cap is the killer you beat with arithmetic, and the tool is segmentation: if the whole catalog will not come through one query, slice it into facets that each fit. Category, brand, price band, geography, date; any attribute that partitions the catalog into pieces small enough to clear the ceiling.



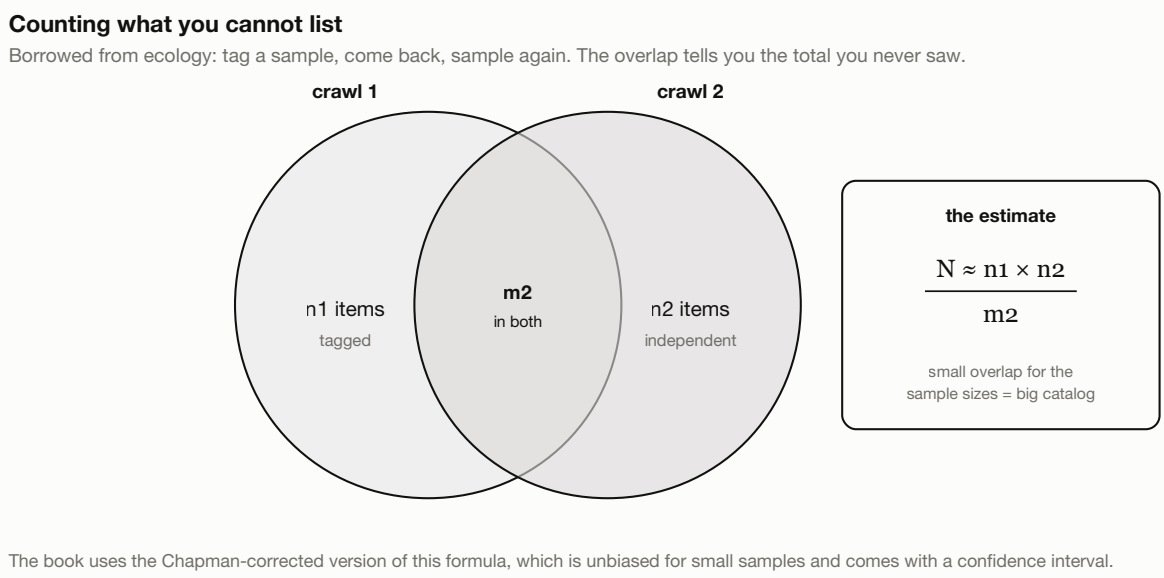
A 40,000-item catalog behind a 10,000-result cap. Crawled whole, only 10,000 come back and 30,000 are simply never offered. Sliced into category facets that each fall under the cap, every slice returns in full, and the union of the slices is the whole catalog.

Picture a catalog of 40,000 items behind a 10,000 cap. Crawl it whole and you get 10,000, with 30,000 you never even knew to miss. Read the site's own category counts first, slice into facets each under 10,000, collect each in full, and stitch them back together on a stable identity key. The union is the whole catalog only if the facets actually cover it, so check that rather than assume it: real category

systems let an item live in two facets or none, so compare the distinct keys in your union against an independent count before you believe you are whole. Two disciplines make the slicing trustworthy. When a facet is itself over the cap, split it again, halving a price or date range until every piece fits. That terminates as long as no single value on the axis holds more than the cap; when a pile of items shares one price or one brand with nothing left to bisect, switch to a different axis or accept a residual over-cap slice and flag it as incomplete. And treat any slice that returns exactly the cap as unresolved rather than finished: a count that lands precisely on the ceiling is a truncation warning to subdivide or verify, not a total to book, however plausible the page makes it look.

Counting what you cannot see

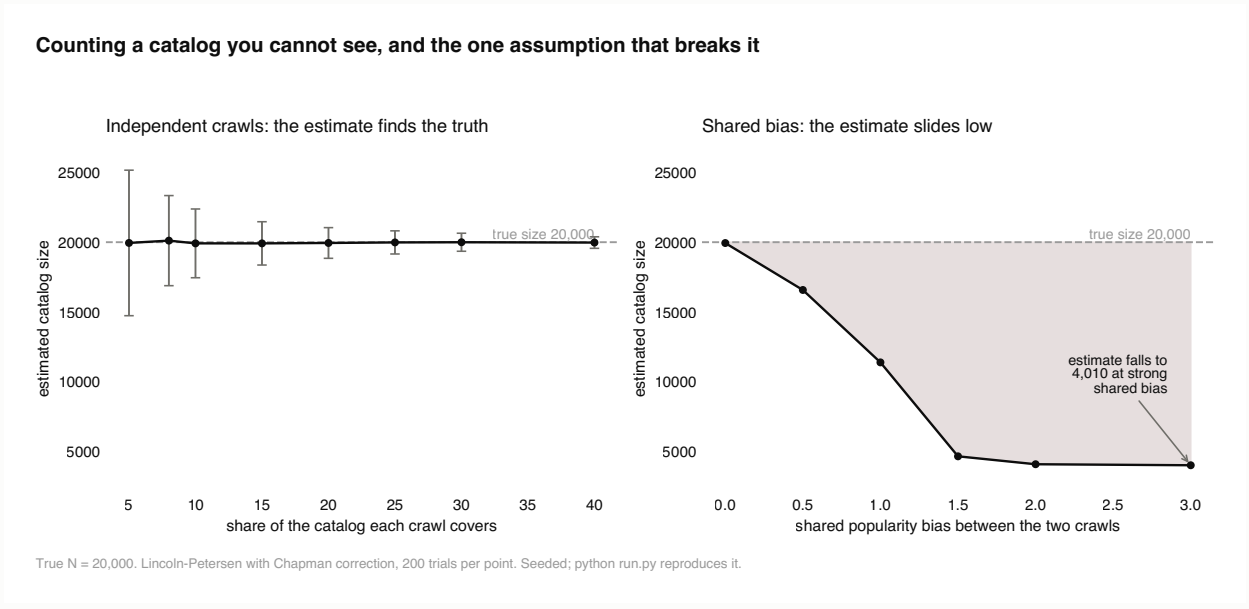
Segmentation gets you more of the catalog. It does not tell you how much you are still missing, and for that you need to estimate a total you can never enumerate. The method is borrowed from ecology. To count fish in a lake you cannot drain, you net a sample, tag them, throw them back, come another day and net a second sample. The fraction of the second catch that carries a tag tells you how big the lake's population must be: if few of your tagged fish come back, the lake is large; if most do, it is small. Catalogs make this easier than lakes, because you never have to tag anything. Every item already wears a stable identifier, a product code or a canonical link, so you crawl the catalog twice by independent routes, record the identifiers each pass saw, and the items that turn up in both passes are your recaptured tags.



The capture-recapture idea. The first crawl tags a sample of items, a second independent crawl tags another, and the overlap between them, the items seen in both, sets the total. A small overlap relative to the sample sizes means a large catalog.

The United States grades its own census this way. After the 2020 count, the Census Bureau ran a second, independent survey and used the same two-sample overlap idea, dual-system estimation, to measure how good the count had been. The stakes are why it bothers: on the order of a trillion dollars a year in federal funding rides on census numbers, so knowing how far off they are is worth a great deal. Decades earlier, two researchers estimated the size of the entire indexable web the same way, by measuring how much different search engines' results overlapped, and put a floor under it of 320 million pages. The same method will count a product catalog.

The estimator this chapter uses is the Lincoln-Petersen formula with a small-sample correction, the Chapman estimator, which removes nearly all the bias the raw version carries, and it comes with a confidence interval so the answer arrives knowing how firm it is. Two assumptions ride under it and both matter for catalogs. The catalog must hold still between the two crawls, so run them close together, ideally the same day, or the estimate lands on whatever size the catalog was at the second pass. And the method can only ever size the part of the catalog your two routes could reach; items no route can touch, gated behind a login, bound to a region, linked from nowhere, sit outside the number entirely. The example puts the estimator to the test on a synthetic catalog whose true size is known only to the grader, with each crawl drawing at random, so every estimate can be marked right or wrong.



Two panels from the run, the estimator graded against a true catalog of 20,000. Left, with independent crawls, the estimate sits on the true line at every sample size and its interval tightens as the crawls grow. Right, when both crawls share the same popularity bias, the estimate slides far below the truth.

The good news first, on a simulated catalog of 20,000 items with each crawl drawing at random. Two independent crawls covering just a fifth each estimate its size well. Repeat the experiment many times and the estimates average within a third of a percent of the truth, so the estimator carries almost no bias. Any single pair of crawls is looser than that, and candid about it: the 95 percent interval is about

plus or minus 5.5 percent, and it contains the truth 94 percent of the time, close to the 95 it promises. Even at a fifth each you can bound a catalog you never saw whole, and the interval tells you how tightly. Crawl more and it shrinks; at 40 percent each the interval is down to about 2 percent. The estimate is trustworthy, and the interval tells you how far.

Then the run breaks it on purpose, because the method rests on a third assumption that catalog crawling loves to violate: the two crawls must be independent. In the experiment both crawls are given the same popularity bias, favoring the same easy-to-reach items, which is exactly what happens when you crawl the same relevance-sorted list twice or route both crawls through the same discovery path. Independence is gone, and the damage is severe. Both crawls keep finding the same popular items, so the overlap inflates, and an inflated overlap says the catalog is small. At a strong shared bias the estimate collapses from 20,000 to about 4,000, an 80 percent underestimate, and its confidence interval now contains the truth zero percent of the time. Sit with that last number. The estimate is wrong and sure of itself, reporting a tight interval around a number that is nowhere near right. The reason is that the overlap sits in the denominator of both the estimate and the interval, so a fat overlap pulls the estimate down and pulls the interval tight at the same moment. The formula did what it was asked. Independence was false, and the interval had no way to know.

Follow that underestimate one step further, because its business meaning is the trap. If the method says the catalog is smaller than it is, then the haul you actually collected looks like a larger share of it than it is, and your coverage number comes out flattering. Worse, this is the common direction on the open web. Search rank, sitemap freshness, category placement, and internal links all favor the same popular head items, so any two crawls that lean on the site's own structure lean the same way and agree to overstate how complete you are. The error that feels comfortable is the dangerous one.

There is a cheap check for this, and it costs one line of arithmetic. Under true independence the overlap should land near n1 times n2 divided by the catalog size. Take a rough size you trust to be roughly right or a little low, and compare. If the overlap you observed is well above what that predicts, say half again or more, then either your two crawls are correlated or the true catalog is far smaller than your rough figure, and both readings forbid trusting the estimate. An overlap far below the prediction is its own warning, of crawls that repel each other and overstate the catalog. Run the check, in both directions, before you believe any capture-recapture number.

So the practical rule the experiment points to is that the two crawls must reach the catalog by different routes. Different discovery mechanisms above all, one by category walk and one by sitemap or a public interface, then different sort orders and segmentation axes, enough that the items one crawl finds easily are not the same items the other finds easily. Two crawls that share a proxy pool, a login, a geography, or a discovery path share their blind spots, and shared blind spots are what turn this estimator into a confident lie. Different routes reduce the bias; they do not abolish it, because items that are hard for everyone to reach stay hard for both crawls, so treat the corrected number as a lower bound on the catalog, not a fact. And re-running the same crawler a week later is no second sample, only a reliability test, and it will report near-total overlap and tell you that you have everything.

The second pass can also be small, which matters because this book never stops caring about fair load. It does not have to be a whole second crawl; a modest independent probe carries enough overlap to size the catalog, as long as the overlap is actually there. Size the probe from the arithmetic before you run it, since expected overlap is n1 times n2 over the catalog size, and if that lands in single digits the estimate will be noise. For a fixed request budget the tightest interval comes from two equal passes, so if grading coverage is the goal, split the budget in half. If collecting the most items is the goal instead, a large main crawl and a small probe gather more of the catalog while still buying a rough bound, and that is a real choice, not an oversight.

The two techniques answer different halves of the question. Segmentation locates items, pulling more of the catalog through the caps, but it can never tell you when you are done. Capture-recapture sizes the gap, telling you what fraction is still missing, but it never tells you which items those are, and it sizes only the catalog your two routes could reach. Segmentation to collect, capture-recapture to grade the collection. That is how you answer "did we get all of it."

Coverage in production

A single estimate answers "did we get it all today." Production needs a standing answer, because coverage rots on its own: sites restructure, caps change, a category stops rendering, and a slice drops out between one delivery and the next with nothing in the logs. Three cheap habits keep the answer current.

Reconcile distinct keys, not raw rows, against the site's own claimed counts, and test for a stable ratio rather than equality. Raw row counts lie in both directions, rising on duplicate pagination while true coverage falls; distinct identifiers do not. And the site's number is itself a second noisy measurement, capped and rounded and sometimes personalized, so the accurate statement is never "we match the site" but "our count and the site's sit in their usual ratio." A ratio that suddenly moves is the alarm. A third independent number sharpens it: a site's sitemap files, capped by protocol at fifty thousand URLs each, give a count that neither the listing pages nor your crawl produced, and three numbers that usually agree catch a problem two never would.

Plant sentinels: a fixed panel of known items that must appear every run, so their disappearance is a coverage alarm you do not have to compute. The power is worth knowing, limits and all. A panel of 50 sentinels catches a uniform 5 percent loss about 92 percent of the time in a single run. A 1 percent loss it catches only about 40 percent of the time in one run, though that climbs toward 92 percent across five runs. Two conditions hide in those numbers. They assume the loss is spread evenly, and it rarely is, so a panel scattered at random can hold nothing from the one category that broke; weight the panel toward the segments you most need to be sure of. And the five-run climb assumes the loss reshuffles run to run, as flaky fetches do; a permanent structural gap either sits in a sentinel from the first run or never will, so it shows immediately or not at all. Sentinels catch a real loss cheaply, and they promise nothing about the corners.

Watch coverage per segment, not just in total, because the total is the weakest signal you have and the one most teams rely on. Losses are almost never spread evenly; they concentrate in the corner that broke, one category, one region, the newest and fastest-changing items, which are usually the commercially valuable ones. I have shipped a feed that came in two percent light overall, comfortably inside the band, while one whole category had silently dropped by forty percent. The invoice said complete, the customer priced their own decisions on the delivery, and the missing forty percent surfaced weeks later not as a scraper bug but as a wrong business call someone had already made on it. The total hid the gap. A per-segment view shows it at a glance, and it is the difference between finding the hole yourself and having a customer find it for you.

One caution, because it cuts the other way. A big overnight drop is not always your bug. Real catalogs do cliff: in 2016 a well-known supermarket pulled a major supplier's brands from its site overnight in a pricing dispute, and dozens of brands vanished at once. A collector that cried failure that morning would have been wrong; the site really had changed. And a drop of a few dozen items in a feed of forty thousand is a rounding error against the total while being a total wipeout of that one supplier's segment, which is the case for watching segments rather than the sum. Coverage monitoring has to tell a collection regression from a real catalog change, which is why you reconcile against the source instead of only against yesterday.

The coverage contract

All of this rolls up into how you state coverage to whoever pays for the data. Do not say "we scraped the whole site." There is no method behind that claim, and without a method it is a hope. State four things instead. The scope: exactly which universe you set out to collect. The method: how you collected it and how you segmented to beat the caps. The two independent routes you crawled, with the result of the overlap check, so the estimate's own assumption sits on the record. And an estimated completeness with its basis. For a category that might read: an estimated 96 percent, by two-sample capture-recapture over a category walk and a sitemap, 95 percent interval 94 to 98 percent. That is a number a buyer can weigh, and a number a serious buyer will ask for. The buyer's move is to write it into the agreement as a term, not a courtesy: a completeness floor, a cadence for re-measuring it, and a remedy when it slips.

Two riders belong in that contract, and both come straight from the experiment and the research behind it. First, a completeness percentage is a population estimate, never a promise about any one item; 96 percent coverage is fully compatible with missing the single product a buyer cares most about, so an aggregate number and a spot-check of the items that matter are different assurances and a buyer should want both. The census makes the point at national scale: its 2020 net national miss was about a quarter of a percent, not even distinguishable from zero, while one major demographic group was undercounted by nearly five percent. A blended figure can hide a segment gap large enough to matter, which is why coverage is always reported per segment, never as one comforting number. Second, coverage estimates expire. A figure measured at onboarding and never rechecked is a stale

claim the day a site restructures, so tie the re-measurement to the commercial rhythm the reader already has: every delivery or billing cycle ships with a coverage number dated inside that cycle, and a number that stops moving is a number nobody is checking.

The buyer's side of this is one question, and it separates a real data operation from a hopeful one: how do you know that is all of them, and what is your estimated coverage with its interval. "How many rows" does not answer it. Volume without a denominator says nothing. Ten million records is a big number attached to an unknown fraction, and a smaller dataset with a measured 97 percent is worth more for any use that leans on the catalog being representative, which is most uses worth paying for. That measured denominator also changes how a buyer shops. It turns a vendor comparison from price per row, which rewards padding, into price per percent of the universe covered, which rewards the thing you actually want. Watch the scope line while you do it, because a coverage percentage is trivially inflated by shrinking the universe it is measured against. The number that means anything is completeness against the scope you were promised, not against whatever was easy to reach. Coverage is a fraction, and you estimate the denominator instead of pretending it does not exist.

One more thing can go wrong after all of this. It was all true when you checked, and it stopped being true afterward, because the source changed without telling you. Sources do that, all of them, and the next problem is catching it from the data itself before a customer does.

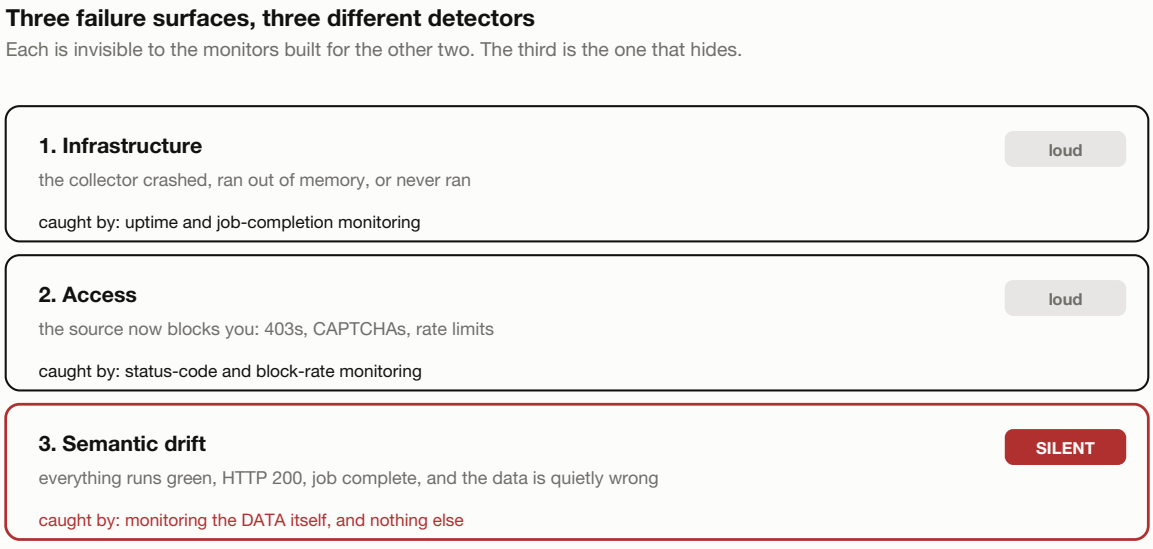
Chapter 10. Monitoring, Drift, and Healing

A source I collected for years redesigned its product pages on a Tuesday. I found out the following Tuesday, from the customer. For seven days the collector had run every night without a hiccup: every request came back 200, every job finished, every dashboard was green. And every night it had delivered a catalog of products with no prices, because the price now lived in a node my selector no longer matched, and the code, finding nothing, wrote nothing. Seven days of clean, complete, on-time deliveries of garbage. Nothing had broken in any way my monitoring could see, because I was monitoring the pipe, not the data inside it.

