ProxyHorizon

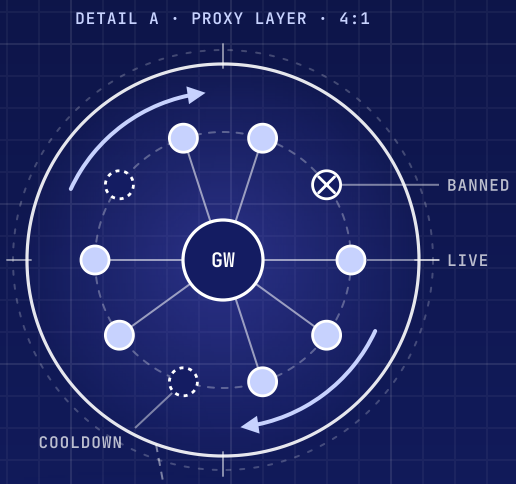
2026 EDITION

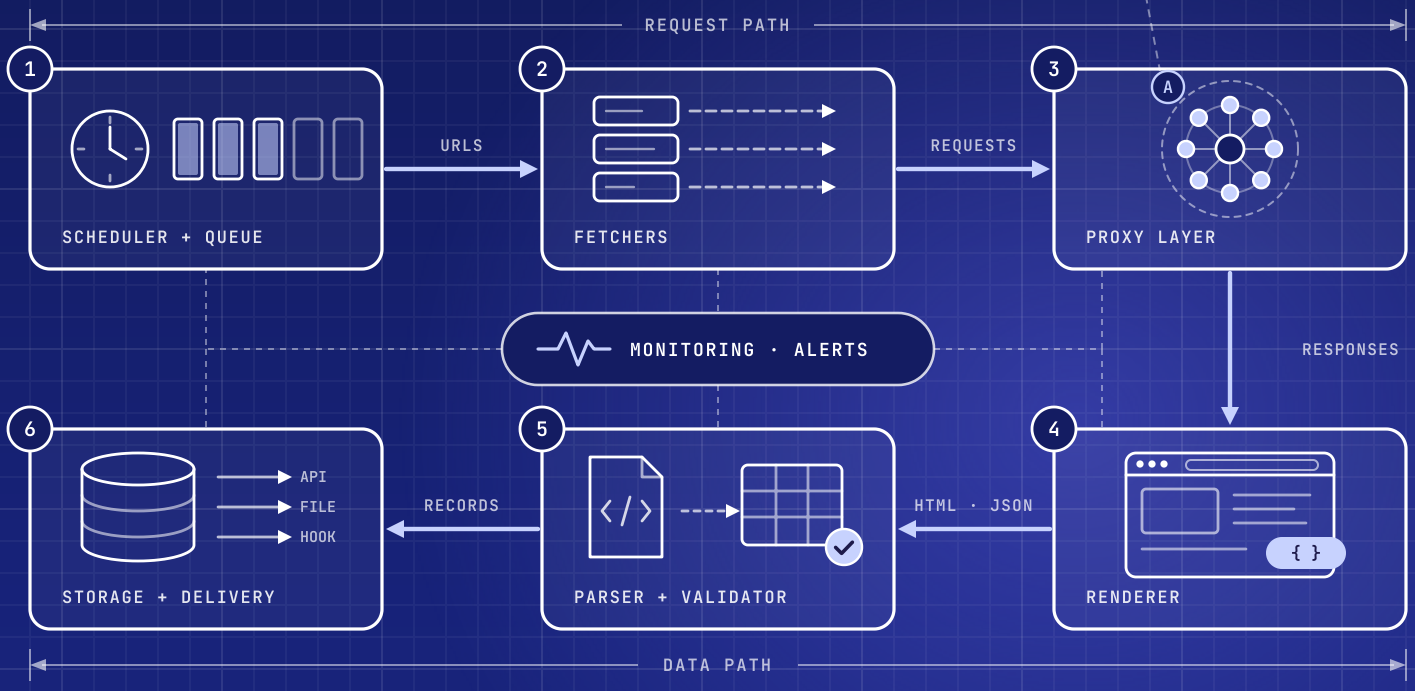
A PROXYHORIZON ENGINEERING GUIDE

The Scraping Blueprint

Build Powerful Data Extraction Systems

Design, build and run scrapers that stay reliable, affordable, hard to block and responsible.





LEGEND

DATA FLOW — MONITORING BUS

(1) LAYER — (A) DETAIL VIEW

TITLE THE SCRAPING BLUEPRINT	REVISION REV. 2026
DRAWING REFERENCE ARCHITECTURE	SHEET 01 OF 11
DRAWN BY PROXYHORIZON.COM	SCALE NOT TO SCALE

- A reference architecture, layer by layer
- Working Python for httpx, Playwright and asyncio
- Checklists, cheat sheets and cost models

BEFORE YOU BEGIN

About This Blueprint

Most scraping tutorials stop at the happy path: one script, one page, one CSV. This book covers everything around that script, the queue, fetchers, proxies, browsers, parsers, storage and monitoring that turn it into a system you can trust with real decisions.