That week taught me the last lesson of this part of the book. You can collect the data, prove each row correct, and estimate how much you have, and still wake up shipping wrong data, because a source changed under you and told no one. Every source will. Catching that from the data itself, fast, before a customer does, starts with admitting that the dashboards you already have are looking in the wrong place.

Three failure surfaces

A collector fails on three distinct surfaces, often separately and sometimes stacked, and each is blind to the monitors built for the other two. Keep them separate, because the fix for each is different.



Three failure surfaces. Infrastructure: the collector crashed or never ran, caught by uptime and job monitoring, and loud. Access: the source now blocks you with 403s or CAPTCHAs, caught by status-code and block-rate monitoring, and loud. Semantic drift: everything runs green and the data is quietly wrong, caught only by monitoring the data itself, and silent.

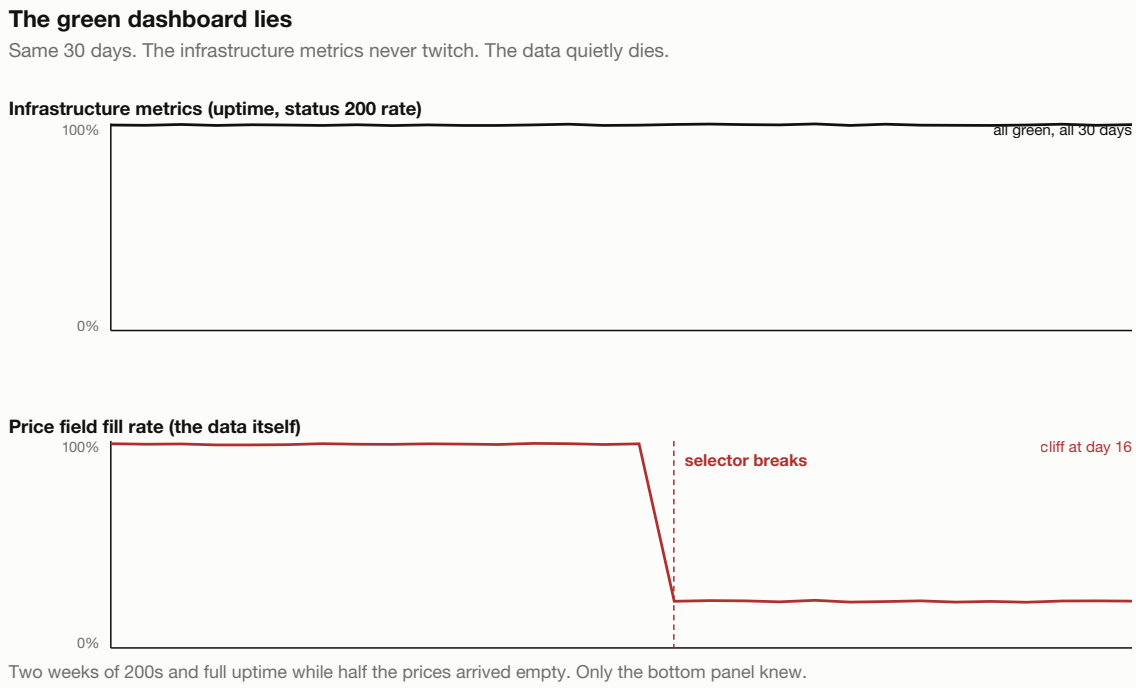
The first is infrastructure. The program crashed, ran out of memory, threw an exception, or, the sneakiest version, never started because a schedule quietly failed. This surface is loud and well served by ordinary monitoring: an exit code, a missing heartbeat, a job that did not complete.

The second is access. The source started refusing you, with a spike of 403s or rate-limit responses, a challenge page, a redirect to a sign-in wall, or a sudden collapse in response size as a real page turns into a short error. Also loud, if you watch for it: status codes and block rates catch it.

The third is semantic drift, and it is the one that ended my week. The program is healthy, so infrastructure monitoring is green. The source served a real page with a 200, so access monitoring is green. The job extracted its rows and finished. And the data is wrong, because the page changed shape and the parser is now reading the wrong thing or nothing at all. No infrastructure or access monitor can see this, because both watch the transport and the transport worked perfectly. Only a monitor that looks at the content can catch it, and most teams never build that one.

Why the green dashboard lies

The reason is structural, not a shortcoming you fix with a nicer dashboard. Every standard operational metric, uptime, latency, error rate, status mix, queue depth, is computed on the pipe: did the request go out, did a response come back, how fast, with what code. None of them is computed on the meaning of what came back. A request that succeeds and returns a page whose price field is empty is, to every transport metric, a complete success. The dashboard is answering a different question than the one you care about, and answering it correctly.



Two time series on one thirty-day axis. On top, the infrastructure metrics stay flat and healthy the whole month. On the bottom, over the same days, the price field's fill rate holds near 100 percent and then falls off a cliff when the selector breaks. The dashboard was green the entire time the data was dying.

So the rule is simple, and most operations I meet violate it: monitor the data, not just the pipe. A green transport dashboard is never evidence of good data. Treat data quality as its own surface, watched by its own signals, or you are watching a smoke detector wired to the wrong room.

Signals from the data itself

The good news is that the data-quality signals are cheap, and you met the most important one in Chapter 8. Per-field fill rate, the share of rows where a field arrived with a real value, is one counter per field per run and the most useful drift detector I know. My Tuesday break would have shown as the price fill rate falling off a cliff, on the day it happened.

But fill rate has a blind spot, the most dangerous false comfort in monitoring, so never read a full fill rate as health. A large class of breaks keeps a field 100 percent populated while its meaning silently changes. A site starts emitting prices in cents instead of dollars and a median of 49.99 becomes 4999. A locale flips and the parser reads a European 1.299,00 as 1.299. A rating field collapses to a default and every row reads 0.0. In each case the field is present, non-null, and completely wrong, and fill rate never twitches. This is Chapter 7's silent-wrong failure and Chapter 6's forbidden edge, arriving now as a monitoring problem.

Catching it needs one more signal: watch the distribution of a field's values, not just whether they are present. Store a small fingerprint of each field each run, a handful of quantiles or a histogram binned

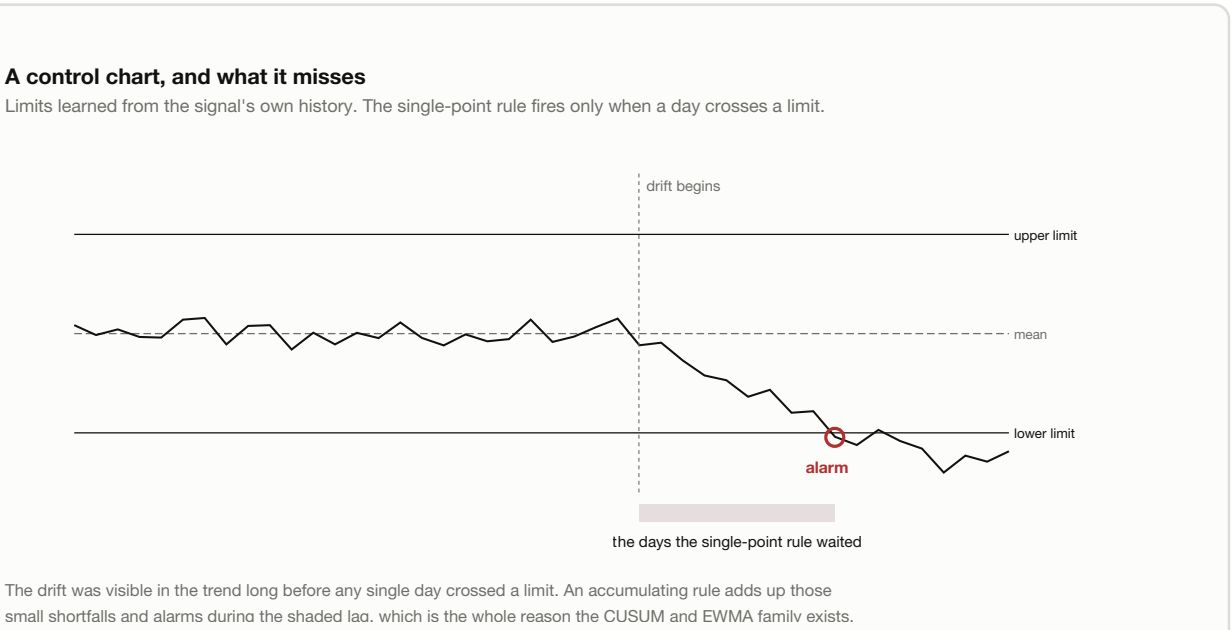
against a fixed baseline, and compare this run's shape to the baseline's. The cents shift jumps every quantile by a factor of a hundred. The collapse-to-default drops the field's variety to a single value. A standard tool for the comparison is the population stability index. It scores how far this run's distribution has moved from the baseline's, weighting each bin's change by the log of its ratio. A rough convention: below 0.10 is noise, 0.10 to 0.25 is worth investigating, above 0.25 is a real shift to act on. Reach for a library that computes it rather than hand-rolling a plain distance. Those thresholds are calibrated to that specific weighting, and the index misbehaves when a baseline bin is nearly empty or a new value lands outside every baseline bin, cases a good implementation guards against and a quick one does not. One caution that saves you from a monitor nobody trusts: do not alarm on the p-value of a statistical test. At scraping volumes any tiny difference is statistically significant, so a p-value alarm fires constantly. Alarm on the size of the shift, not on whether a test could detect it.

Two more disciplines make these signals trustworthy. Watch them per segment, never only in aggregate, because a run with the right total row count can hide one category that collapsed while another grew: a high-volume category holding steady masks a low-volume one whose page template broke and whose fill rate went to zero, and the blended number never twitches. And build baselines from statistics that shrug off a bad day, a median and a scaled median absolute deviation rather than a mean and a standard deviation, because a single bad run poisons a mean-based baseline and then either mutes real alarms or floods you with false ones afterward. Scale the median absolute deviation by about 1.48 to make it stand in for a standard deviation, or the limits come out half as wide as they should and the alarms flood.

Alert design

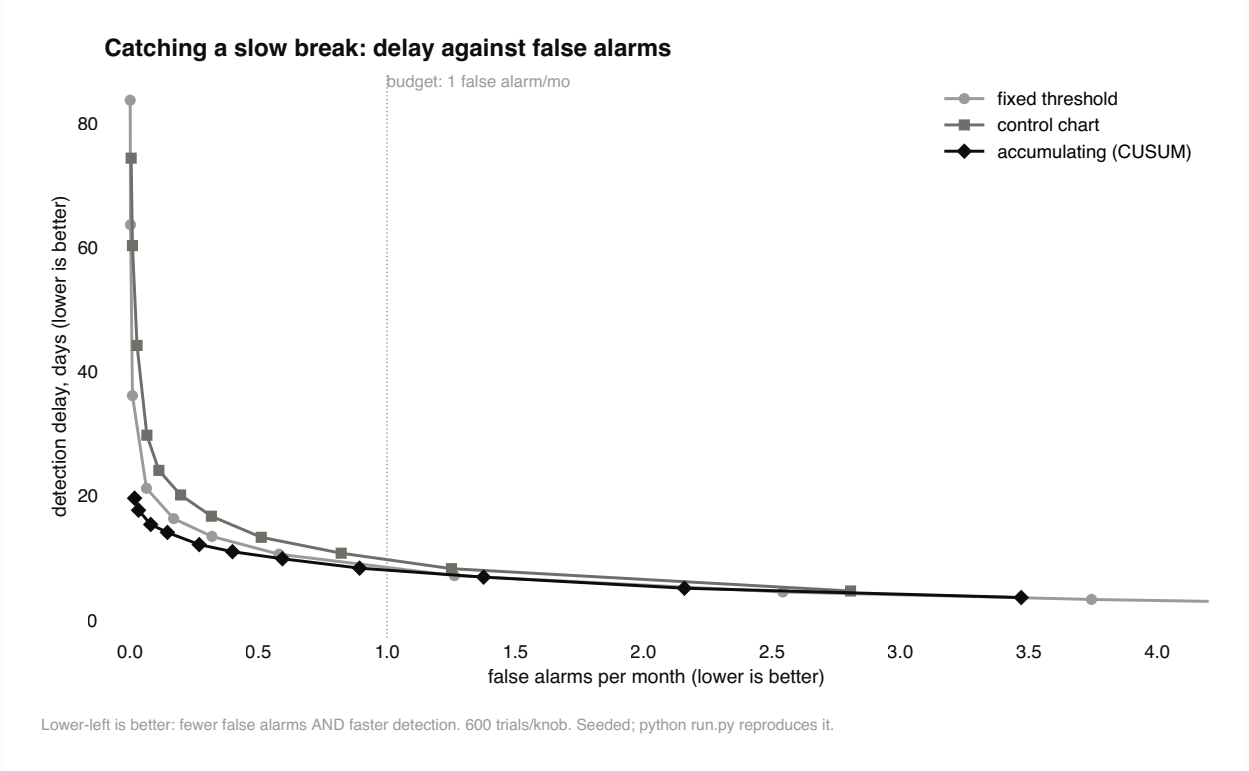
Once you have a signal, you have to decide when it fires, and this is where most monitoring is bad. The naive choice is a fixed threshold: alarm when the fill rate drops below some line. It is either too jumpy, set close enough to catch small problems that normal noise trips it, or too slow, set far enough out to avoid false alarms that a real degradation has to become severe before it crosses. A history-aware rule does better by learning the signal's normal wobble from its own past, which is the control chart: alarm when a day falls more than a few standard deviations below the baseline the signal itself established.

But even the control chart has a weakness, and it is the one that matters most for scraping. The basic control chart judges one point at a time against its limits, so it is built to catch a sudden jump. A slow drift, a redesign rolling out to more pages each day, moves the signal down in steps too small for any single day to cross a limit, and the one-point rule waits, and waits, until the drift is finally large enough. A family of accumulating rules exists for this case. A rule like CUSUM keeps a running sum of each day's small shortfall below normal, floored at zero so a good stretch cannot bank credit against a future break, and fires when the sum builds up, so it catches a consistent gentle drift while every individual day still looks fine.



A fill-rate control chart with a learned mean and upper and lower limits. The signal wobbles inside the limits, then a gradual drift pulls it down and finally crosses the lower limit, where the single-point alarm fires. The shaded band is the stretch of days the drift was visible in the trend but no single point had crossed a limit, the days an accumulating rule would have alarmed and the control chart did not.

This chapter's experiment measures the trade. It simulates 150 days of a field's fill rate, injects a gradual break that drifts the rate down by a total of about twice the daily noise, spread over 25 days, and runs three rules over it: a fixed threshold, a control chart, and an accumulating CUSUM. The only fair comparison is at a matched false-alarm budget, so the code tunes each rule to the same rate of false alarms and then measures how fast each one catches the real break, across hundreds of trials. Everything is offline and seeded, so `python run.py` reproduces it.



The delay-against-false-alarms frontier from the run. Each curve is one rule swept across its settings; lower and further left is better, meaning fewer false alarms and faster detection. The accumulating rule sits lowest, and its advantage widens toward the left, where the single-point rules' detection delay climbs steeply.

One thing the experiment holds fixed decides whether any of this survives contact with a real deployment, so it goes before the numbers: the budget is per signal. The run tunes one signal to one false alarm a year. Watch every field across every segment with two rules each and you are running hundreds of signals, and a per-signal budget of one false alarm a year becomes a page every few days across the fleet. The alarm you carefully tuned is drowned by the arithmetic of how many you are running. A real deployment needs a budget for the fleet, not for each signal: group related fields, page only on the few that carry the delivery, and let the rest raise a ticket instead of a phone call. Otherwise the most sensitive monitor in the world just trains the team to silence it.

At a loose budget of one false alarm a month, the three rules are close: the accumulating rule catches the break in 8 days against 11 for the control chart. That gap looks small, and it is where the story would end if you monitored the way demos do. But no real on-call rotation wants a false page every month. Tighten the budget to roughly one false alarm a year, the rate a team actually tolerates, and the rules pull apart. The accumulating rule still catches the break in 15 days and catches every break. The control chart now takes 30 days and misses about one break in twenty-five entirely. The frontier chart shows why: as you demand fewer false alarms, the single-point rules' detection delay climbs steeply, while the accumulating rule barely moves. For a slow drift, at the budgets real operations run, the accumulating rule is the difference between catching the break and not.

The experiment has two edges. The advantage is specific to slow, small drifts. As a rule of thumb from process control, once a shift is both large and abrupt, a cliff of two sigma or more in a single step, a plain one-point control chart catches the sudden drop as fast or faster, because one day is already extreme. The practical answer is to run both: an accumulating rule for the slow creep and a one-point rule for the cliff. And in this test the fixed threshold looked competitive, even beating the control chart at the strict budget, but only because the baseline sat still and I placed the line at the right level. On a real signal whose normal level trends, or swings with a weekly cycle, a fixed line is either too jumpy or too slow, and learning the normal from history is the control chart's entire reason to exist. The durable lesson is that choosing the alarm is a statistics question. Monitoring quality lives or dies there, not on how the dashboard looks.

Baselines, and the trap inside them

Every rule above rests on a baseline, so two things about baselines decide whether any of it works. First, never alarm without one, and give it long enough to learn the signal's real rhythm. A field with a weekly cycle needs a baseline that handles the cycle, either spanning several weeks, or comparing each day only to the same weekday, or watching the difference from seven days earlier so the weekly swing cancels out. Skip that and the rule cries wolf every weekend, the team tunes it out, and you have bought alert fatigue, which is worse than no alarm at all because now a real one gets ignored too.

Second, and this is the subtle one, beware the baseline that absorbs the thing you are trying to catch. If your rule always compares today against a trailing window of recent history, then a slow degradation becomes part of the history it is compared to. Each day looks normal against a yesterday that was already a little broken, and the drift walks the baseline down with it, and no alarm ever fires. The break you most need to catch, the gradual one, is the break a rolling baseline hides. The defense is to anchor the baseline to a known-good period rather than a floating recent window, and to re-anchor it deliberately after a verified change, never automatically.

The healing loop

A break should run through the same loop every time. Detect it, from the data signal. Diagnose it, starting with which of the three surfaces failed, because the response is different for each: restart the box for infrastructure, rotate identity or back off for access, repair the parser for semantic drift. Fix it. Then verify the fix with the same monitor that caught the break, so you know the signal came back, not just that you changed some code. And last, leave a regression sentinel behind: a check that this break, this field going empty or this distribution shifting, would fire immediately if it ever returned. A break you fixed without a sentinel can come back silently, and the second time is always more expensive.