Who It's For

Developers and data engineers

You write the scrapers and get paged when they break. You want fewer blocks and code that survives a redesign.

START WITH chapters 3, 4, 5 and 8

Technical founders

You're deciding whether to build a scraping stack or buy the data, and you want real costs before you commit.

START WITH chapters 1, 7 and 10

Data and analytics teams

You own the tables downstream and need fresh, deduplicated, validated records, with a contract to match.

START WITH chapters 5, 9 and 10

SEO and growth engineers

You track rankings, prices and competitors across many sites and countries, on a schedule that can't slip.

START WITH chapters 2, 6 and 7

How to Read It

Read Chapter 1 first: it introduces the six layers and the system map on every chapter opener. Then jump to whichever layer hurts most. Every chapter stands alone and ends with key takeaways. New to proxies? Read The Ultimate Proxy Handbook before Chapter 7.

Pro tip · Note · Watch out · Risk · Pro move · Go deeper

2026 Edition. Provider figures (pool sizes, coverage and entry prices) come from the ProxyHorizon directory as of November 2026. Prices move often, so check proxyhorizon.com/proxies for live numbers. Code targets Python 3.10+ with httpx, Playwright, lxml and pydantic v2. Hosts such as `shop.example.com` and `proxy.example.com` are placeholders. Nothing here is legal advice.

© 2026 ProxyHorizon. Share this PDF freely, unchanged and with credit.

YOUR MAP

Contents

	From Script to System Why scrapers break, and what this blueprint fixes.		04
01	**The Anatomy of a Scraping System** Six layers, a maturity ladder and the build-or-buy call.	All layers	05
02	**Recon: Read the Target Before You Write Code** Robots, sitemaps, hidden APIs and a page-type inventory.	Target	09
03	**Fetching Pages Reliably** HTTP clients, headers, timeouts, retries and backoff.	2 · Fetcher	14
04	**Rendering JavaScript Without Burning Money** When to launch a browser, and how to keep it cheap.	4 · Renderer	20
05	**Parsing and Extraction That Survives Redesigns** A selector hierarchy, typed schemas and fill rate.	5 · Parser	24
06	**Crawling at Scale** Frontiers, canonical URLs, dedup and throughput math.	1 · Queue	29
07	**The Proxy Layer** Type ladders, sessions, pool health and nine networks.	3 · Proxy	34
08	**Staying Unblocked** How anti-bot systems score you, and the escalation ladder.	2 · 3 · 4	40
09	**Data Quality, Storage and Delivery** Validation, drift, change detection and data contracts.	5 · 6 · Storage	44
10	**Running It in Production** Scheduling, SLOs, runbooks and a full cost model.	Monitoring	48
11	**Scraping Responsibly** Public data, terms, personal data and rate etiquette.	Every layer	52
A	**Appendix** Pre-launch checklist, cheat sheets, glossary and tools.		56

INTRODUCTION

From Script to System

The first scraper always works. Forty lines of Python, one CSS selector, and by lunch there's a CSV of competitor prices on your desktop. Then someone asks for it every morning, for six more sites, in four countries, with an alert when a price drops.

That's when a script has to become a system, and it's where most scraping projects stall. Requests fail at 3 a.m. A redesign quietly empties a column. The proxy bill doubles because a headless browser downloads every image. Nobody notices for a week, because the job still exits with code 0.

> **A scraper is a script. A scraping system is a promise that the data will be right tomorrow.**

This book is the blueprint for keeping that promise. It splits a scraping system into six layers, shows what each one owns and how it fails, and gives you working patterns for all of them.

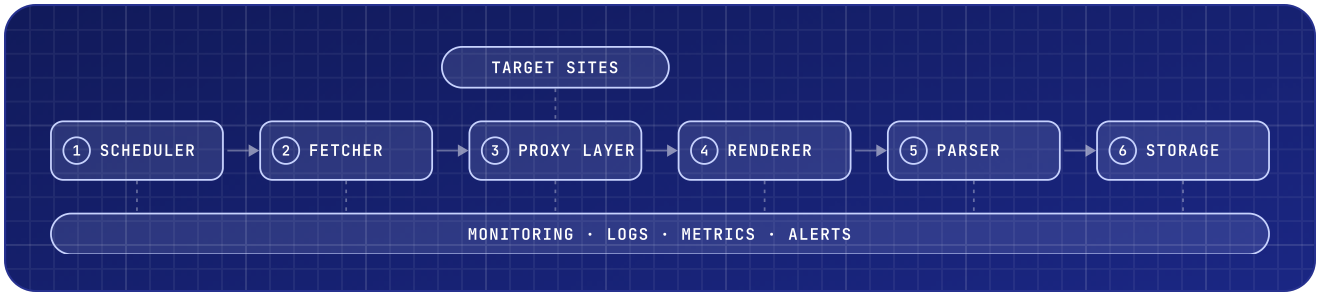


Figure 0.1 The system map. Every chapter opener carries it, with the layer that chapter covers lit up, so you always know where you are in the machine.

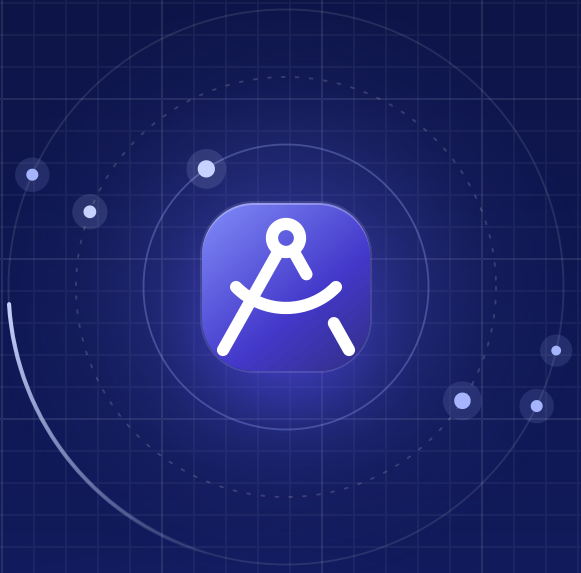
By the End, You'll Be Able To

- Read a target and find the cheapest route to its data before you write a line of code.
- Build fetchers that retry, back off and stay polite, with a browser only where it pays.
- Write extraction that survives redesigns and validates every record it emits.
- Crawl millions of pages with a frontier, per-host limits and deduplication.
- Route each target to the cheapest proxies that work, and know when to stop escalating.
- Run it all in production with SLOs, runbooks, a cost model and a clear conscience.

Score your own system as you read

Rate each layer 1 to 5 for how well your setup handles it today. Your lowest score is your first chapter.

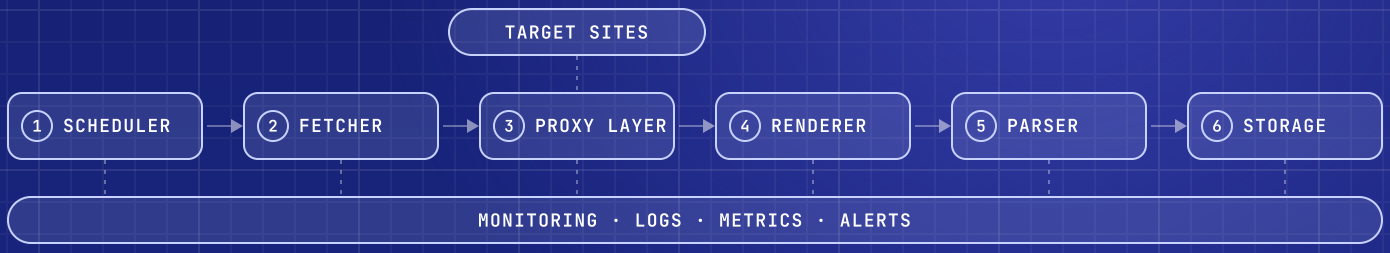
01



The Anatomy of a Scraping System

Every scraper that survives production grows the same six parts, whether anyone planned them or not. Name them, give each one a single job, and you can fix, scale or swap any part without rewriting the rest.

YOU ARE HERE THE WHOLE SYSTEM · ALL SIX LAYERS PLUS MONITORING



IN THIS CHAPTER

1.1 The six layers
1.2 Static and dynamic targets
1.3 The maturity ladder
1.4 Build, buy or both

1.1 The Six Layers

Strip away the frameworks and every production scraper does the same six jobs. Some teams run them inside one Python process; others spread them across a dozen services. The shape doesn't change, and neither do the ways each part breaks.