There is a fashionable idea that this loop can be automated end to end, a collector that detects its own drift and re-derives the broken selector without a human. Some of that is real and useful, especially the detection and the fallback to a second extraction method. But treat automatic repair with

suspicion, because an auto-fix that makes the signal look healthy again can just as easily be papering over a deeper change with a plausible wrong answer, which is the failure this book keeps warning against, now dressed as a feature: the pipeline reports success while shipping data that is wrong. Automate the detection freely, and put a human at the gate of the fix.

What monitoring actually buys

The promise is easy to oversell. Monitoring does not stop sources from changing, and it does not prevent breaks. It bounds how long you stay ignorant of one, and that bound is what you are buying. A data operation runs on two clocks: the time to detect a problem and the time to recover from it. For many collector breaks recovery is the shorter clock, because once you know, a selector swap or a parser fix is quick, though a hardened block or a full schema redesign can drag it out. Detection is the clock you control least by default, because without a data signal you do not learn about a silent break until it has shipped for days and a customer has already made decisions on it. The cost of an error climbs steeply the longer it goes unseen, from cheap to catch at the moment it enters, to costly to correct once it is in the data, to ruinous once someone downstream has acted on it. The experiment sets the ceiling on how much it buys: a hard cliff that empties a field moves detection to about a day, but the slow drift the chapter spends most of its ink on still takes on the order of two weeks to catch even with the best rule. Two weeks beats the alternative, which is a customer telling you a month later, and moving detection from a month to a fortnight is the largest lever you have on the total cost of being wrong.

PART IV. RUNNING IT AT SCALE

Chapter 11. Cost Engineering

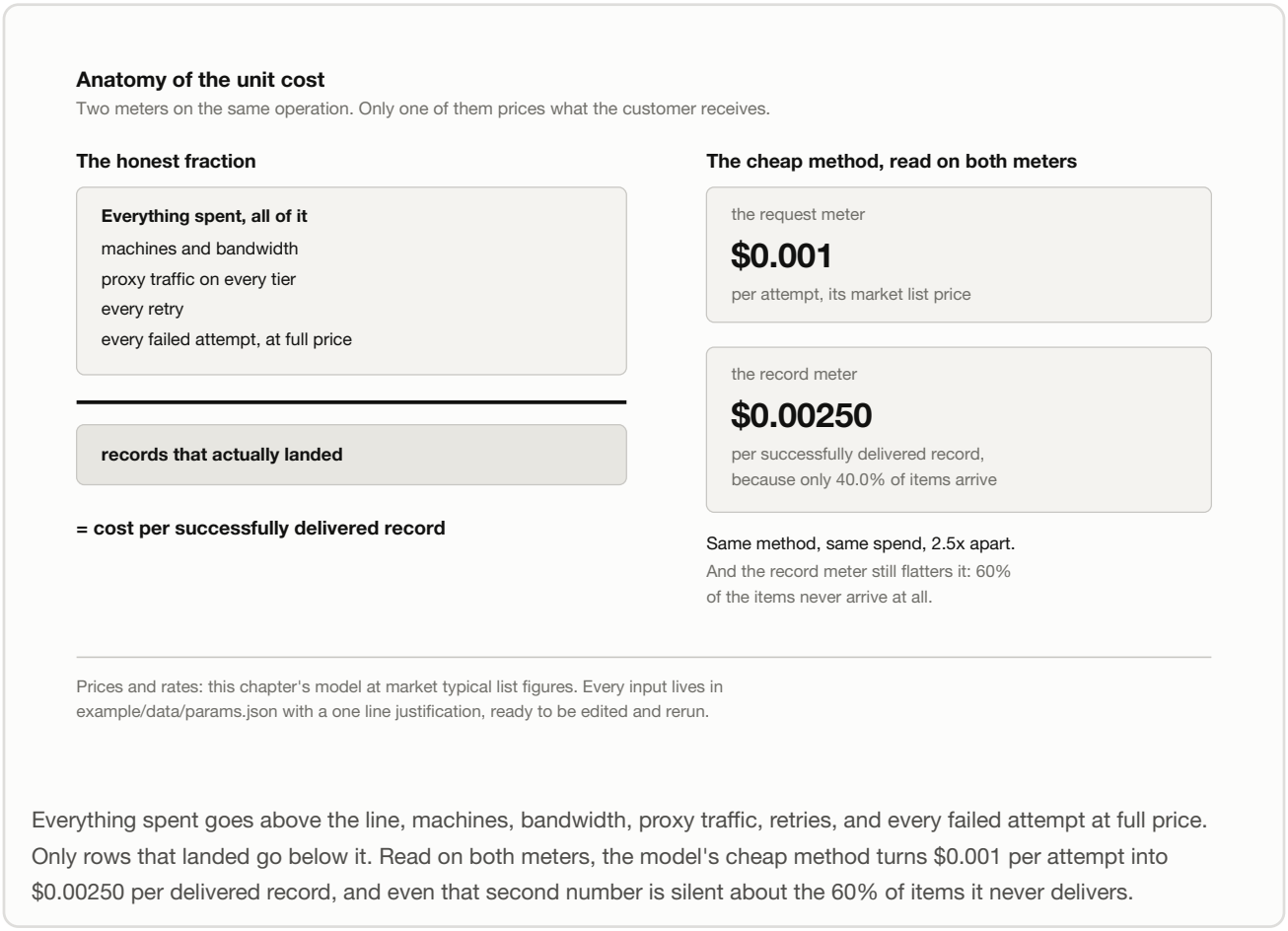
I have sat through many versions of the same budget review. A slide reports that the collection stack runs at a tenth of a cent per request, the room relaxes, and the meeting moves on to something that looks riskier. The number on the slide is usually true. It is also the most flattering number the operation could have chosen to report, because nobody in that room is buying requests. The customer is buying records that arrive, and the economics of this trade live in the distance between those two things.

So this chapter is about one number: cost per successfully delivered record. Take every dollar the operation spends and divide it by the count of rows that actually reached the customer. The definition sounds obvious. Its bite is in what it does with failure. A failed attempt is a cost with no denominator, spend that cannot carry itself, so its price has nowhere to go except into the records that did arrive. Count failure that way and several habits that look normal in this industry start looking off.

The only number that matters

Cost per request divides spend by everything you tried, and trying is the one thing your successes and your failures have in common. Trying is the biggest divisor you can pick, which is why the number comes out so small, and that is how it ends up on slides. Cost per delivered record divides the same spend by what landed, and nothing else about the arithmetic changes.

The gap between the two meters is wide. Take the cheapest method in this chapter's model, a plain HTTP fetch from datacenter addresses at $0.001 per attempt, the model's market list figure. On a defended site the model gives it a 40% success rate, and the delivered-record meter turns a tenth of a cent into $0.00250 per delivered record. It was always this expensive; the request meter could not see it. Even that second meter is still flattering this method, because 60% of the items never arrive at all, and a unit cost says nothing about the rows it walked away from.



Everything spent goes above the line, machines, bandwidth, proxy traffic, retries, and every failed attempt at full price. Only rows that landed go below it. Read on both meters, the model's cheap method turns $0.001 per attempt into $0.00250 per delivered record, and even that second number is silent about the 60% of items it never delivers.

Anatomy of the unit cost

What belongs in the numerator is all of it. The machines that run the collectors. The bandwidth. The proxy traffic on every tier, which Chapter 4 taught you to buy deliberately rather than by reflex. Every retry, because Chapter 6's backoff discipline spends real attempts. Every failed attempt, at full price. And the people, which polite cost models omit and serious ones put first: Chapter 1's build-vs-buy model found engineering at 92 percent of the in-house bill at its breakeven volume, and nothing in this chapter repeals that finding.

What this chapter's model prices is deliberately narrower: the attempt costs, what you pay each time you press the button. People costs move slowly, and Chapter 1 already modeled them. Attempt costs respond to routing decisions you can change this week, which makes them the lever worth isolating. The model uses three methods at market typical list figures. A plain datacenter fetch at $0.001 per attempt, succeeding on 40% of items. The same fetch through residential addresses at $0.010, succeeding on 75%. And a rendering or unblocker style product, the managed services that fetch a hard page for you at a per-page price, at $0.050, succeeding on 98%. Every one of those assumptions lives in a small shipped file (example/data/params.json) with a one line justification beside it. Disagree with any of them, edit the file, rerun the model.

Run the delivered-record meter over the premium method alone and you get the first lesson in unit anatomy. At $0.050 per attempt and a 98% success rate, the cost per delivered record is $0.05102. The missing 2% of attempts did not vanish from the books: they were paid for, delivered nothing, and reappeared inside the price of everything that landed. That mechanism is failure amortization. It sits inside every unit cost, and it grows fast as success rates fall.

The waterfall as an economic instrument

Chapter 3 built the waterfall as an architecture: find every door into a source, order the doors by price, send every item to the cheapest first, and let only the failures fall through to the next one. There the argument was machine time and the numbers were illustrative. Here the waterfall becomes the main instrument of cost engineering, and the numbers come from a model you can rerun and argue with.

Walk a thousand items through the model's three tiers, taking the list rates at face value for a moment and rounding to whole items. The cheap tier tries everything and delivers 400. The 600 leftovers fall to the mid tier, advertised at 75%, so it should deliver 450 and pass 150 on. The premium tier, at its advertised 98%, rescues 147 of those and leaves 3 undelivered. Total spend: $1.00 on the cheap tier, $6.00 on the mid, $7.50 on the premium, $14.50 for 997 delivered records. That is $0.01454 per delivered record, against $0.05102 for sending everything through premium. A 3.51x saving, and the waterfall even delivers more, 99.7% of items against 98.0%, because three chances at an item beat one. One definition, since this is the number the mix produces: the blended cost is the average cost per delivered record across all tiers together, and it is the only unit cost a mixed operation actually has. On those numbers you would sign today.

Do not sign yet.

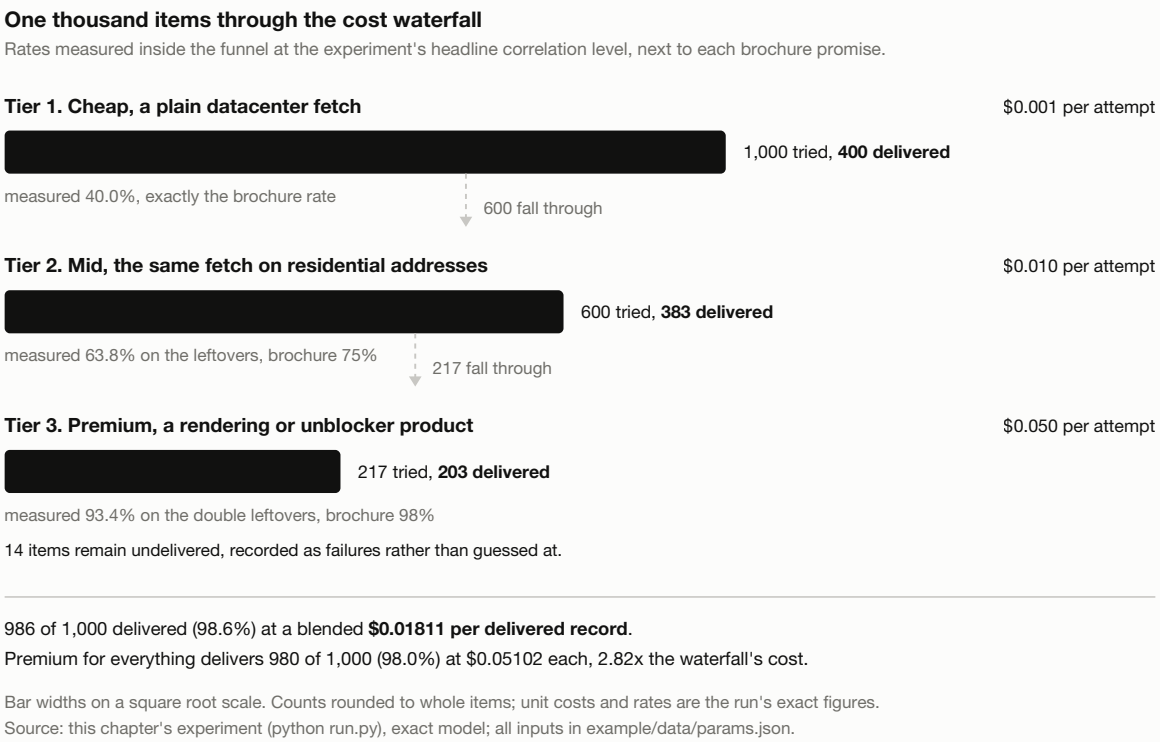
The success rates in that arithmetic are brochure rates, each method's score when the whole population of items is thrown at it, and the arithmetic assumed, without saying so, that they were independent, as if every attempt were a fresh coin flip. In production they are not. An item that failed the cheap method is usually a hard item, a defended endpoint, an odd template, a page behind aggressive protection, and hard items fail the next method more often too. Difficulty travels with the item rather than the attempt. The mid tier never sees the population its brochure was measured on; it sees the cheap tier's rejects. How much that one fact moves the answer is exactly what this chapter's experiment measures.

What correlation does to the promise

I wrote the hypothesis in the brief before the model ran. It said: a tuned waterfall cuts blended cost several-fold against premium-only, and correlated difficulty erodes the naive estimate enough to matter, which is why the mix must be tuned from measured per-tier success rates rather than brochure numbers. Two clauses. The run judged both.

The model gives every item a hidden difficulty drawn from a bell curve and lets each method's chance of success fall as difficulty rises. One dial sets how much of success lives in the item against the individual attempt. At one end, every attempt is an independent coin flip, the brochure world above. At the other, difficulty is a strict ladder: an item too hard for premium is too hard for every method beneath it. And the uncomfortable part: at every level of the dial, each method is re-calibrated so its population-wide success rate still matches its brochure, so correlation leaves every brochure true and changes only what the leftovers look like. The run evaluates all 15 orderings and subsets of the three methods at 11 correlation levels, computes exact answers by numeric integration, and cross-checks its headline cell with a seeded 200,000-item simulation. Everything is offline; `python run.py` reproduces every number in this chapter and the chart below.

Its tables print a moderate correlation level, one that lands the mid method at a measured 63.8% on the cheap method's leftovers, nearer the strict ladder than independence. At that level, the same thousand items come out differently.

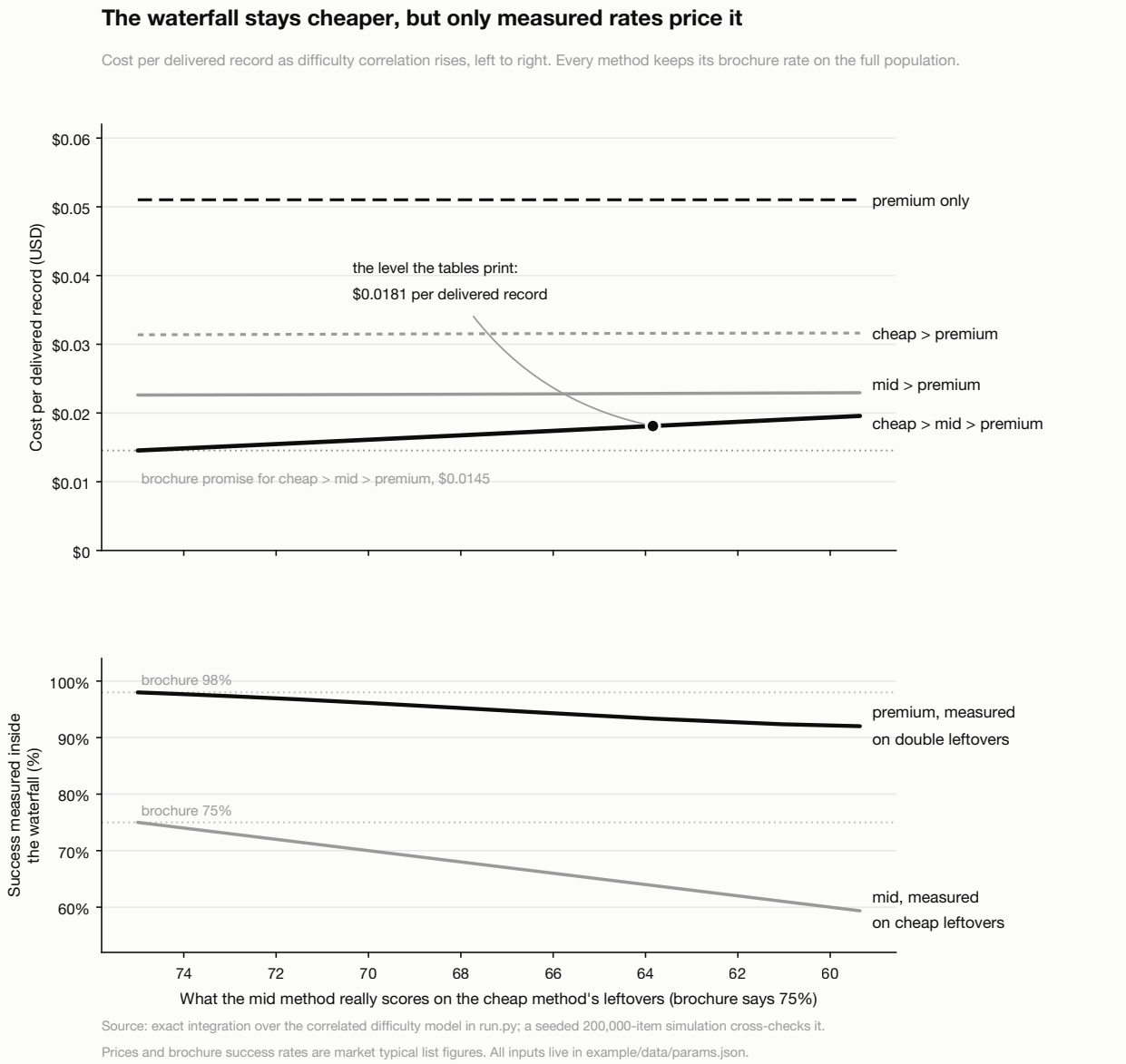


The measured funnel at the experiment's headline correlation level. The cheap tier goes first, sees the whole population, and scores its brochure 40.0%. The mid tier, advertised at 75%, measures 63.8% on the leftovers it actually receives. The premium tier, advertised at 98%, measures 93.4% on the double leftovers. 986 of 1,000 items arrive, and the blended cost is $0.01811 per delivered record against $0.05102 for premium alone.

The cheap tier still delivers its 400; it goes first, so the brochure rate is the measured rate. The mid tier delivers 383 of its 600, a measured 63.8% against the advertised 75%. The premium tier rescues 203 of the final 217, a measured 93.4% against the advertised 98%. Fourteen items end the day as

recorded failures. Delivery lands at 98.6% instead of the promised 99.7%, and the blended cost lands at $0.01811 per delivered record instead of the promised $0.01454.