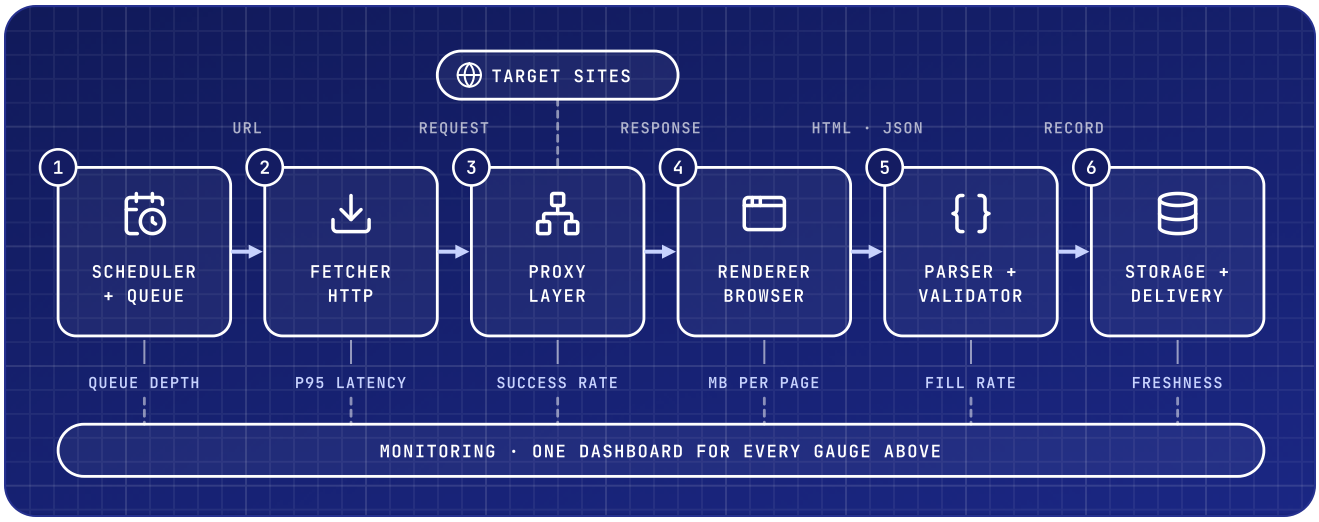


Figure 1.1 The reference architecture used throughout this book. Under each layer is the one number that tells you it's healthy; monitoring collects all six.

Layer	Its one job	Typical tools	How it breaks
1 · Scheduler and queue	Choose what to fetch, when	cron, Celery, SQS	Floods, duplicate work
2 · Fetcher	Send, retry, time out	httpx, aiohttp, Scrapy	Retry storms, 429s
3 · Proxy layer	Pick exit IP, session, location	Provider gateways	Bans, bandwidth bills
4 · Renderer	Run JavaScript when needed	Playwright, Puppeteer	Memory leaks, big bills
5 · Parser and validator	Turn pages into typed records	lxml, parsel, pydantic	Redesigns, silent nulls
6 · Storage and delivery	Keep history, ship data	Postgres, Parquet	Duplicates, schema drift
Monitoring	Watch it all, wake someone	Prometheus, Grafana	Green jobs, bad data

Keep the seams, even in one script

You don't need six services. You need six functions with clean inputs and outputs. When the proxy layer is one function, switching providers is a config change instead of a rewrite.

Debug in layer order

When a field goes missing, walk the map left to right. Was the URL scheduled, fetched, blocked, rendered, parsed, stored? The first "no" is your bug.

1.2 Static and Dynamic Targets

The biggest cost decision in scraping is whether you need a browser, and the target makes it for you. Look at what the server sends in its *first* response. Pages come in three flavors, and only one really needs JavaScript.

Server-rendered

```
<li class="product">
  <h3>Trail Runner 4</h3>
  <span itemprop="price">
    89.00</span>
</li>
```

The data is in the HTML. An HTTP client and a parser are all you need.

Hydrated

```
<script id="__NEXT_DATA__"
  type="application/json">
{"product": {"name":
  "Trail Runner 4",
  "price": 89.0}}
```

The data ships as JSON inside the page. Fetch over HTTP, then parse the JSON, not the DOM.

Client-rendered

```
<div id="root"></div>
<script src="/app.js">
</script>
// data arrives later from
// GET /api/products?page=1
```

The HTML is an empty shell. Call the JSON API it uses, or render it in a browser.

Figure 1.2 The same product page, three ways. Only the third needs a browser, and even then the API behind it is usually the better door.

The 30-second test

Open **View Source**, not DevTools' Elements panel, which shows the DOM *after* JavaScript ran. Search for a value you can see on screen, like a price. Found it? Plain HTTP will do. If it only appears in Elements, JavaScript put it there, and the Network tab shows which request delivered it. Chapter 2 shows you how to call that request yourself.

1.3 The Maturity Ladder

Projects climb the same four rungs. Climb only when your current rung costs more than the next.

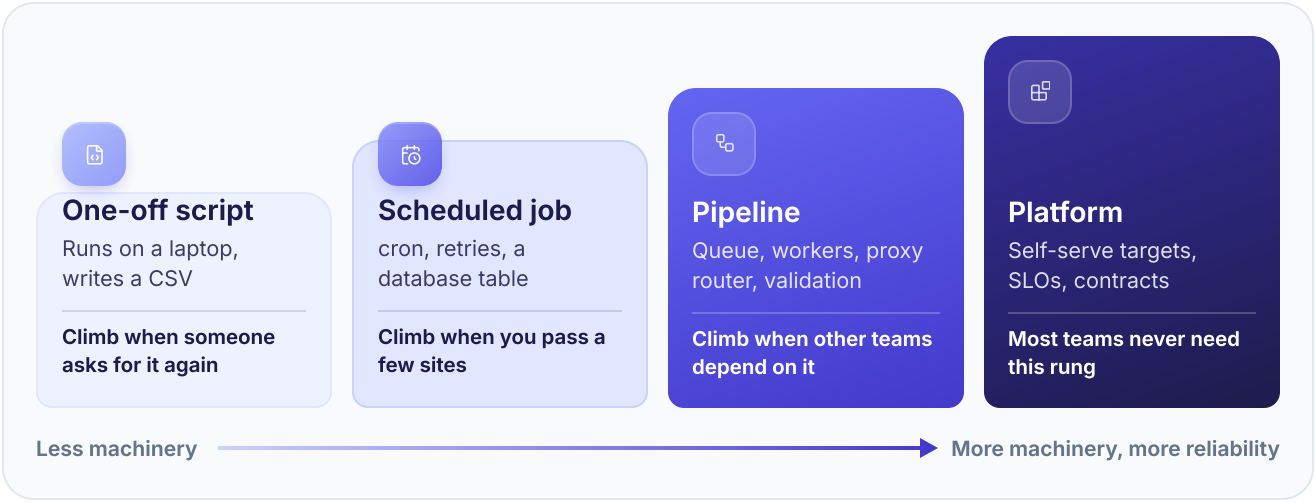


Figure 1.3 Skipping rungs is how teams end up running Kafka for three websites.