Set the two funnels side by side and take the two clauses in turn. The saving survives: $0.01811 against premium-only's $0.05102 is 2.82x, and no correlation level in the sweep pushes the ratio below 2.56x, the strict-ladder worst case, where the waterfall costs $0.01990 per delivered record. The erosion is real too, and material: the true unit cost ran 25% above the naive plan, and coverage came in under the promise, 99.7% promised against 98.6% real. The three chances mostly stopped being three chances, because the leftovers are the hard items every tier struggles with. Both clauses held, the first at nearly threefold even in the worst case. And notice what the brochures did through all of this: nothing wrong. Every method held its advertised rate on the full population at every correlation level, and the plan built on those rates still missed. The numbers stayed true and still misled, which is harder to catch than a vendor fudging.



The full sweep. Cost per delivered record as difficulty correlation rises from left to right, with every method holding its brochure rate on the full population throughout. The cost-ordered waterfall drifts from its $0.0145 brochure promise toward the $0.0199 strict-ladder bound and never stops being the cheapest strategy that clears the 95% delivery floor, while every premium-first strategy stays pinned near $0.051. The lower panel is the mechanism: the rates measured inside the funnel slide away from the brochure lines as correlation rises.

The sweep behind the chart adds a finding I did not put in the brief, and it is the one I use most. The erosion has a floor. As correlation rises from coin flips to a strict ladder, the waterfall's cost climbs from $0.01454 to $0.01990 and stops there, because once difficulty is a perfect ladder there is nothing left to correlate. Cheapest-first remains the winning ordering at every level of the sweep, and the waterfall never comes close to losing to premium-only. Correlation re-prices the decision without ever reversing it. So you can choose a waterfall without knowing your correlation level. You need that number to budget one.

If you own that budget, the rule I would take to finance is this: never submit the brochure number. Compute the strict-ladder bound from the same brochure rates and prices, no measurement needed, and budget there; in this model that is $0.01990 against the brochure's $0.01454, about 37% higher. A budget set at the bound covers the worst case this arithmetic allows, while the brochure plan missed by $3,565 a month at a correlation level that was moderate, not extreme. Then let the measured blended cost replace the bound once you are measuring your own funnel, and hand back the difference.

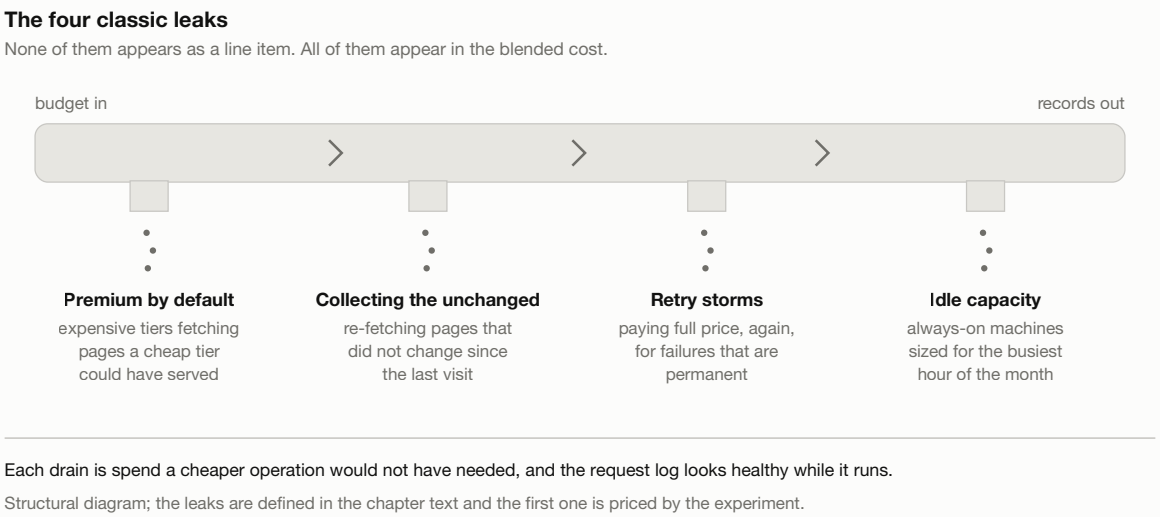
The same table holds three smaller results. Every ordering that runs the premium method first costs between $0.05094 and $0.05098 per delivered record, within a quarter of a cent of premium-only, no matter what runs behind it, because once the expensive method has taken the spend there is nothing cheap left to save; a waterfall's savings live in its ordering more than its membership. A cheap tier in front is never a detour either: cheap then mid beats mid alone on cost, $0.00894 against $0.01333, and on delivery, 78.3% against 75.0%. And the cheapest number in the whole table is a trap. The cheap method alone posts $0.00250 per delivered record and abandons 60% of the items, which is why cost per delivered record only means something with a delivery floor standing next to it, a contractual minimum share of items that must arrive. Against the model's 95% floor, the full waterfall wins at $0.01811 and 98.6% delivered. A unit cost quoted without its coverage is cream-skimming: delivering the easy items and letting the average look complete.

Unit costs this small look harmless, so translate them at the model's volume, 1,000,000 delivered records a month, a mid-size production feed. Premium-only spends $51,020 a month. The tuned waterfall spends $18,109 for the same delivered count. The gap is $394,938 a year. And a team that budgeted its waterfall from brochure rates still misses its own budget by $3,565 a month, because it planned with someone else's numbers. One verification note: a seeded 200,000-item rerun of the same funnel lands 0.20% from the exact answer.

If you sell data, that gap is your competitive position. Two vendors delivering the same million records a month can carry costs of $51,020 and $18,109, and the one with the tuned waterfall can sell below the other's raw cost and still make money, which makes routing order part of the business model rather than an engineering detail. I treat a month of routing work as a product feature now, and this arithmetic is why.

Where the tenth of a cent goes

Most collection budgets erode rather than die of one bad decision, through four drains I have seen often enough to keep as a checklist.



Four drains on the same pipe. Two of them waste spend on requests that succeed, one wastes it on requests that fail, and the last wastes it on machines doing nothing. The request log looks healthy during all four, which is what lets them run for months.

Premium by default is the expensive one, and the experiment has already priced it. Run the premium method first and your cost pins to within a quarter of a cent of premium-only, whatever else you built. In operations this leak rarely looks like a decision. A team adopts a rendering browser or an unblocker product for the hard cases, discovers it always works, and routes everything through it, because reliability feels like a virtue. Chapter 3 called this the browser's gravitational pull. The unit math above is what surrendering to it costs.

Collecting the unchanged is subtler. A daily collector happily re-fetches pages whose content has not moved since yesterday, paying full collection price to learn nothing. The remedy is change detection, any cheap signal that a page changed since your last visit, such as a listing's own updated timestamp or a sitemap date the source already publishes, consulted before the full fetch is paid for. It pairs with the raw-response cache Chapter 7 made the default: never re-buy a page your own storage already holds. In slow-moving catalogs, and most catalogs move slower than their collectors do, this is the largest single discount available. I say that as operator judgment, because how much of your catalog changes per day is a number only your own data can give you.

Retry storms are Chapter 6's lesson arriving as a bill. That chapter split failures into ambiguous ones, worth retrying with growing pauses, and definitive ones, where the source has already answered. A retry against a permanent failure buys the same answer again at full price, as many times as the policy allows, and a fleet doing this on a bad night multiplies its own cost while delivering nothing new. The fix is to classify the failure before any retry fires, and it costs nothing to build.

Idle capacity is the least glamorous drain. Collection is bursty by nature, nightly jobs and weekly sweeps with quiet stretches between, and a fleet of always-on machines sized for the busiest hour

keeps billing through the empty ones as well. Compute that scales to zero between jobs fits this workload unusually well. This one is operator judgment rather than a measured result, but of the four leaks it is the one your cloud invoice will confirm fastest.

Cost observability

None of the measured rates in this chapter exist anywhere except inside an operation that records them. The 63.8% and the 93.4% belong to the mid and premium methods meeting your leftovers, and no vendor can print your leftovers in a brochure. So the instrument that makes cost engineering possible is embarrassingly small: tag every attempt with its method, its unit price, and its outcome, and keep Chapter 3's habit of writing which door served each record into the record itself. From those tags, the per-tier rates inside your funnel and the true blended cost fall out as queries instead of modeling exercises. That is the rule underneath this chapter: measure the blended number rather than modeling it, because the modeled one ran 25% under the measured one in a world where every input stayed true.

Pricing from the measured cost

Sooner or later the unit cost has to face a customer, and the order of operations matters. Price from the measured cost instead of the modeled one; a contract priced against the naive $0.01454 gives away $3,565 a month of imagined margin before anything has even gone wrong. Put the delivery floor into the contract next to the price, because a unit price without a coverage promise invites the cream-skimming trap from the run, and a buyer who has read this chapter will ask for both numbers.

Build the margin against the bound rather than the sunny day. Sources change, and Chapter 10 was about how they change without telling you. When a source hardens, its items get harder, which in this model's terms is correlation rising, and your blended cost drifts up the way the chart's waterfall line drifts from $0.01454 toward $0.01990. A price with margin at the sunny-day cost and none at the bound is a price that a single source redesign turns into a loss. The bound is knowable in advance. Price above it, and a hard month re-prices your funnel without re-opening your contract. The tags from the last section then finish the job: when the cheap tier's share starts sliding, you see the migration and its cost the week it begins, long before the invoice does.

Sit in the buyer's chair for a moment. Chapter 1 made you collect three quotes on one specification, and its model prices the spread you should expect: half of list to double it. This model shows where such spreads come from: two vendors selling identical records can sit 2.82x apart on cost purely from routing order. So the spread comes from routing, a low quote is not automatically corner-cutting, and a seller should put both numbers in writing before a buyer has to ask. A low price with no floor is the $0.00250 trap from the table, the one that delivered 40% of the items.

Every division in this chapter had the same denominator, records successfully delivered, and I have been treating that word, delivered, as solid ground, as if an operation always knows which rows

reached the customer. It is time to admit that it often does not. Pipelines mark records delivered because an upload call returned. Status fields drift away from what the customer's storage actually holds. A bill computed on this chapter's arithmetic is only as good as the count underneath it. That count is won or lost at the destination, in the customer's own storage.

Chapter 12. Delivery You Can Trust

The customer writes that Monday's files are not in the bucket. Your own status table says delivered. Now there are two accounts of one bucket, and one of them is wrong, and it is probably yours, because the customer is reading the actual storage while you are reading a note your own code wrote to itself. That three-sentence ticket is the most expensive one in this business. I have answered a few. They start as missing files and become an argument about whether anything your system says can be believed.

Chapter 11 made cost per delivered record the number that runs a collection business. Every part of that calculation leans on one word staying true. Delivered, according to what? In most pipelines the real answer is a status field the pipeline asserted about itself, and this chapter measures how often that assertion is false. The last mile has its own disciplines. Verify at the destination instead of trusting your own flags. Make re-delivery safe to repeat. Ship proof the customer can check without you. Bill only what landed. Four small habits, and the rest of the chapter puts numbers on what ignoring them costs.

Where the drift is born

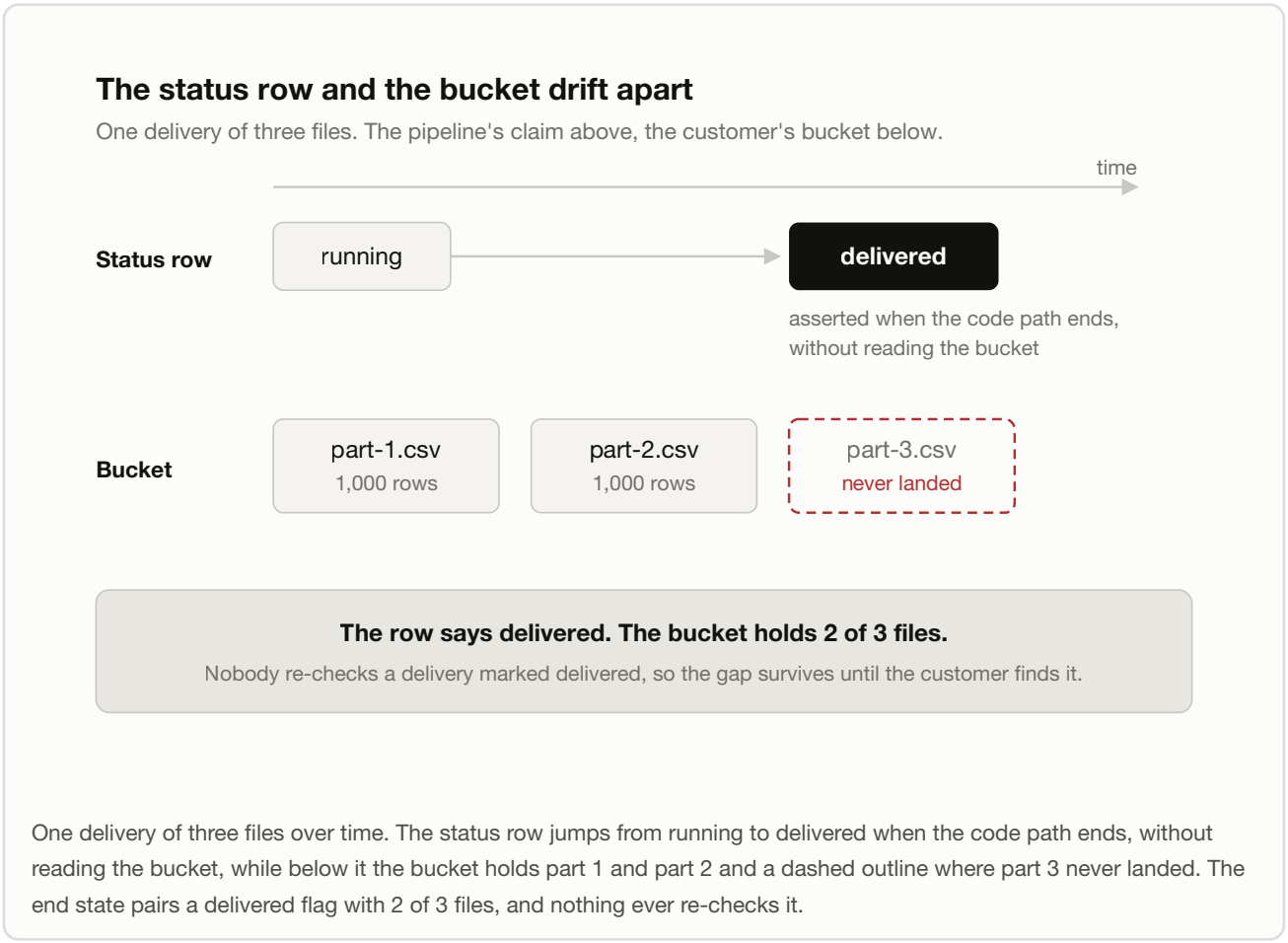
A delivery pipeline keeps a small record per delivery, the status row, saying running, delivered, or failed. The customer's storage holds the actual files, usually in an object store, a service such as S3 or Google Cloud Storage that keeps files in named containers called buckets. The row and the bucket are written by different actors, at different moments, across a network that fails. That gap is where they drift apart. Given enough deliveries, the two will disagree.

Three mechanics do most of the damage. A process can die in the boundary between the last upload and the status write, and the delivery it just completed stays at running forever. An upload can time out ambiguously: the client gave up waiting, but the bytes may have landed anyway, and nothing visible to the client separates the two cases. And a loop can reach its final line with a partial result and mark delivered anyway, because the code finished even though the data did not.

You do not need a bad engineer for any of this. A network, a deadline, and a pipeline that treats its own memory of events as reality will produce the same drift.

Truth lives at the destination

The rule of the last mile is short enough to put on a wall: truth lives at the destination, in the bucket the customer reads, and in neither your database nor the green tile on your dashboard. A status field is a claim. The bucket is a fact.



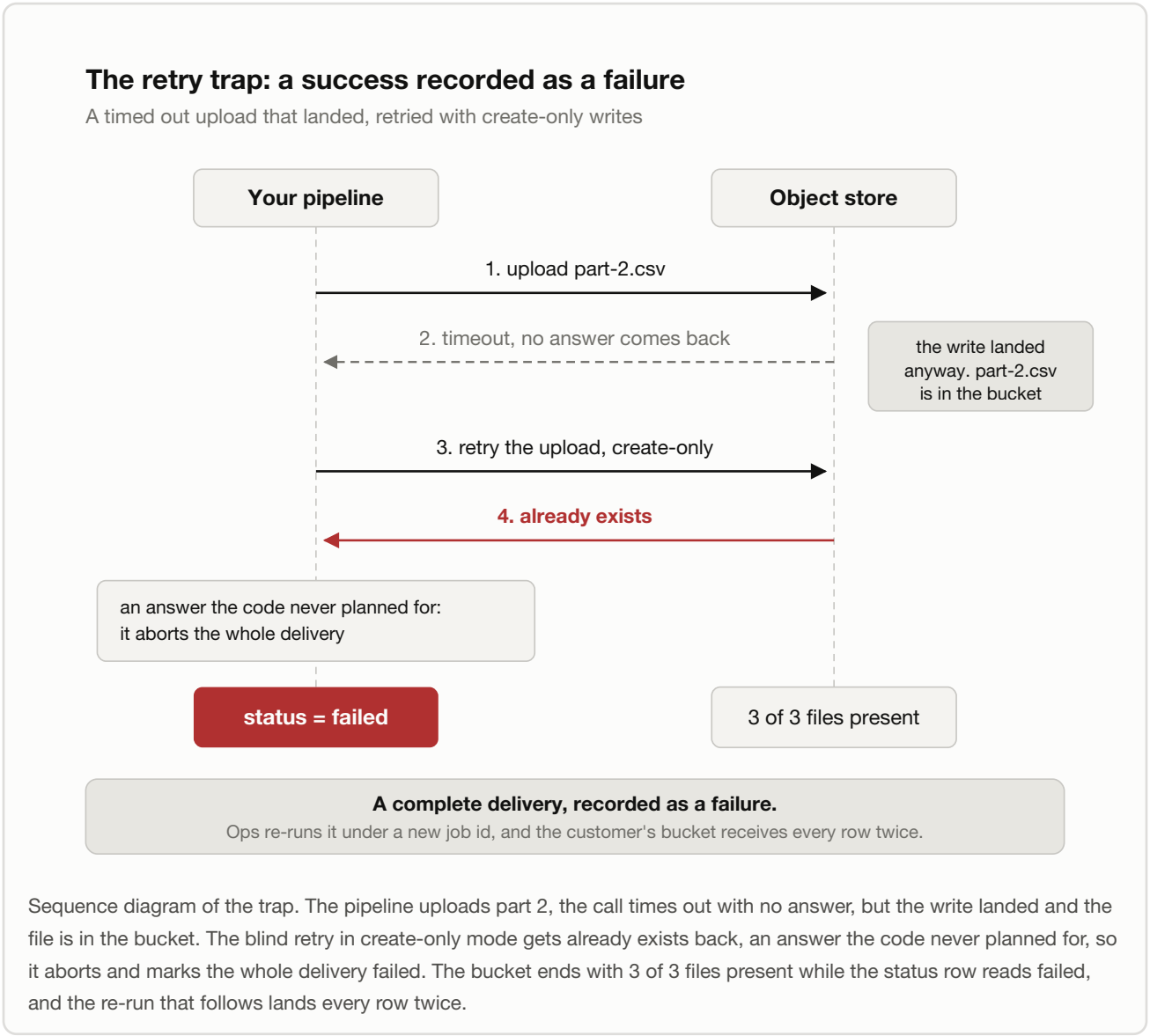
One delivery of three files over time. The status row jumps from running to delivered when the code path ends, without reading the bucket, while below it the bucket holds part 1 and part 2 and a dashed outline where part 3 never landed. The end state pairs a delivered flag with 2 of 3 files, and nothing ever re-checks it.