1.4 Build, Buy or Both

Before you build anything, decide how much of the stack you actually want to own. There are three ways to get web data, and they trade control for convenience in a fairly predictable way.

	DIY stack	Scraper API	Ready-made dataset
You send	Your requests, your proxies	A URL; HTML or JSON returns	Nothing; you download data
You own	All six layers	Parsing, storage, scheduling	Storage and analysis
You pay for	Proxies, compute, engineers	Successful requests	Records or a subscription
Wins when	Volume is high, data is core	Targets fight back	The source is common
Watch for	Maintenance never ends	Unit price at scale	Freshness and coverage

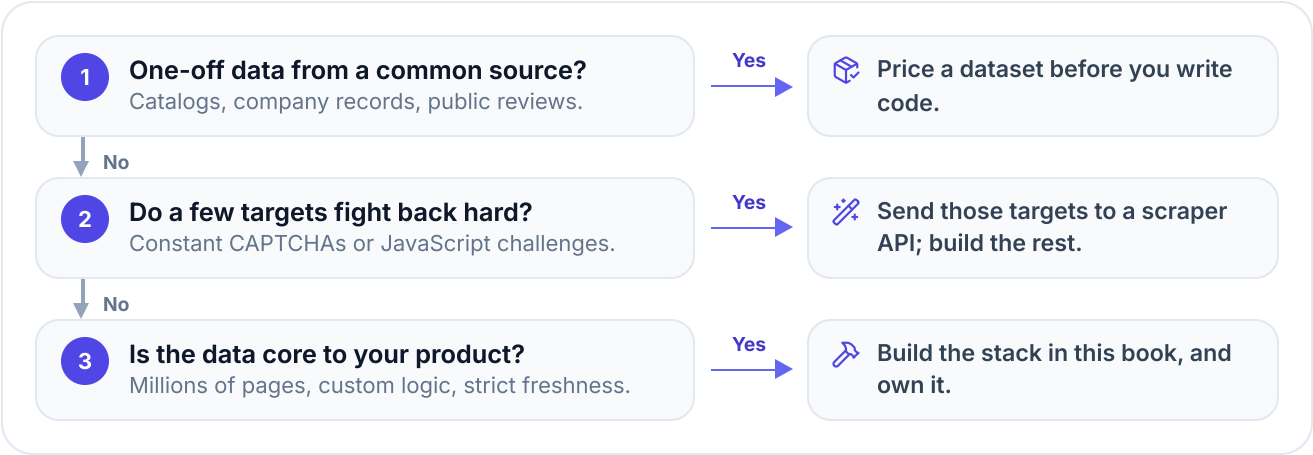


Figure 1.4 Three questions that settle most build-or-buy debates. The middle answer, a hybrid, is where most mature teams end up.

Bright Data and Oxylabs sell scraper APIs alongside their proxy networks, so a hybrid needn't mean a second vendor. Whatever you choose, budget in **cost per successful record**. Chapter 10 builds the full cost model and revisits this call with numbers.

KEY TAKEAWAYS

- Six layers, each with one job and one health metric, plus monitoring across all of them.
- Check what the first response contains before you reach for a browser.
- Climb the maturity ladder one rung at a time, when the current rung starts to hurt.
- Build what's core, buy what's hard, and compare options in cost per successful record.

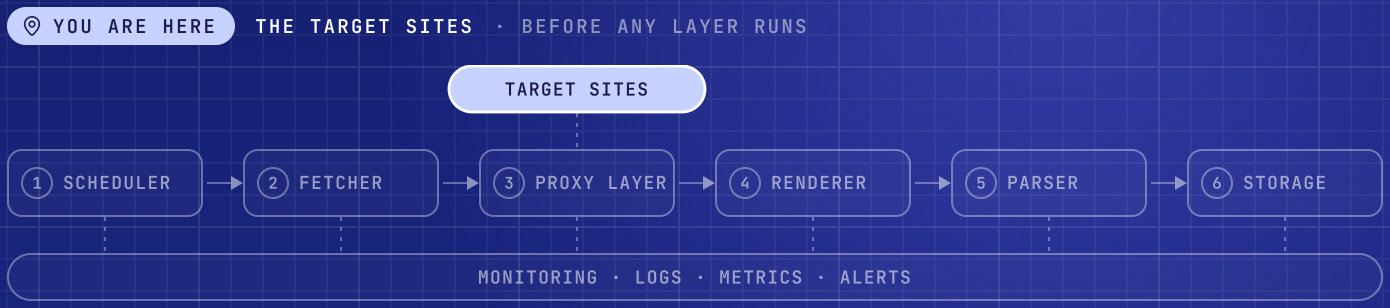
Go deeper: What Is Web Scraping? · Best Web Scraping APIs · Web Scraping at Large Scale

02



Recon: Read the Target Before You Write Code

An hour of reading saves a week of scraping. Find out what the site publishes, where its data really lives, how it paginates and how fast it lets you go. Half the time the answer is an API nobody told you about.