Living by the rule means closing a loop most pipelines leave open. After the last upload, list the destination: ask the store which objects exist under this delivery's prefix, the folder-like path its files share, and at what sizes. Compare the listing against what you meant to send, file by file, and write the status only then, as a reading of that comparison. The row becomes a cache of something observed, and anything cached from storage can be re-derived from storage the moment a doubt appears. Everything else in the last mile leans on that. When the three sentence ticket arrives, you do not scroll logs reconstructing the night from memory; you run the same listing the customer can run, and the conversation ends in minutes, whichever way it goes.

The retry that records a success as a failure

Verification tells you a delivery is incomplete. Whether you can fix it safely is a separate discipline, and its name is idempotency: an operation is idempotent when running it twice leaves the world exactly as running it once did. In the last mile that comes down to naming. Give every file a deterministic name, computed from the delivery and the part, orders_2026-08-19_part-2.csv, never anything containing a random job id. A re-run then writes the same names it wrote before, replaces its own partial leftovers, and cannot mint a second copy of anything.

Plenty of pipelines do the opposite twice over. They name files after the run, so each retry creates new objects beside the old ones. And some opt in to create-only mode, where the store refuses to overwrite an existing object. Create-only sounds prudent, and it hides a trap with teeth.



Sequence diagram of the trap. The pipeline uploads part 2, the call times out with no answer, but the write landed and the file is in the bucket. The blind retry in create-only mode gets already exists back, an answer the code never planned for, so it aborts and marks the whole delivery failed. The bucket ends with 3 of 3 files present while the status row reads failed, and the re-run that follows lands every row twice.

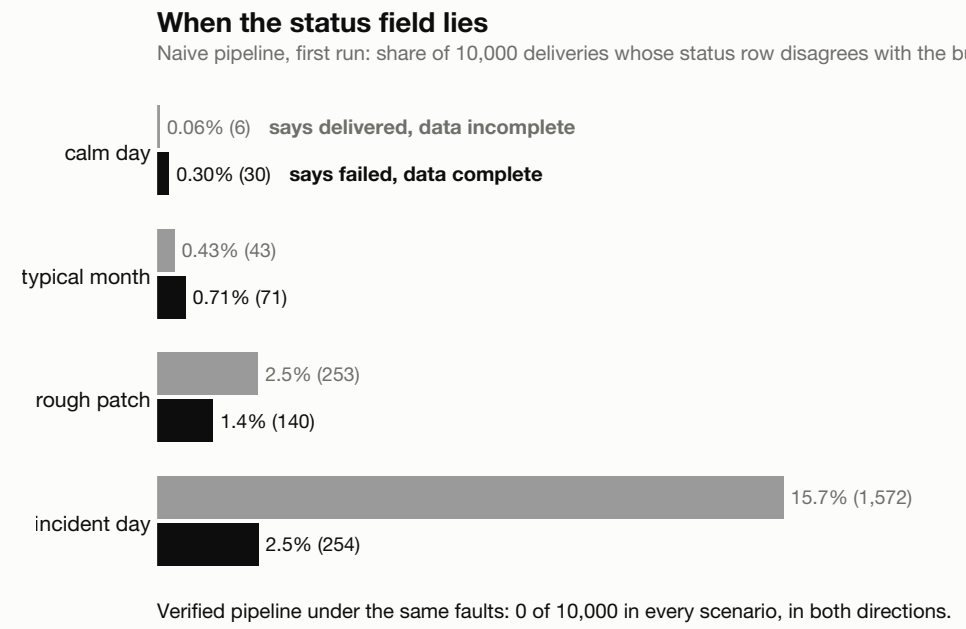
Walk the sequence. An upload times out. The pipeline cannot know the bytes landed, so it does the reasonable thing and retries. The store answers already exists, an error the retry logic never anticipated, and code meeting an unanticipated error does what unhandled paths always do: it aborts and marks the whole delivery failed. Every row is in the customer's bucket, and your table says failed. Ops, doing its job, will now re-deliver everything under a fresh job id, landing every row twice. One ambiguous timeout has become duplicate data and a wrong invoice, and the bug is nowhere you would think to look.

Ten thousand deliveries, four kinds of day

I wanted to know how often this actually happens, so I built the sloppy pipeline on purpose and measured it. The chapter's example simulates deliveries to a mock object store, 10,000 per scenario, each delivery 3 files of 1,000 rows, so one scenario moves 30,000,000 rows. Every upload attempt can fail cleanly, time out ambiguously with the write landing anyway at probability 0.5, or die in a crash at any step boundary, including the one between the last upload and the status write. Four fault environments span a factor of twenty on clean upload failures, from a calm day at 1% to an incident day at 20%. Those rates are my modeling choices, not measurements of any provider. Each one lives in the example's `data/params.json` with a one line justification, and you can change them and re-run; everything is offline and seeded, so `python run.py` reproduces every number in this chapter.

Two pipelines face identical deliveries under identical pre-drawn fault schedules, with the same budget of 2 upload attempts per file. The naive one behaves exactly as the last two sections warned: create-only writes, a blind retry, anticipated errors skipped, delivered asserted at the end of the loop, re-delivery under a fresh job id. The verified one writes idempotent names, settles its status from a bucket listing, and reconciles crash victims from storage. Afterward an ops pass re-runs everything marked failed, because that is what a team that trusts its status field does.

Scoreboard after the first pass, in the typical month scenario: the naive status row disagreed with the bucket on 142 of 10,000 deliveries (1.42%). The breakdown: 43 rows said delivered while data was missing. 71 said failed while every row was present. 117 sat at running forever, 28 of them over complete data, and only those 28 join the lie count, because a running row over an unfinished bucket is at least telling the truth. All of that on an unremarkable month.



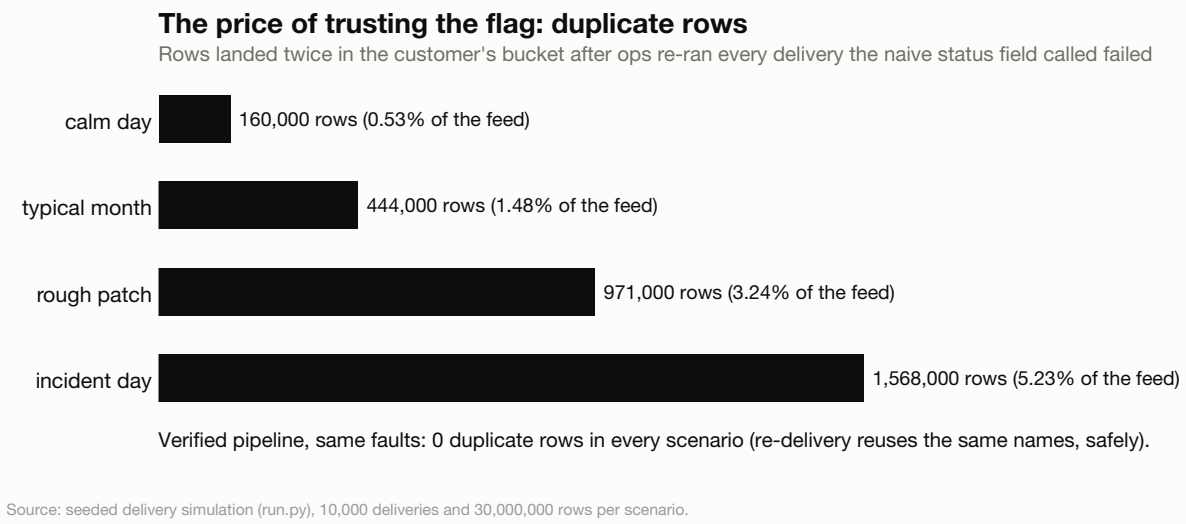
Source: seeded delivery simulation (run.py), 10,000 deliveries per scenario, 3 files of 1,000 rows each, 2 upload attempts per file.

Share of 10,000 deliveries whose naive status row disagreed with the bucket after the first run, split by direction. Says delivered with data incomplete grows from 0.06% (6) on the calm day to 15.7% (1,572) on the incident day, while says failed with data complete grows from 0.30% (30) to 2.5% (254). The verified pipeline recorded 0 in every scenario, in both directions.

The fault dial moves the rate and never removes it. On the calm day the flag still misled on 45 of 10,000 deliveries (0.45%), one misleading delivery per 222 on a healthy network. On the incident day it misled on 2,021 (20.21%). And the verified pipeline, under the same pre-drawn faults: 0 of 10,000. In every scenario, in both directions, both before and after re-delivery. It did not lie at all, because a status computed from the bucket has nothing to drift from.

The bill for trusting the flag

Disagreement counts feel abstract until the ops pass runs. In the typical month the team re-ran the 228 deliveries the naive flag called failed, and 71 of those re-runs were unnecessary, deliveries whose every row was already sitting in the bucket. Because re-delivery wrote under fresh job ids, the blind re-runs landed 444,000 duplicate rows in the customer's bucket, 1.48% of the whole feed. Under per-row billing that is double billing, and under per-delivery billing it is corruption, rows the customer's loader ingests twice.



Duplicate rows landed in the customer's bucket after ops re-ran everything the naive flag called failed: 160,000 rows (0.53% of the feed) on the calm day, 444,000 (1.48%) in the typical month, 971,000 (3.24%) in the rough patch, 1,568,000 (5.23%) on the incident day. The verified pipeline, re-delivering under the same names, landed 0 duplicates in every scenario.

The invoice came out wrong in both directions at once: billed from the status table, it charged for 44,000 rows that never landed and skipped 189,000 rows that landed without ever being marked delivered. Which direction hurts more is not close. The 189,000 unbilled rows are revenue you gave away, and you will never even know the size of the gift unless you reconcile. The 44,000 phantom rows are the ones a customer's auditor finds, and after that audit every invoice you send gets re-checked line by line.

Re-delivery healed selectively, which is the finding that surprised me most in the whole run. The pass cut says failed with data complete from 71 down to 3, at the price of the duplicates, and even that residue has churn inside it: tracked delivery by delivery, 67 healed, two re-runs crashed into stuck-running, two stayed put, and the pass minted one brand-new hidden success by walking into the same create-only trap. It cured 0 of the 43 phantom deliveries. Says delivered with data incomplete stayed 43 of 43, because no process anywhere re-checks a delivery marked delivered. The phantom is permanent until the customer finds it. The stuck-running rows did not heal either; nobody re-runs those, and only a reconciler that reads the bucket, the verified pipeline's habit, ever settles them.

The permanence has a dashboard consequence. A tile that counts status rows can never show a phantom, because a phantom is marked delivered, and alerts keyed to failed stay quiet for the same reason. Make the green tile count bucket listings that matched rather than rows that say delivered, and treat any delivered number that nothing re-checks as testimony. A monthly storage query, the same one the billing section will ask for, settles the delivered tile too; put it on a schedule, and phantoms stop being permanent.

One result decided the design for me: it delivered more data, on top of the cleaner bookkeeping. End state in the typical month, 9,995 complete deliveries of 10,000 against the naive pipeline's 9,863. On the incident day, 9,612 against 7,464, under identical faults. No extra retries were granted anywhere. The verified pipeline spent its ops pass on the right deliveries, because a failed list read from the bucket is a correct repair list, while the naive team re-ran 71 healthy deliveries as its 43 phantoms sat marked delivered.

Those end states are what I read before signing a service level agreement, the completeness promise written into a data contract. Under identical faults and the same retry budget, the naive pipeline truly completed 98.63% of deliveries in the typical month, so a 99% promise breaks on an unremarkable month, and its delivered tile still includes 43 deliveries with data missing, so the customer reports the breach before you do. The verified pipeline finished the same month at 99.95% and held 96.1% on the incident day, against 74.6%. My rule: promise only a completeness number a bucket listing can prove, and never sign 99% with a last mile that asserts its own status.

Judging my own hypothesis

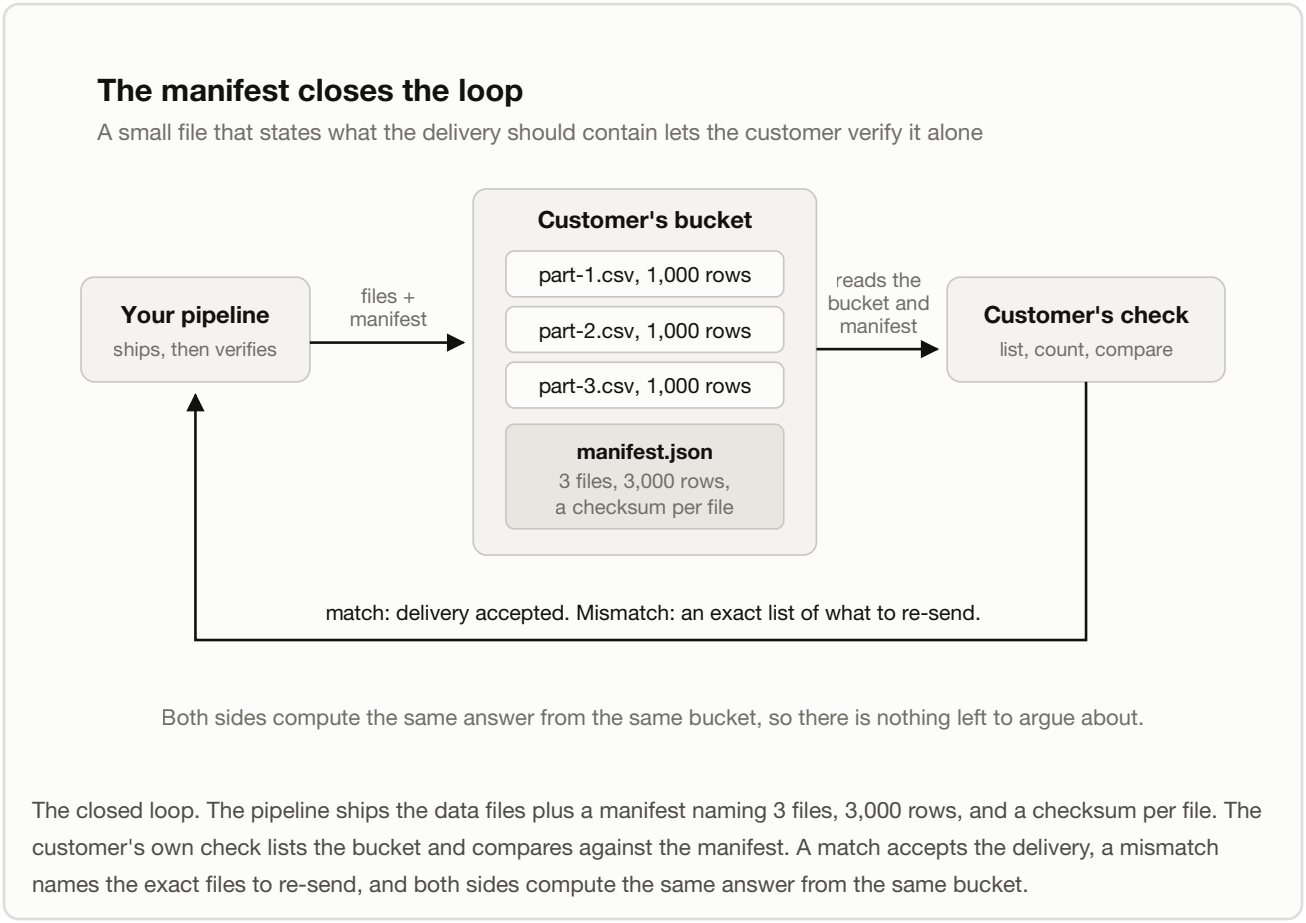
The chapter plan I published stated the hypothesis before anything ran: the naive design lies in both directions at material rates; the nastier direction is says failed but actually delivered, because it triggers duplicate re-deliveries and double billing; and storage-level verification drives both directions to zero.

Two of those three claims survived contact with the data cleanly. The rates are material everywhere, 0.45% of deliveries misled on the calm day rising to 20.21% on the incident day, both directions present in every scenario. The zero is exact, 0 of 10,000 in every scenario, both directions, both measurement points. The middle claim needed a correction I did not see coming. At calm and typical fault rates the hidden success direction is the bigger problem, 71 deliveries against 43, and the 211,000 duplicate rows its unnecessary re-runs landed against 44,000 overbilled. The pass's other 233,000 duplicates came from re-running genuinely incomplete deliveries, a cost of the fresh-name design rather than of either lie. Push the fault dial to incident levels and the balance flips harder than the raw counts suggest: 1,572 phantoms against 254 hidden successes, and 1,740,000 rows overbilled against the 667,000 duplicates the hidden successes caused. Which lie is nastier turns out to track how bad your day is. Permanence does not flip. Re-delivery cut the hidden successes from 71 to 3 and repaired 0 of 43 phantoms, so on any kind of day, the lie that says delivered is the one that never heals. I predicted one villain and the data handed me two, taking turns.

A manifest the customer can check alone

Everything so far still runs on your side of the fence, and a customer who has been burned once is entitled to ask why your verification deserves more faith than your old status field did. The durable answer is to hand over the means of verification. Alongside every delivery, ship a manifest: a small machine-readable file stating what the delivery should contain, the file names, a row count per file,

and a checksum per file, a checksum being a short fingerprint computed from a file's bytes that changes if a single byte changes. Write the manifest last, after every data file, so its presence marks a finished delivery and a loader that waits for it can never ingest a half-written one.



The closed loop. The pipeline ships the data files plus a manifest naming 3 files, 3,000 rows, and a checksum per file. The customer's own check lists the bucket and compares against the manifest. A match accepts the delivery, a mismatch names the exact files to re-send, and both sides compute the same answer from the same bucket.

The customer's side then shrinks to a small script: list the bucket, count rows, compare checksums. A match accepts the delivery with nobody trusting anybody. A mismatch names the exact files to re-send, which turns re-delivery from an argument into a work order. Months later this is also what keeps disputes short. Both sides compute the same answer from the same bucket, and what remains to argue about is arithmetic.

Billing from the bucket

An invoice is a status field with money attached, so treat it with the same suspicion and bill from what landed, never from what was attempted. Concretely, the billing job reads the verified statuses, or better, the bucket listings themselves, and never touches the raw status table. In the run, the verified pipeline's ops pass made 134 re-runs, every one needed, landed 0 duplicate rows, and produced an invoice that matched storage to the row: 29,985,000 rows billed, and the same 29,985,000 verified present for the billed deliveries. The 5 deliveries that could not complete were billed at nothing and reported as failures, their 10,000 rows of partial data named rather than hidden. Reporting your own

failures with that precision costs some pride. A customer who believes those failure reports will also believe your invoices.