IN THIS CHAPTER

2.1 Start with what the site publishes
2.2 Find the data behind the page
2.3 Pagination and rate limits
2.4 The page-type inventory

2.1 Start With What the Site Publishes

Every site tells crawlers two things for free: what it would rather you skip, and where its URLs are. Both live at predictable addresses, and both are worth reading before anything else.

robots.txt: the site's stated wishes

The Robots Exclusion Protocol, standardized as RFC 9309 in 2022, is a request, not an access control. It still tells you a lot: which paths the owner considers off-limits, which crawlers they single out, and where the sitemaps are.

https://shop.example.com/robots.txt

```
User-agent: *
Disallow: /cart/
Disallow: /search          # internal search: endless URL space
Allow: /search/help
Crawl-delay: 5             # not in RFC 9309, but a clear hint

Sitemap: https://shop.example.com/sitemap_index.xml
```

Python · reading robots.txt

```python
from urllib.robotparser import RobotFileParser

rp = RobotFileParser("https://shop.example.com/robots.txt")
rp.read()
rp.can_fetch("AcmePriceBot/1.0", "https://shop.example.com/search?q=shoes")  # False
rp.crawl_delay("AcmePriceBot/1.0")                                           # 5
rp.site_maps()     # ['https://shop.example.com/sitemap_index.xml']
```

> **The standard library parser is simpler than the standard**
>
> Python's `urllib.robotparser` applies rules in file order and ignores wildcards, while RFC 9309 says the most specific match wins. For exact matching, use **Protego**, the parser Scrapy ships with.

Sitemaps: the cheapest complete URL list you'll ever get

A sitemap file holds up to 50,000 URLs and 50 MB uncompressed, so big sites publish a **sitemap index** that points to many of them. Each URL can carry a `<lastmod>` date, which turns a full re-crawl into an incremental one: fetch only what changed since your last run. Sitemaps are often more complete than anything you could discover by following links.

Robots.txt Tester
Check any URL against a site's rules.

Sitemap Visualizer
See a site's sections and sizes at a glance.

URL Extractor
Pull every URL a site exposes.

2.2 Find the Data Behind the Page

The value you see on screen usually exists in several places at once, and the rendered HTML is the worst of them: it's the part designers change most. Look for the same value higher up the list first.

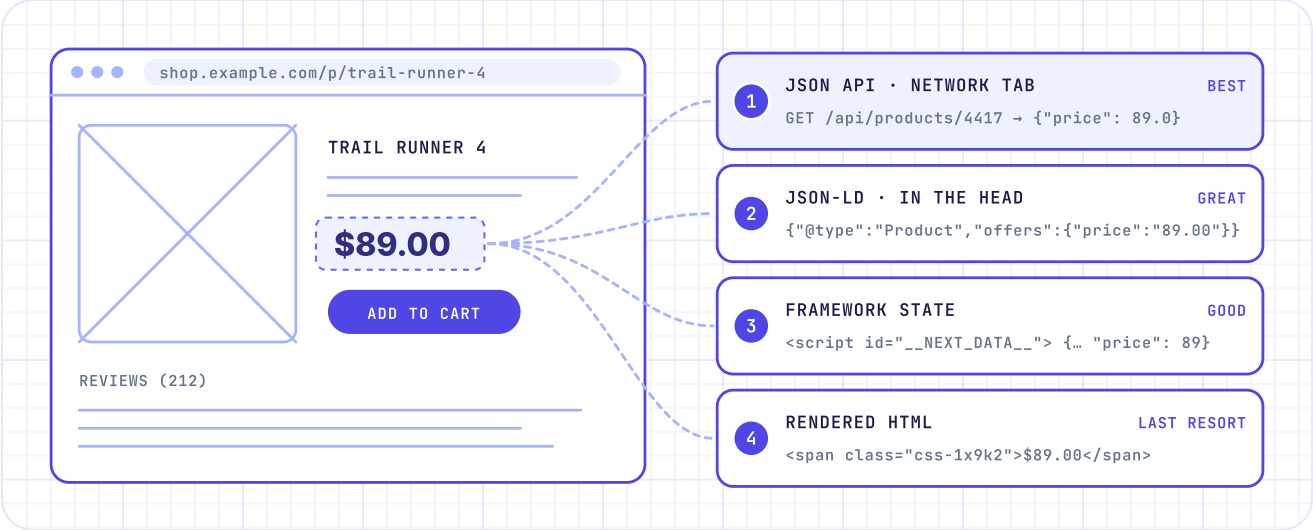


Figure 2.2 One price, four hiding places. Work down from the top: each step down is closer to what designers change, and more likely to break.