Reconcile monthly anyway, delivered rows against billed units, computed from storage. It is one query, and this chapter's run just priced what it catches.

The delivery contract

None of this discipline survives being informal, so agree it in writing with each customer before the first delivery. One page. Mine carries six lines.

1. **Destination.** The exact bucket and prefix, who owns them, and which credentials may write there.
2. **Naming.** The deterministic pattern for every file, date and part number in the name, nothing random.
3. **Format.** File format, compression, encoding, and how corrections are represented.
4. **Manifest.** Its name, its fields, and the promise that it is written last.
5. **Verification.** The check each side runs, and the agreement that the bucket plus the manifest settles any disagreement, not either party's dashboard.
6. **Re-delivery.** Who may request it, how, and the guarantee that it rewrites the same names and can never duplicate.

The page reads like bureaucracy until the first bad night. Then it is why that night stayed boring.

If you are on the buying side, this page is your checklist read in reverse, and there are two questions I would put to any vendor before signing. First, how is delivered computed: from a listing of my bucket, or from your pipeline's own flags? In this chapter's run the flag was wrong on 142 of 10,000 deliveries in the typical month. Second, does a re-delivery rewrite the same file names? The vendor who answers no is the one whose routine re-runs landed 444,000 duplicate rows in the customer's bucket. Make lines 4 through 6 the vendor's written obligations. The full buyer's audit arrives in Chapter 14; these two questions open the door.

It also assumes something larger than any one delivery: that repeating work is always safe. Running a fleet of collectors spends that assumption everywhere, because every rescue of a job that died with work half done is a re-run someone must be able to fire at three in the morning without fear. You just made repetition harmless. That is what lets the fleet stay calm.

Chapter 13. Operating at Scale

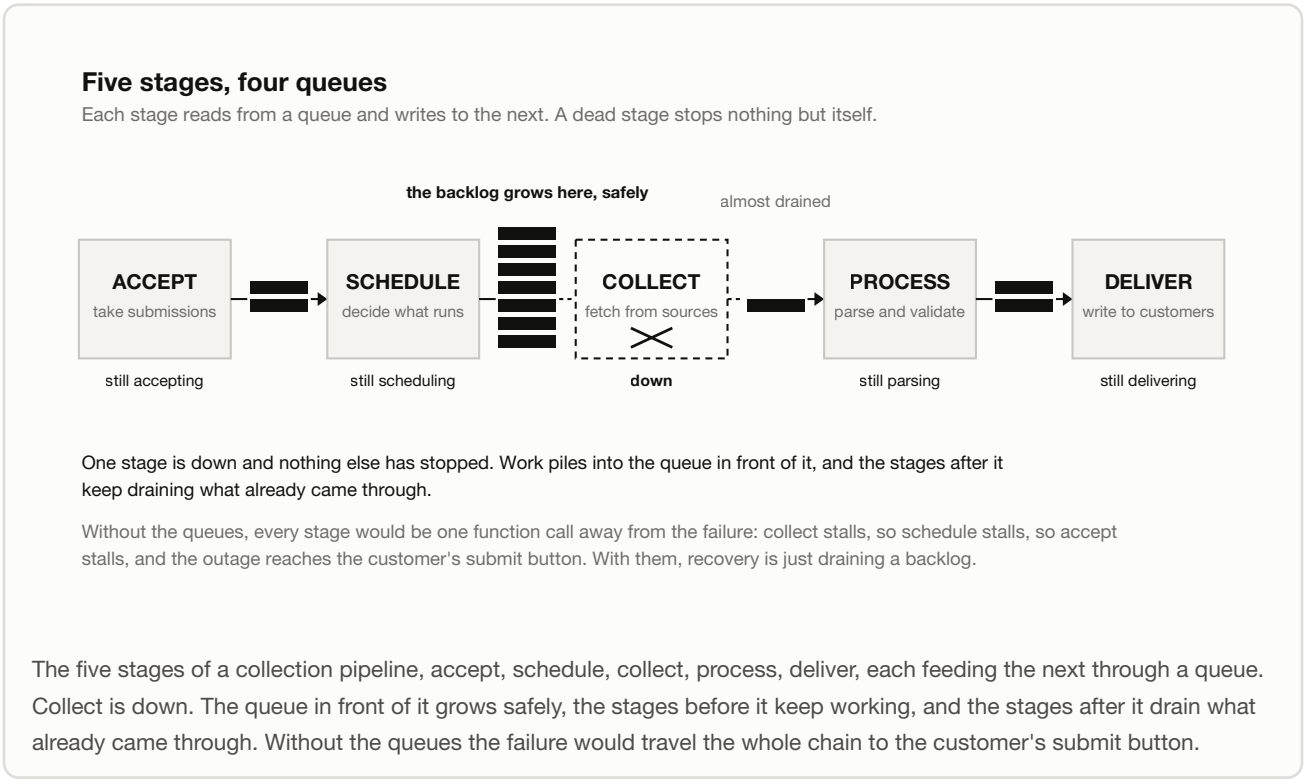
A fleet I ran once locked up with every worker alive and answering its health check. The intake was accepting jobs. And no customer got data, because one large retail site had slowed to a crawl, every worker in the pool was holding one of its half-finished jobs, and every other customer's work sat in line behind them. A whole fleet, one slow source, zero deliveries. I spent an hour restarting things in the wrong order before I understood there was nothing to fix. The system was doing exactly what I had built. I had built one line.

Growing from one collector to a fleet changes the collectors less than you would expect. What changes is the wiring between them, and the wiring is where fleets die. So this chapter is about wiring: queues that keep one slow source from stalling everyone, concurrency treated as a promise instead of an accident, rescue for work that falls asleep, a scheduler that starves no one, and the discipline for the nights when all of it fails anyway. In the middle sits a sizing question with a trap in it, and the experiment walks into the trap on purpose, with the meter running.

Queues between stages

A queue is a humble thing: a list of work waiting its turn, kept somewhere that survives a crash. That humility is the point. The night I described had no queues worth the name, just workers calling sources and each other directly, so the slowness of one site traveled through the whole system like tension through a rope.

A fleet that survives is built as stages with queues between them. Five stages cover almost every collection operation I have seen. Accept takes the customer's submission and records it. Schedule decides what runs next and under which limits. Collect fetches from the sources. Process parses and validates what came back, the work of Chapters 7 and 8. Deliver writes the result where the customer receives it and proves it landed, the work of Chapter 12. Each stage reads jobs from a queue, does its one thing, and writes to the next queue. No stage calls another. The queue is the only thing they share.



The five stages of a collection pipeline, accept, schedule, collect, process, deliver, each feeding the next through a queue. Collect is down. The queue in front of it grows safely, the stages before it keep working, and the stages after it drain what already came through. Without the queues the failure would travel the whole chain to the customer's submit button.

Now replay my bad night in this design. The slow site drags the collect stage, and only the collect stage. Submissions still land. Scheduling still happens. Parsed results still flow to customers whose data was already fetched. The backlog piles up in one queue, in one visible place, where you can measure it, alarm on it, and drain it when the site recovers. An outage becomes a number going up instead of a company going dark.

One word makes the design hold: backpressure. When a queue grows past a set depth, the stage feeding it slows down or starts answering not yet. The pile-up travels backward, stage by stage, until it reaches the edge of the system, where the customer is told queued instead of getting a fast acceptance that means nothing. A system without backpressure accepts work it cannot do and hides the truth in the middle of the pipeline. With it, the truth surfaces at the intake, which is the only place anyone can act on it.

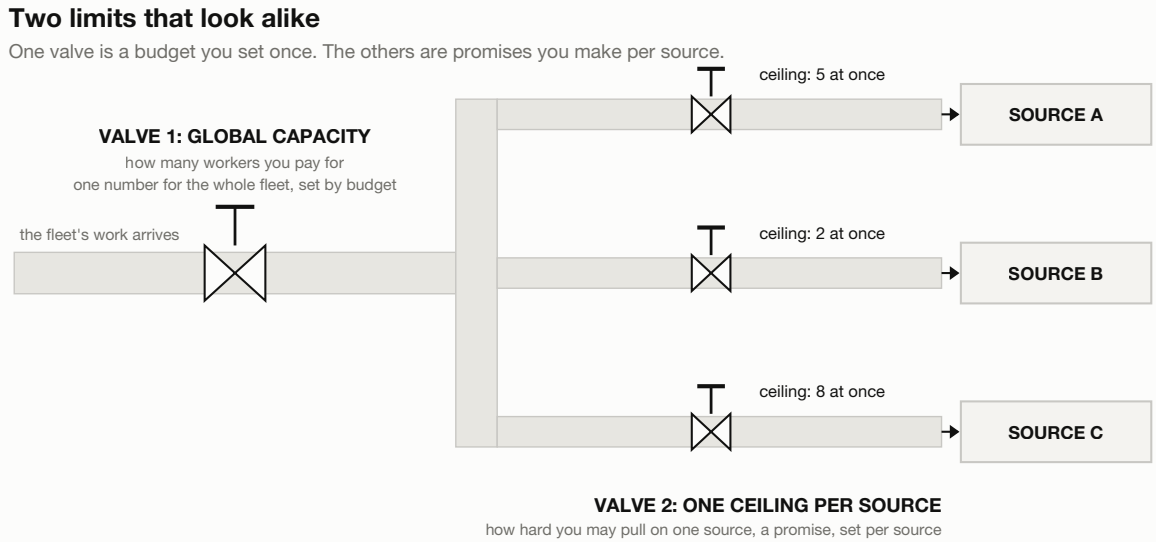
Concurrency is a promise

Two limits govern how hard a fleet pulls, and confusing them is the commonest operations mistake I know. Concurrency, how many jobs run at the same moment, feels like one knob, and it is actually two.

The first is the per-source ceiling: how many requests you allow against one source at once. Chapter 2 argued that the load you put on one source belongs to the source, sized to what it can absorb without noticing you; a simultaneous-request ceiling is that rule translated into fleet vocabulary, and Chapter 6 taught the reflexes, backing off and breaking the circuit, for when you got it wrong. At fleet scale the

ceiling stops being a politeness and becomes an entry in a table: this source, at most this many at once, enforced by the scheduler no matter how much work is waiting or how many workers are free.

The second is global capacity: how many workers you pay to exist. That number belongs to your budget and to the promise you made customers about wait times. It has nothing to do with what any source tolerates.



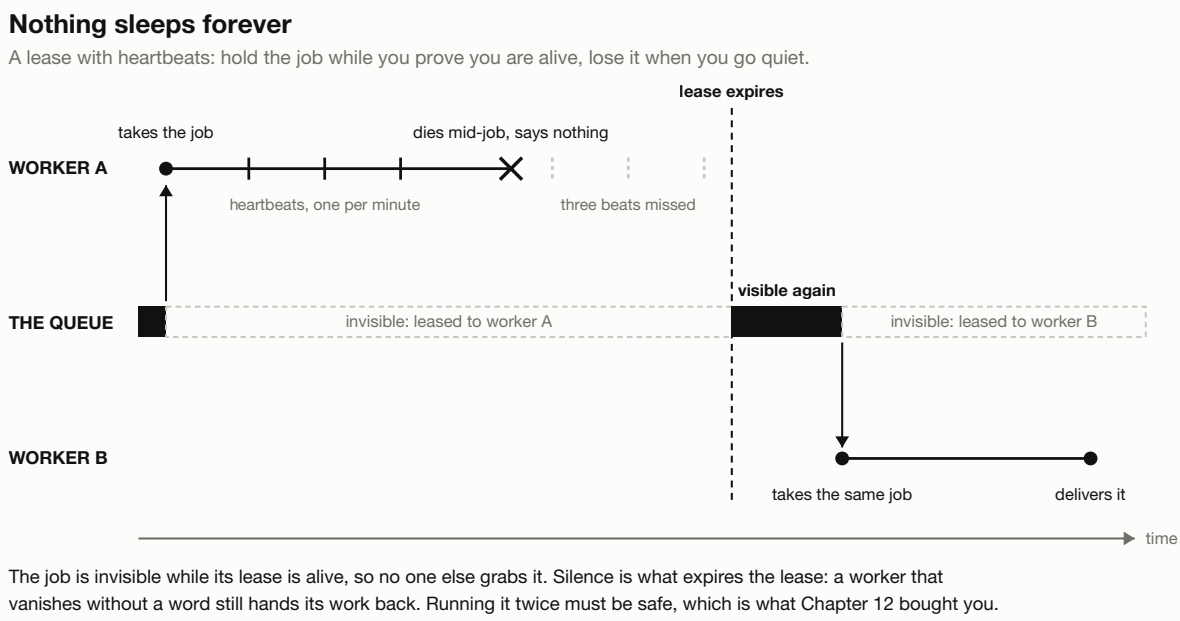
Two valves on the same flow of work. The first is global capacity, how many workers you pay for, one number for the whole fleet, set by budget. After it the pipe splits per source, and each branch carries its own smaller valve, the ceiling that source was promised, 5 at once for one source, 2 for another, 8 for a third. Adding workers opens only the first valve.

Keep the two valves separate and boring things stay boring. You can double the fleet for a growing customer and no source feels a thing, because the ceilings did not move. You can tighten one source's ceiling after a rough week without touching capacity anyone else uses. But if your only limit is worker count, then every scaling decision silently changes how hard you hit every source, and the day you add twenty workers for one customer is the day a different customer's source starts blocking you. On my bad night the coupling ran the other way, one source's slowness absorbing the fleet, and the missing piece was that same per-source table.

Nothing sleeps forever

Workers die mid-job. A machine gets reclaimed or a process runs out of memory; sometimes a network partition swallows a whole box. None of that is exceptional at fleet scale; with dozens of workers it is a normal week. The design question is what happens to the job the dead worker was holding, because the naive answer is: nothing, forever. The records say running, no process is actually running it, and the customer waits. Operators call these zombie jobs, and a fleet without a plan for them leaks work continuously.

The plan has three parts. First, taking a job is a lease rather than an ownership transfer: the job disappears from the queue for a limited time, sometimes called a visibility timeout, rather than being deleted. Second, the worker holding a lease sends heartbeats, small I am alive signals on a fixed rhythm, and each heartbeat extends the lease. Third, silence expires it. Miss a few beats and the job simply reappears in the queue, where the next free worker picks it up as if nothing happened.



Timeline of a rescued job. Worker A takes the job and heartbeats once a minute, then dies mid-job and goes silent. After three missed beats the lease expires and the job reappears in the queue, where worker B takes it and delivers it. The job spends the whole episode either invisible under a live lease or visible and waiting, never lost.

Notice what the design refuses to depend on: the worker saying goodbye. A crash announces nothing. Recovery keyed to error messages misses the deaths that matter, so recovery is keyed to the absence of a signal instead. Silence is the one thing a dead worker reliably produces.

There is a catch, and it is why this chapter comes after Chapter 12. A lease that expired because a worker was slow, not dead, means two workers now run the same job. You will tune timeouts to make that rare. You cannot make it impossible, and a distributed fleet that promises exactly-once execution is lying to itself. The real contract is at least once, which is only safe because Chapter 12 made delivery idempotent: running the same job twice lands the same files under the same names, and nothing downstream doubles. Rescue without idempotency is a duplicate generator.

One habit turns all this into something you can trust: run the rescue sweep on a schedule, every few minutes, as a boring background chore. A recovery path that only executes during disasters is untested by definition. Mine runs all day and finds nothing, and the finding nothing is the test.

Scheduling the fleet

With the stages decoupled and the zombies handled, someone still decides which waiting job runs next. The default answer, first in, first out, has a fairness problem that shows up the first week you have two customers. One submits a 10,000-job catalog backfill at 9:00. Another submits 3 urgent jobs at 9:01. Under first in, first out, those 3 jobs wait behind all 10,000, and your smallest customer just learned that your biggest customer owns the fleet.

The fix is to stop pretending there is one line. Give each customer their own queue and serve them round robin, one turn each in a circle, or in weighted shares if contracts differ. Inside a customer's queue, let priorities exist: a small urgent refresh should overtake that same customer's own background backfill. And push heavy recurring work into off-peak windows, the source's quiet hours, which Chapter 2 already argued is kinder to the source; it also happens to be when the fleet has slack.

Scheduling cannot create capacity. The experiment below held its scheduling fixed at plain first in, first out, and its final limit says why that was fine: priorities and fairness reallocate waiting between jobs; they do not reduce the total. When the whole fleet is behind, no ordering saves you.

How many workers does the promise really take

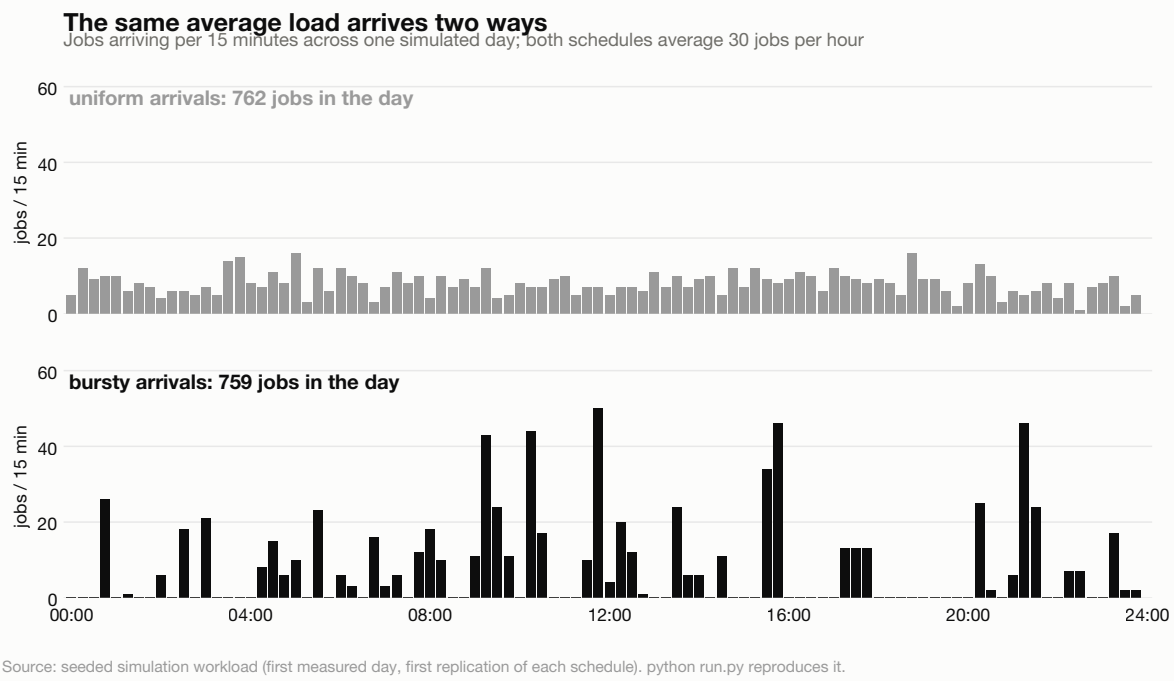
The last knob is the size of the fleet itself, and it hides the most expensive mistake in this chapter. The promise at stake is the one queues make visible: how long a job waits before it starts. Customers feel that wait directly, so it ends up in contracts as a service level agreement, an SLA. The experiment uses a common shape: 95% of jobs start within 5 minutes of submission.

The scenario's numeric assumptions sit in one editable file (`example/data/params.json`), and its two distribution shapes, geometric batch sizes and exponential service times, live in `run.py`: jobs arrive at 30 an hour on average, each takes 10 minutes of work on average, with individual jobs running shorter or longer the way real jobs do, so customers offer the fleet 5 hours of work every hour. A worker costs $65 a month, the scenario's assumed price for a small always-on cloud machine. The whole thing is seeded and offline; `python run.py` reproduces every number and both charts.

How many workers? There is a formula, and it is a century old. A. K. Erlang, a Danish engineer at the Copenhagen Telephone Company, worked out in 1917 how many operators an exchange needs so that callers rarely wait. Callers dial one at a time. The queueing mathematics named after him, Erlang C, answers our question, provided jobs arrive the way callers do, smoothly, one at a time at random moments, and run long or short the way calls do. For this load it says 8 workers, with a p95 wait of 4.0 minutes, inside the promise. The p95 wait is the wait the unluckiest 1 job in 20 experiences. And the formula is no relic: the experiment's discrete-event simulation, a program that replays the queue arrival by arrival, measures 3.8 minutes at those same 8 workers under smooth arrivals. A hundred years on, Erlang is still right about the world he described.

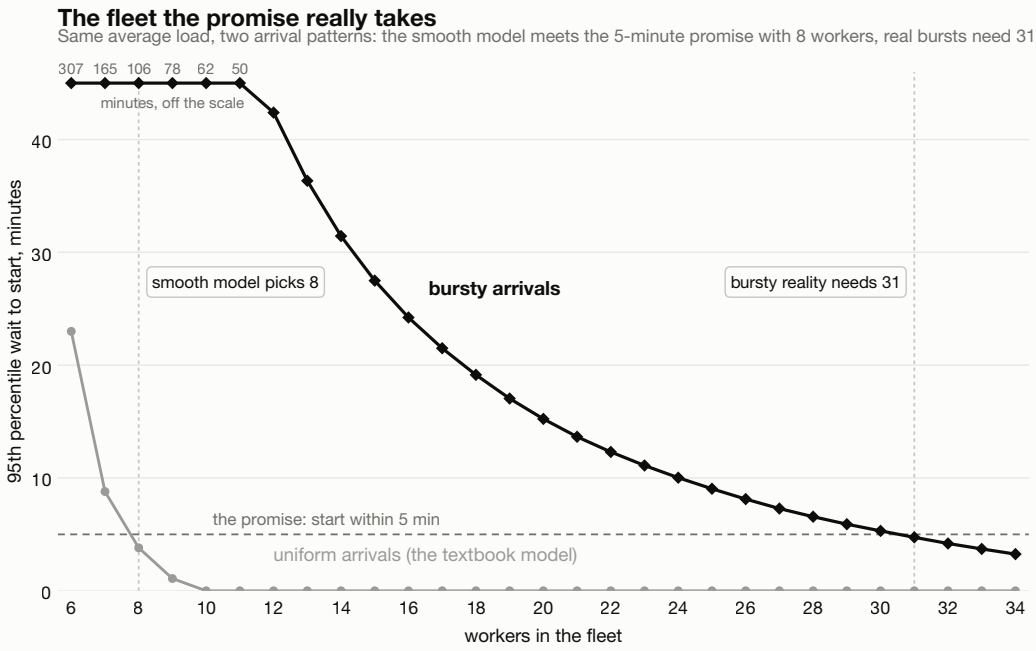
The trap is that fleets do not live in that world. Nobody submits jobs one at a time. A customer presses go on a category refresh and a dozen jobs land in the same second. So the experiment runs the same

average load a second way: batches land at random moments, each carrying a random number of jobs averaging 12. Statisticians call the arrival pattern compound Poisson. Same 30 jobs an hour. Same 5 hours of offered work. Only the clumping changes.



One simulated day of arrivals in 15 minute bins, the uniform schedule above the bursty one. The uniform day holds 762 jobs spread thinly across every bin. The bursty day holds 759, the same average, stacked into tall spikes with near silence between them. Every number in this section comes from the difference between these two textures.

Under the batched arrivals, the promise takes 31 workers, holding a p95 of 4.7 minutes with 95.3% of jobs inside the promise (202,287 of 212,352 across ten simulated months). The formula's answer of 8 undershoots by a factor of about 4, at the same average load, on the number customers feel.



Source: seeded discrete-event simulation, 10 replications x 30 days per point, at least 212,352 jobs each. python run.py reproduces it. At 8 workers the bursty p95 is 106 min.

The p95 start wait against fleet size for both arrival schedules, from the sweep of 6 to 34 workers, with the 5 minute promise as a horizontal line. The uniform curve dips under the promise at 8 workers. The bursty curve stays above it until 31. Same average load on both curves; only the clumping differs.

Run the 8-worker fleet the formula recommends under the real arrivals and the failure is not subtle. The p95 wait is 106.4 minutes against the 5 promised, 21 times the promise. Of 212,352 jobs, 149,679 miss it, 70.5% of everything submitted. And while that happens, the utilization dashboard reads 62%. Utilization, the share of time the average worker is busy, is exactly the number an average-load dashboard shows you, and at 62% it looks like a healthy shop with room to spare. Chapter 10 showed a green dashboard hiding rotting data. The same thing happens here on the operations side: a green dashboard hiding a broken promise.

The mechanism is a job's position inside its own clump. With batches averaging 12, some batches run long, and 10,767 of the 212,352 jobs, 5.1%, arrive 35 or more jobs deep in their own submission. Starting a job that deep within 5 minutes means clearing the 34 ahead of it within 5 minutes, and a busy worker frees up about once every 10 minutes. No plausible service speed clears a clump that fast. The only thing that does is workers already standing idle when the batch lands.

This is why the right-sized fleet looks absurd on every chart a finance review will ever produce. At 31 workers, utilization is 16%. At a random moment, about 5 of the 31 are busy. Those 26 idle workers are the promise, held in reserve for the next batch. And the promise has a price: at $65 per worker the formula's fleet costs $520 a month and the fleet that keeps the promise $2,015, a gap of $1,495 for identical average throughput. When you quote an SLA, that gap is what the SLA costs to be true, and Chapter 11's cost-per-record arithmetic should be charged with it.

That price will be challenged, so arm whoever defends it. In this sweep, cutting the fleet from 31 workers to 20 saves $715 a month and drops the share inside the promise from 95.3% to 82.3%, which is 2,756 more late jobs a month averaged over the ten simulated months, roughly four newly late jobs for every dollar saved. When a budget review calls the reserve idle, do not argue philosophy. Put the late-job count next to the savings and let the review own both numbers.

The brief's hypothesis, written before the code ran, was that burstiness, not average load, sets the fleet size, and that the smooth-arrivals model undersizes badly at the promise customers feel. The run confirmed both, 8 against 31. One finding I did not predict: the damage is not confined to the tail. At the 8-worker fleet the median wait is 18.4 minutes, so half of all jobs break the 5-minute promise. I had expected a story about unlucky jobs at the 95th percentile. The data says the typical job fails, and half of everything a customer submits arrives late.

The numbers come with three limits. The factor of about 4 belongs to this scenario; heavier batch tails push it up, capped batches pull it down, and the right move is to rerun the sweep with your own measured arrivals, which is what the parameters file is for. Elastic capacity, starting workers when a batch lands, shrinks the $1,495 rather than the worker count, because the burst still has to be served by capacity that exists at burst time. And even 31 is a pooled answer: single simulated months range from 3.7 to 5.9 minutes at the p95, so a contract read strictly month by month takes 33.

Read those limits again as a price list, because two of them are contract clauses. The arrival shape is the expensive one: both fleets serve the same 30 jobs an hour, so the $1,495 buys the customer's habit of landing a dozen jobs in the same second. A customer who agrees to pace a big refresh is cheaper to serve, which is a discount you can offer or a burst fee you can charge. The measurement window is the cheap one: reading the promise strictly month by month costs two extra workers, $130 a month. I price both on purpose now, instead of discovering them in a bad quarter.

Incident discipline

The bad night comes anyway. Queues, ceilings, leases, and a fleet sized for real arrivals make it rarer; nothing makes it never. Operations that survive their incidents agreed the steps in advance; the ones that skip that get defined by their incidents. And 3 a.m. is a bad time to invent process.

Start with a severity ladder, written down while everyone is calm. A SEV1 means a customer promise is being broken right now: deliveries failing, or wrong data flowing. A SEV2 means the promise is at risk but holding, a backlog growing faster than it drains. A SEV3 is internal pain nobody outside would notice. The ladder matters because at 3 a.m. everything feels like a SEV1, and the severity written next to each alarm is what stands between a team and treating them all the same.

Then the first three moves, in order. First, stop the bleeding: mitigate before you diagnose. Pause intake for the affected source so backpressure does its job, flip to the fallback method from Chapter 3's waterfall, let Chapter 6's breaker stay open. Understanding why can wait an hour; the meter of broken promises cannot. Second, one owner, speaking out loud: a single person runs the incident and posts

short factual updates on a fixed rhythm, including to affected customers. Chapter 12 made the case that customers verify you anyway; a customer who hears about the incident from you trusts the next delivery more, not less. Third, preserve the evidence before you heal the patient. Snapshot the queue depths and capture the logs before anything restarts, and write down the times, because a restart erases the evidence the write-up will need.

Tomorrow's job is the write-up, and the rule that matters there is that it names the mechanism rather than a culprit. That is mechanics as much as kindness: an operator who expects punishment stops reporting near-misses, and near-misses are the cheapest early-warning signal an operation ever gets; Chapter 10 priced what late detection costs. The write-up has three parts, a timeline, a mechanism, and a change: some ceiling, timeout, alarm, or runbook line that is different now. My one-line test for whether an incident is closed: point to the thing that changed. If nothing changed, the incident is still open, however long ago the outage ended.

Who notices first

I rank a data operation by one question: when something breaks, who notices first? The answers form a ladder, and every operation I have seen sits somewhere on it. On the first rung, a person watches it: dashboards and a morning scroll. Detection is whoever happens to be looking. On the second, the system pages a person: Chapter 10's alarms wired to queue depth, zombie counts, and the promise itself, the p95 start wait measured per customer on the same clock the contract uses. On the third, the system handles its own routine failures, the zombie sweeps and open breakers and backpressure of this chapter, and pages a human only for what it lost. On the fourth, the system proves its promises: delivery verified at the destination as Chapter 12 demanded, the start-time promise reported from measurement, the fleet resized when measured burstiness drifts from the assumption it was sized for.

One demotion protects the second rung: take utilization off the service page entirely. In the experiment it read 62% while seven of every ten jobs ran late, and 16% while the promise held, so it measured cost correctly both times and never measured service at all. My rule for the weekly review: the first fleet number anyone sees is the p95 start wait per customer, and utilization lives in the budget review, where it tells the truth.

Most operations I meet live on rung two and describe themselves as rung four. The tell is always the same: the promises are stated, never measured.

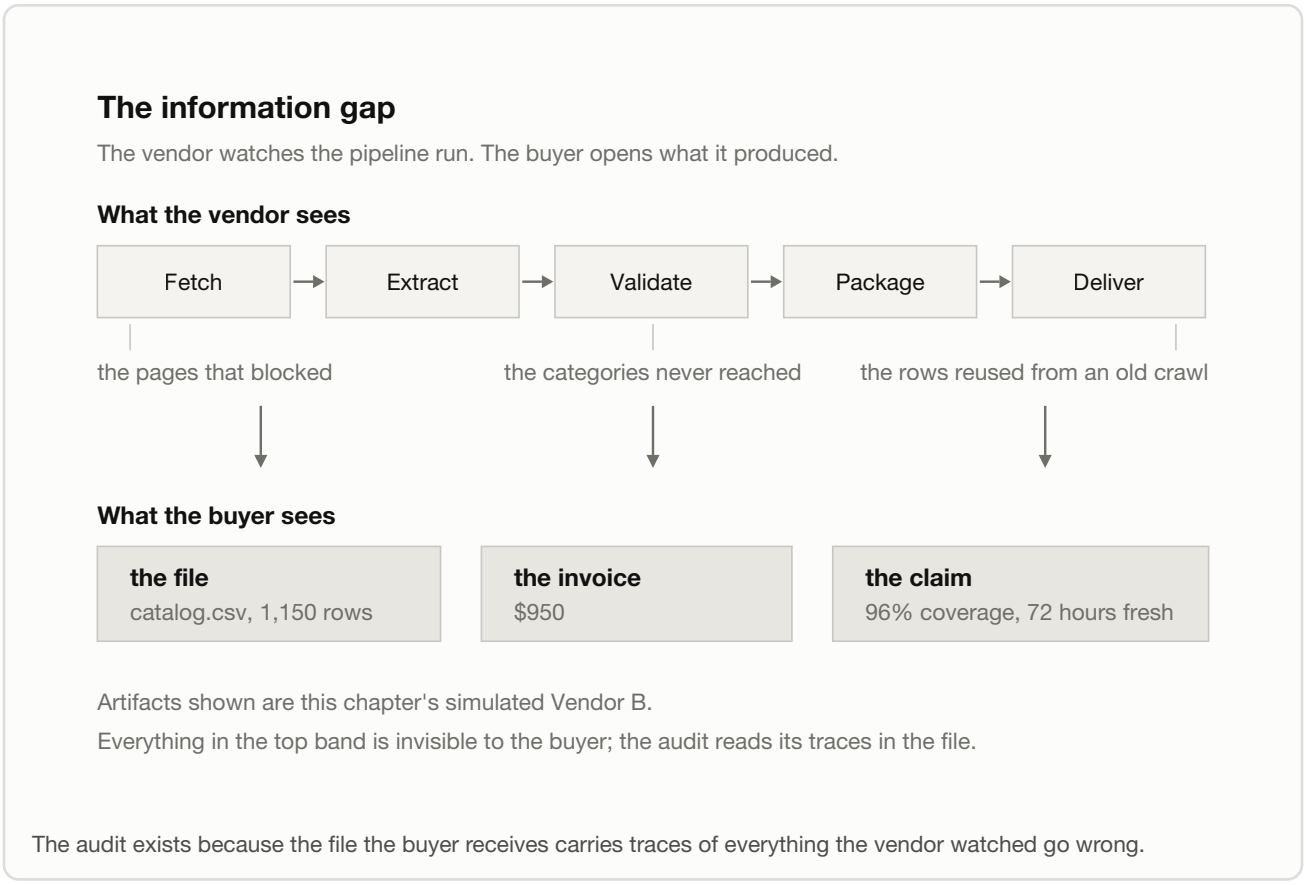
PART V. THE OTHER SIDE OF THE TABLE

Chapter 14. How to Buy Data Well

I sell data for a living. For thirteen chapters that worked in your favor; on this last one it makes me the party under audit, because the chapter switches chairs and teaches you to buy. I am writing it anyway, mostly out of self-interest. A buyer who can check the work steers money toward vendors who do the work, and a vendor with nothing to hide would rather compete on an audit than on a claim. The ones who lose when buyers learn these checks were never selling data in the first place.

Start with the shape of the disadvantage. Your vendor watched the collection run: which pages blocked, which categories the crawler never reached, which rows came from an old cache, what broke midweek and how it was patched. You see none of it. You receive a file, an invoice, and a claim, and the claim is usually a round, confident number. Economists call this information asymmetry, one side knowing more than the other about the goods. George Akerlof's Nobel-winning example was used cars, and the used-car world's working answer, the independent inspection before purchase, is what this chapter hands you.

The gap never fully closes, and it does not need to. What you need is this book's method held from the other end: checks you can run yourself, questions whose answers are hard to fake, the contract terms a measurement can trigger, and a rule for walking away. None of it needs code, and most fits in an afternoon.



The audit exists because the file the buyer receives carries traces of everything the vendor watched go wrong.

An audit a spreadsheet can run

Ask every vendor on your shortlist for a sample delivery against your real specification; some will refuse, and that is worth knowing before any money moves. When the sample lands, run five checks. They are Chapter 8's validation discipline with the roles reversed, and each ends in one number against one limit, which later lets the same five lines go into a contract.

Fill. One COUNTBLANK per promised column, the spreadsheet function that counts empty cells; every required column must come back at least 98% filled. It catches broken extraction and forgotten fields, and it is the weakest of the five, because a fabricated row fills every cell beautifully. Run it first and trust it least.

Duplicates. Add three helper columns: the SKU (the product code) in upper case with spaces and punctuation stripped, the product name in lower case with punctuation stripped, the URL cut at the question mark. One COUNTIF per helper column, counting matches among the rows above, flags every collision. More than 2% of rows colliding fails. Honest pipelines leave a little residue; padded files leave a lot, because a disguised copy is the cheapest padding there is.

Freshness. Split the timestamp column and count rows whose time part reads exactly 00:00:00. More than 5% fails; the allowance covers schedulers that genuinely fire at midnight. A crawler stamps 14:05:09 because it ran at 14:05:09; a person re-dating a column types 2026-08-12 and the clock silently becomes midnight. While you are there, count the distinct days and set them against the claimed crawl window.

Impossible rows. Write down rules the real world enforces and flag rows that break one: a strikethrough price, the crossed-out was-price next to a discount, must sit above the selling price; a rating cannot rest on zero reviews; one SKU cannot hold two prices at the same moment; price and stock must hold legal values. More than 1% flagged fails. Real catalogs produce a few of these through source bugs, and padding produces them in bulk.

The coverage claim. Homework first: an afternoon on the public source, writing down 60 products by hand, spread across categories, popular and obscure. Look up each of the 60 in the delivery with VLOOKUP, the spreadsheet function that finds a value in another table. The hit rate is the vendor's coverage as measured by you, and a claim more than 10 points above it fails. Ten points, because a 60-item sample carries its own noise: at 95% confidence near an 80% true rate the sampling margin is about 10 points, so the tolerance forgives sampling luck and nothing more. The overlap also carries Chapter 9's capture-recapture trick, pointed the other way: the delivery's distinct product codes times 61, divided by your hits plus 1, estimates the size of the catalog the file was really drawn from.

The sample audit on one page

Every check ends in one number against one limit; no code anywhere

1. Required-field fill — **pass: at least 98%**

One COUNTBLANK per promised column; every required column must clear the line.
Runs first, trust it least: fabricated rows fill every cell.

2. Duplicate share — **pass: at most 2% of rows**

Helper columns: SKU upper case, spaces and punctuation stripped; name lower case, punctuation stripped; URL cut at the question mark. One COUNTIF per column.

3. Freshness — **pass: at most 5% of rows**

Count timestamps whose time part is exactly 00:00:00.
A crawler stamps odd seconds; a hand-written date has no clock.

4. Internal consistency — **pass: at most 1% of rows**

Flag impossible rows: a strikethrough price at or below the price, a rating with zero reviews, one SKU carrying two different prices at the same moment.

5. Coverage claim — **pass: gap at most 10 points**

Hand-gather 60 products from the public source, then VLOOKUP each against the delivery. The hit rate is your measured coverage; set it against the claim.