Hidden JSON APIs

Open DevTools, select the **Network** tab, filter to **Fetch/XHR** and reload. Click through pagination or a filter and watch which requests fire. When one returns JSON with your data, right-click it and choose **Copy as cURL**, then replay it in a terminal and strip headers until it breaks. What's left is the minimum request your fetcher has to send.

Terminal · replaying the request the page makes

```
curl 'https://shop.example.com/api/products?category=running&page=2&limit=48' \
  -H 'accept: application/json' \
  -H 'accept-language: en-US,en;q=0.9' \
  -H 'referer: https://shop.example.com/c/running'
```

Structured data and embedded JSON

Many pages carry **JSON-LD** for search engines: `Product`, `Offer`, `Article`, `JobPosting` and more, in a `<script type="application/ld+json">` block. Framework apps also embed their state: `__NEXT_DATA__` on Next.js sites and `__NUXT__` on Nuxt. Knowing the stack tells you where to look.

Free tools: Schema Extractor pulls JSON-LD from any page · Website Technology Checker names the framework · cURL Builder

2.3 Pagination and Rate Limits

Lists are where scrapers lose data quietly. Identify the pattern before you write the loop.

Pattern	Looks like	Crawl it by	Watch for
Page numbers	`?page=7`	Counting up until a page is empty	Pages past the end that return page 1
Offset and limit	`?offset=96&limit=48`	Raising `limit` as far as it allows	A hidden maximum offset
Cursor	`?cursor=eyJpZCI6…`	Following each next cursor	Expiry; no parallel pages
Infinite scroll	XHR fired on scroll	Calling that XHR directly	It's one of the three above
Faceted filters	`?color=red&size=9`	Crawling categories, not combinations	An endless URL space

Find the speed limit before it finds you

Run a short ramp test against one host: start at one request every two seconds, double the rate every few minutes, and record success and latency at each step. The ceiling is where success starts to fall or latency climbs. Then run well below it.

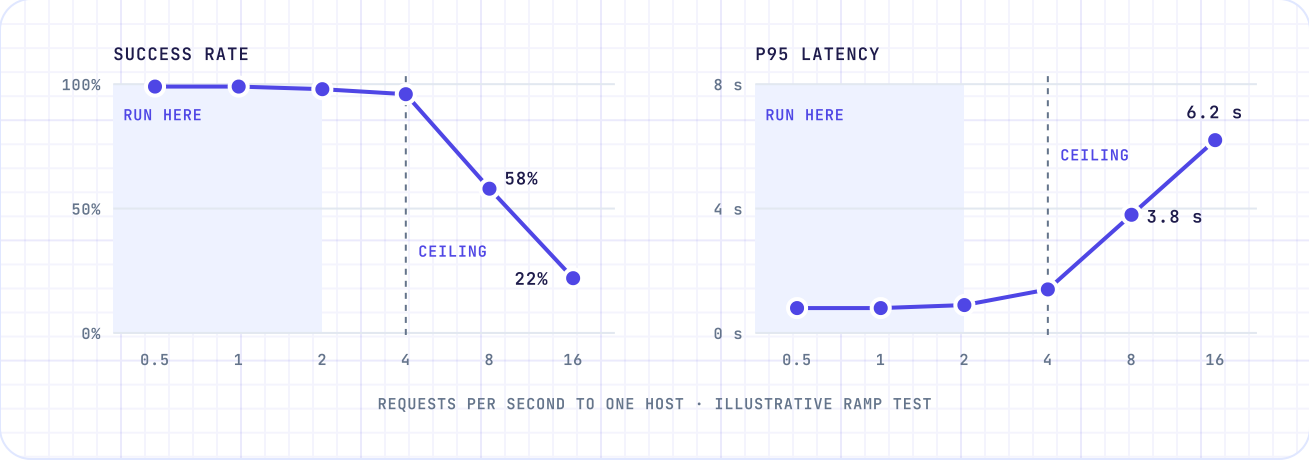


Figure 2.3 A ramp test as two charts on one x-axis. Here the ceiling sits near 4 requests a second, so production runs at 2. Measure your own.

While you ramp, read the response headers. A `429` with `Retry-After` tells you exactly how long to wait, and APIs often expose their budget in headers such as `X-RateLimit-Remaining`. Soft signals count too: a sudden CAPTCHA, a 200 that carries a challenge page, or latency that creeps up before anything fails.

> **Put the ceiling in config, per host**
> Chapter 6's per-host limits read from it. Re-run the ramp test when block rates change.

2.4 The Page-Type Inventory

Close your recon with one table: every kind of page you need, how many there are and where their data lives. Here's one for an imaginary running-shoe store.