Bonus arithmetic: distinct product codes times 61, divided by hits plus 1, estimates the catalog size the file behaves like. A padded file overshoots its own manifest.

Limits are set so ordinary residue passes and padding does not; the same five lines return below as contract acceptance terms.

Two vendors, one of them padded

Would this audit catch a real cheat? The experiment stages it. Both vendors are simulations, generated from a fixed seed (`python run.py` reproduces every number and both charts offline), with padding rates chosen to be plausible rather than measured from any real vendor. Simulation is the only fair option: with real deliveries I could never publish the answer key, and the answer key is what grades the audit itself.

The stage is a catalog of 1,200 products whose true size only the grader knows. Vendor A delivers 989 rows for $1,200 and claims 84% coverage. Its file is what a good real crawl looks like: misses clustered behind two categories' pagination walls, 5 leftover duplicate rows, 3 rows where the source page showed no brand, 2 with a mis-set strikethrough price, real blanks where products have no reviews or discount. Vendor B delivers 1,150 rows for $950 and claims 96%, "every row refreshed within 72 hours." Its file is built the way padded files really get built. A fresh core of 400 truly crawled rows.

Then 260 stale rows re-dated to look fresh. Then 190 copies of rows already in the file, lightly disguised: a case flip on the SKU, an appended suffix, a tracking parameter on the URL, a price moved by one cent. Then 300 products fabricated from a template. Count rows and B is the fuller file at the better price.

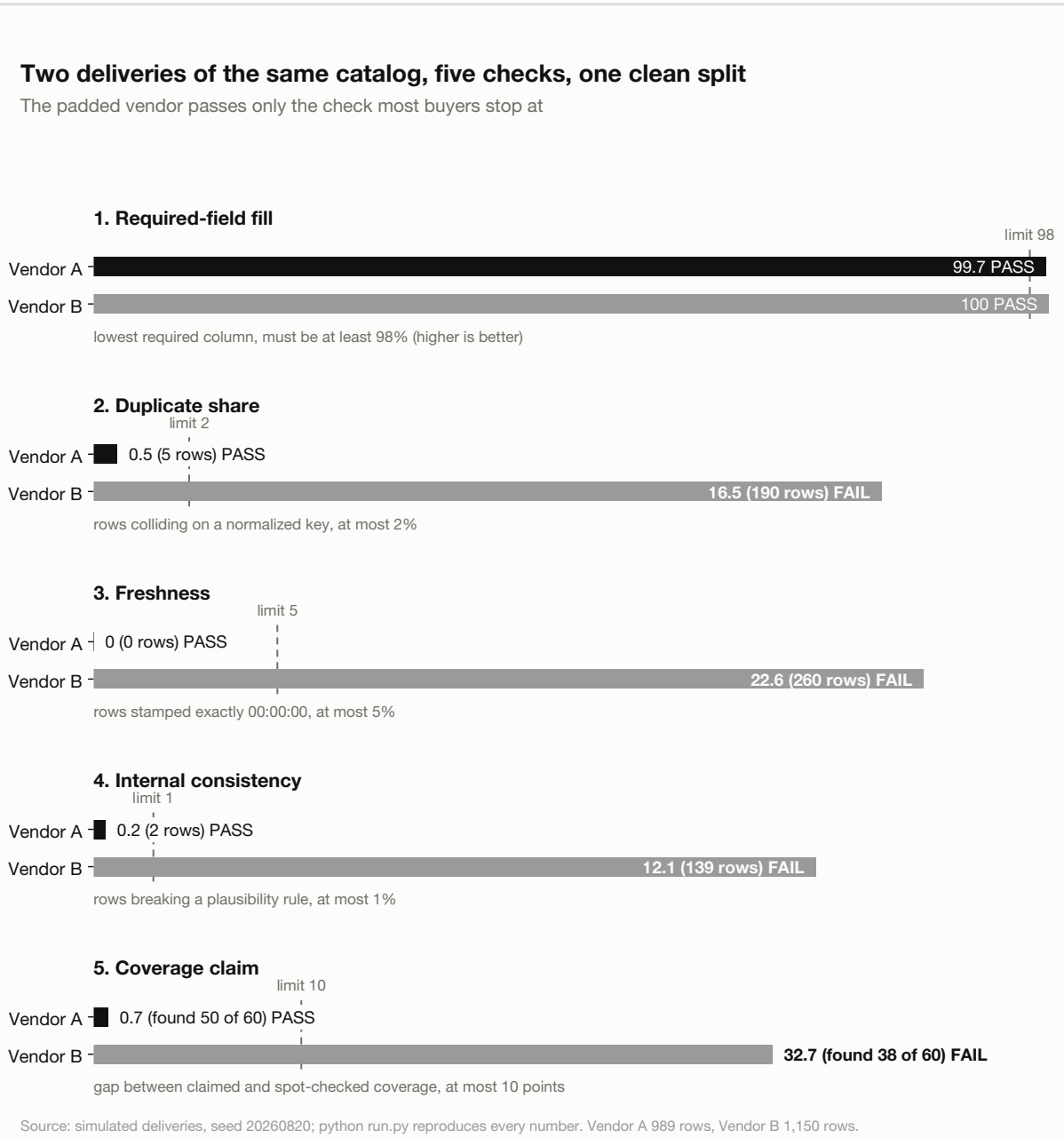
The audit breaks the tie in four moves.

Fill flatters the cheat, as warned. Vendor B scores 100.0% on every required column, because fabricated rows never leave a blank. Vendor A's weakest required column is brand at 99.7%, those 3 pages that showed none. Both pass the 98% line. If the audit stopped here, and most stop here, the padded file wins.

The duplicate check ends the illusion. Vendor A: 5 of 989 rows collide on a normalized key, 0.5%. Vendor B: 190 of 1,150, 16.5%, eight times over the 2% limit, because every disguised copy collides with its source row on at least one helper column. A disguise that fools an eyeball does not survive an upper case and a strip.

Then the clock. Vendor A has 0 of 989 rows stamped at exact midnight, 0.0%, and its timestamps spread across 6 distinct days matching the crawl window it claimed. Vendor B has 260 rows at exactly 00:00:00, 22.6% against the 5% limit, bunched on just 2 of its 3 claimed days. Those are the re-dated stale rows. The prices they carry are months old, which no formula can see; the timestamps gave the re-dating away.

The impossible-row rules flag 139 of Vendor B's rows, 12.1% against a 1% limit: 47 rows with a strikethrough price at or below the selling price, 50 rows carrying a rating with zero reviews, and 49 SKUs holding two different prices at the same moment, that last group being the one-cent disguise contradicting its own source row. Seven fabricated rows break two rules at once, which is why the three counts add past the total. Vendor A has 2 flagged rows, 0.2%, its two mis-set strikethroughs, the small residue the 1% allowance exists for.



The padded vendor wins only the fill check, the one most buyers stop at; each of its four failures lands at a multiple of the limit.

Before any of this ran, the brief committed to a hypothesis: the padded vendor fails at least 3 of the 5 checks on internal evidence alone, no ground truth needed. Judged against the run: confirmed, right at its bar. The 3 failures that use nothing but the delivered file itself, duplicates, freshness, and impossible rows, are the at-least-3 the hypothesis named, and the spot list adds a fourth from outside. Vendor A passes all 5, which matters just as much, because an audit that condemns everyone teaches nothing.

Your 60 rows against their 96%

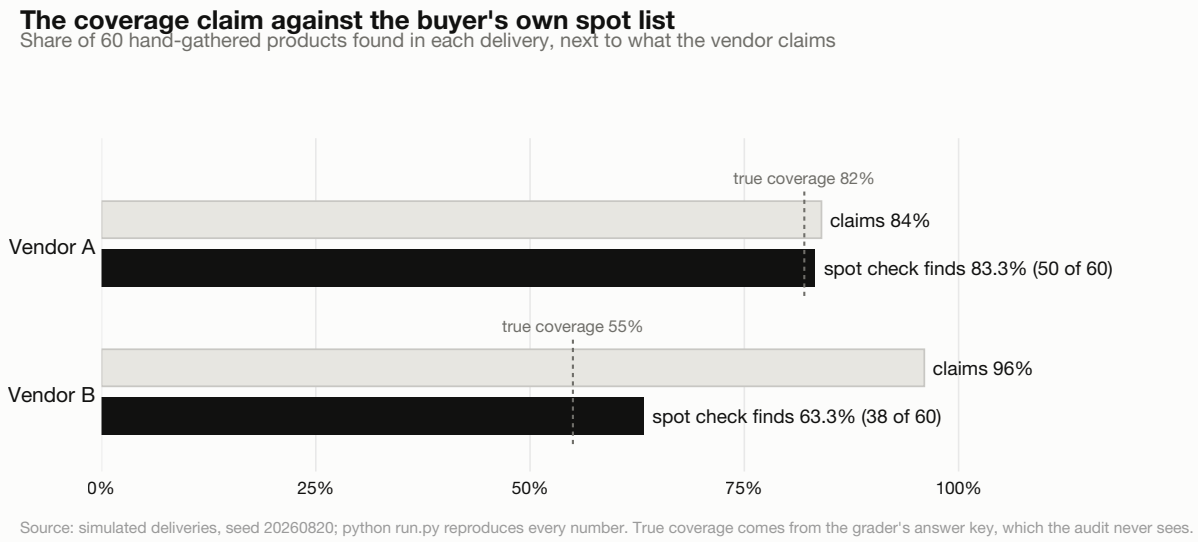
The fifth check catches the one trick the other four cannot.

Look up the 60 hand-gathered products in each delivery. Vendor A: 50 of 60 found, 83.3% measured against an 84% claim, a gap of 0.7 points. Vendor B: 38 of 60 found, 63.3% measured against a 96% claim, a gap of 32.7 points, three times the tolerance. The grader's answer key, which the audit never sees, says the little list read both fairly: Vendor A truly covers 984 of the 1,200 products, 82.0%; Vendor B, 660 of them, 55.0%.

One more calculation from the overlap. Vendor A's overlap implies a catalog of about 1,177 items, close to the true 1,200. Vendor B's 1,021 distinct product codes against its 38 hits imply about 1,597. (Why 1,021 codes when check 2 left only 960 fully unique rows: 61 of the disguised copies changed the product code while still colliding on name or URL. Run the estimate on either base and the manifest is contradicted.) Hold that against B's own paperwork: its manifest states a catalog of 1,200 items, its claim needs 96% of that, and its file behaves like a sample of a universe around 1,600 items. The padding failed to support the claim and then went further, manufacturing a second, contradictory claim inside the same delivery.

There is a five-minute version of this check that needs no hand list at all. Count the rows that survive the duplicate check, then divide by the catalog size the vendor's own manifest states, or better, by the count the public source displays on its own category pages. That number is a hard ceiling on any coverage claim. Vendor B has 960 unique rows against its own stated catalog of 1,200, a ceiling of 80%, so its 96% claim dies before the spot list is even opened. Vendor A's 984 unique rows against its stated 1,170 read 84%, matching its claim. I run this division on every quote before I spend the afternoon.

The real reason the spot list exists: of Vendor B's 300 fabricated rows, 213 survive every row-level check in this audit, because a template can be internally consistent, prices above strikethroughs, ratings resting on plausible review counts, every cell filled. Read one at a time, they are unimpeachable. What they can never do is show up on a list you gathered from the real catalog, so in aggregate they dilute the hit rate and inflate the implied catalog size, and the cross-check converts their existence into evidence. Fabricated rows hide one by one and surface all at once.



The spot list reads Vendor A at 83.3% against a true 82.0% and Vendor B at 63.3% against a true 55.0%; the 96% claim is out of reach of either number.

Two limits. The midnight test catches lazy re-dating; a careful padder randomizes fake times and beats check 3, so the verdict rests on the two things hardest to fake, the duplicate structure and the coverage arithmetic. Against careful re-dating the leftover defense is manual: open a dozen delivered rows on the live site and compare prices. And a fraud built to pass all five would have to actually crawl the catalog fresh and wide, at which point it has become the product it was imitating, which was the point all along. The audit cannot make cheating impossible, but it can make cheating cost more than honesty.

The price per row you actually pay

Sticker prices said Vendor B was the bargain: $950 for 1,150 rows is $0.83 per row, against Vendor A's $1,200 for 989 rows, $1.21. Drop every row the audit flagged and the order flips. Vendor A keeps 982 of its 989 rows, so its usable rows cost $1.22 each, one cent over sticker. Vendor B keeps 650 of 1,150, and its usable rows cost $1.46 each. The quote that looked 32% cheaper per row now runs 24 cents per row more than Vendor A's.

The grader sees one level further than any real buyer could. 213 of B's 650 surviving rows are fabricated, so the rows that are both usable and real cost $950 for 437 rows, $2.17 each. Chapter 11 crowned cost per successfully delivered record the master metric of a collection operation. This is its buyer-side twin, price per genuine usable row, and the sample audit gives you a first estimate before anything is signed.

Seven questions, and how to read the answers

So far you have tested a file. These questions test the operation behind it, each mapping back into this book: you are checking whether the vendor runs the operation these chapters described, without seeing it. Put all seven in writing, and read the answers for numbers and documents you could ask to see.

1. How do you know when your collection breaks? You want monitoring built on the data itself, fill rates that sag and distributions that shift, with alarms that learn each field's normal wobble (Chapter 10). "Our customers tell us" means the monitoring is you.
2. Show me last month's quality report for a customer like me. An operator produces a redacted real one, because one already exists for every delivery; redacting it costs a real vendor an hour, and I know because I have spent that hour. If yours would be the first ever written, the product you are quoting does not exist yet.
3. How do you bill failures? Chapter 6 drew its forbidden edge here: never fabricate a result from a failure. The billing version: does a blocked page cost me money, and how would a made-up success be caught before my invoice? A vendor who charges per attempt has moved collection risk onto your budget. This is the question I would brace for in the seller's chair, because per-attempt billing is comfortable to run and uncomfortable to defend out loud.
4. What happens to my delivery when a source redesigns? A real answer names mechanisms and a window: stored raw responses that re-parse without re-fetching (Chapter 7), and a declared gap while the extractor is rebuilt. "It rarely happens" means there is no plan.
5. How do I verify a delivery landed complete? You want a manifest with every delivery and verification at the destination storage rather than a status flag, Chapter 12 played from your side of the net.
6. When something goes wrong, what exactly gets re-delivered, and who notices first? Re-delivery must be safe to repeat without doubling rows, and "who notices first" should be them, through Chapter 13's incident discipline.
7. What exactly may I do with this data? The license question, and the next section's subject. Watch the answer's speed as much as its content: a vendor who must go ask someone upstream has just told you the collection is not theirs.

Resellers, firms that buy data flows from others and rebadge them, stumble the same way on questions 1, 4, and 6, which only the party operating the collection can answer. Buying from a reseller is sometimes fine, aggregation can be real value, but you should know which kind you are talking to, and the price should reflect it.

Contract terms a measurement can trigger

An SLA, a service level agreement, is the part of the contract that attaches consequences to promises. Most data SLAs promise "high quality" and "timely delivery," which no measurement can trigger, so nothing ever triggers. Rewriting one mostly means importing what this book already built.

- A delivery window with a date, and a completeness manifest with every delivery, the packing slip of a data shipment: file names, row counts per file, checksums, crawl window, delivery date.
- Acceptance defined as the audit: the five checks and their limits, written in as what "delivered" means. You have already rehearsed enforcing them.
- Automatic recredits: rows that fail acceptance, or deliveries that miss the window, convert to credit at the contracted per-row price, no negotiation per incident.
- The quality report arriving with every delivery, not on request.
- Exit portability: at termination you keep the delivered data and manifests under the license, and the vendor owes one final complete delivery. Switching should cost a procurement cycle. Your history stays yours.

A vendor who signs measurable terms is telling you their numbers already clear the bar. Hesitation in front of this list is free information.

One warning before you write check 5 into the acceptance clause: keep the spot list out of it. Write in the mechanism, a buyer-supplied list of 60 items checked against each delivery, but never the items, and draw a fresh list for each acceptance round. The reason sits in the experiment: 213 of Vendor B's 300 fabricated rows pass every row-level check, so the spot list is the only line that catches invention, and a vendor who knows the 60 products in advance can crawl exactly those for real and fabricate the rest, and the audit stops costing cheaters anything. A fresh list is the same afternoon of gathering it took the first time; rotate half of it if a full redraw feels heavy.

Usage rights are a separate purchase

Chapter 2 drew the collection lines and parked one question for this chapter: whether a record was collected lawfully says almost nothing about what you may do with it. Use rights come from your contract, and four clauses do most of the work.

- Internal use versus redistribution. Analysis inside your walls is the default grant. Publishing extracts, reselling, or feeding a customer-facing product each needs its own words.
- Model training. If the data might ever train a model, get the right named, and in writing whether the trained model survives the contract.
- Retention. How long you may keep the data after termination, and what "delete" covers.
- Derived data. Whether aggregates and models built on the data outlive the contract. If your dashboards die with the contract, price that in now.

Chapter 2 already did half of this work. Its eleven-line checklist reverses into a vendor questionnaire: send it, ask for written confirmation line by line, and watch the account and personal-data lines for hesitation. Its hiQ ending doubles as your vetting script, a vendor's legal exposure being your supply risk: ask whether accounts touch your collection path and what happens to your deliveries if a cease-and-desist arrives mid-contract. If training is anywhere in your future, ask for the provenance record that chapter taught collectors to keep, the dated robots.txt stored beside each batch. The vendor who can produce it planned to be asked.

Walking away

Some purchases fail the audit. Others should never reach it. Three exits, each with a document or a number behind it.

The source is hostile. If delivering your specification at the claimed freshness would require what Chapter 2 ruled out, accounts on the collection path or evading a site's defenses, then every vendor quoting you is either quietly under-delivering or carrying a risk that will eventually become your outage. Against a hostile source, I treat the most generous quote as the most alarming document in the folder.

The rights are murky. A vendor who cannot put your intended use in writing is selling you a dispute with a delay on it, and no sample quality changes that.

The economics are negative at real volumes. Rerun Chapter 1's build-versus-buy model with real quotes instead of list prices. That model put the crossing at 316,266 records a month at list, then watched its own answer swing between 106,469 and 746,168 as the vendor's price moved between double list and half, the strongest lever in its sensitivity run. Three written quotes against one specification turn folklore into arithmetic inside a week. And when even the best quote costs more than the decision it feeds is worth, take Chapter 1's quiet third option: no pipeline and no contract, an analyst with a spreadsheet and an afternoon a week, or an official API's free tier. Declining to buy and declining to build is a finding too. File it like one.

That is the book. It opened with a deal: every real question settled by an experiment you can rerun, the prose following the experiment's output rather than my instinct, and wherever a run contradicted me, the run wins. It collected on that promise early, when Chapter 1's model overruled my instinct on the book's first question, and the text kept the correction. Ask a question that has an answer, write down what you expect before you look, run it, keep the numbers beside the code, and say what happened in words the person paying for the work can read. The same loop built every collector in this book and just audited two vendors, and it does not care which chair you sit in. The tools in these pages will age long before the loop does. Check my work. Everything here reruns.