Page type	URL pattern	Count	Data source	Browser?	Refresh
Category listing	`/c/{slug}?page={n}`	1,200	HTML + JSON-LD	No	Daily
Product detail	`/p/{slug}`	80,000	`__NEXT_DATA__`	No	Daily
Price and stock	`/api/products/{id}`	80,000	JSON API	No	Hourly
Reviews	`/api/reviews?cursor=…`	1.5M	JSON API	No	Weekly
Store stock	`/stores?zip=…`	400	XHR after a geolocation step	Yes	Daily
Site search	`/search?q=…`	Unbounded	Disallowed in robots.txt	Skip	Never

Worked example. Counts come from the sitemap index; the browser column is the 30-second test from section 1.2.

Notice what the inventory just bought you. Five of six page types need nothing heavier than an HTTP client, the most volatile field (price) comes from a small JSON endpoint you can poll hourly, and the one page type that needs a browser has only 400 URLs.

The recon checklist

- [ ] robots.txt read, saved and dated
- [ ] Framework and CDN identified
- [ ] Pagination pattern written down
- [ ] Page types inventoried with counts
- [ ] Terms skimmed for scraping clauses
- [ ] Sitemaps found and URLs counted
- [ ] JSON APIs and embedded data checked
- [ ] Rate ceiling measured per host
- [ ] Login walls and paywalls noted, and avoided
- [ ] Sample pages saved as test fixtures

KEY TAKEAWAYS

- Read robots.txt and the sitemaps first; they're free and they shape the whole crawl.
- Prefer data sources in order: JSON API, JSON-LD, embedded state, then rendered HTML.
- Identify the pagination pattern and measure the rate ceiling before writing the loop.
- An inventory of page types tells you what to build, what to render and what to skip.

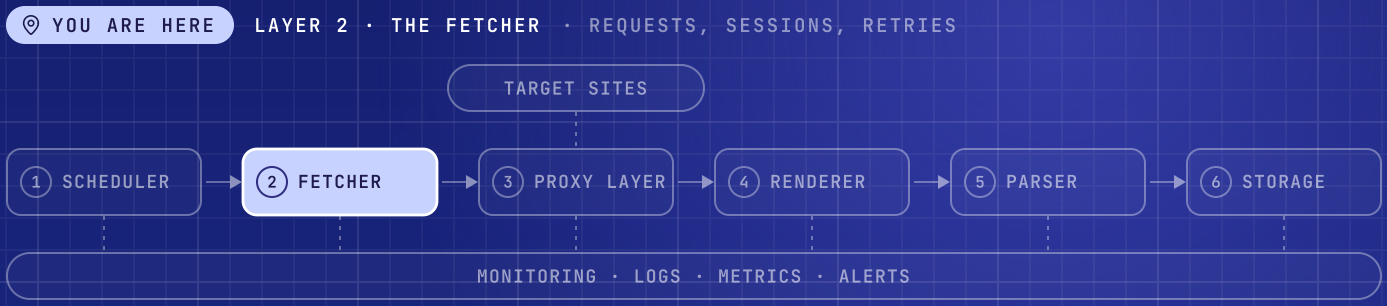
Go deeper: Web Scraping Legality and robots.txt · Scraping JavaScript Websites · Website Link Extractor

03



Fetching Pages Reliably

Most scraping bugs are fetching bugs in disguise: a timeout nobody set, a retry loop that hammers a struggling server, a header that gives the game away. Get this layer right and everything downstream gets calmer.



IN THIS CHAPTER

3.1 Pick an HTTP client
3.2 Sessions, cookies and headers
3.3 Timeouts, retries and backoff
3.4 Read the status code
3.5 A production-ready client

3.1 Pick an HTTP Client

Recon told you most of your pages need no browser. That makes the HTTP client the workhorse of the system, and the choice matters more than it looks: it decides whether you can run requests concurrently, speak HTTP/2 and, on strict targets, whether your TLS handshake looks like a browser's or like a script's.

Client	Language	Async	HTTP/2	Browser-like TLS	Reach for it when
requests	Python	No	No	No	A quick script or a one-off job
httpx	Python	Yes	Opt-in	No	**The default for new scrapers**
aiohttp	Python	Yes	No	No	Very high concurrency on HTTP/1.1
curl_cffi	Python	Yes	Yes	Yes, impersonates	A target fingerprints your TLS
Scrapy	Python	Yes	Experimental	No	You want a full crawl framework
fetch / undici	Node.js	Yes	Opt-in	No	Your stack is already Node

Our default: httpx, with an escape hatch

One API for sync and async code, HTTP/2 when you ask for it, a single `proxy=` setting and sane timeouts. Keep the fetcher behind one function, and you can move a stubborn target to curl_cffi (Chapter 8) without touching anything else.

Framework or library?

Scrapy bundles a scheduler, per-domain concurrency, retries, throttling and item pipelines, so it covers layers 1, 2 and 5 out of the box. Libraries give you more control and fewer conventions to learn. Both are fine choices; the patterns in this book map onto either. If you go with Scrapy, its `AUTOTHROTTLE` and `CONCURRENT_REQUESTS_PER_DOMAIN` settings are the first two to tune, and its defaults allow 16 concurrent requests overall and 8 per domain.

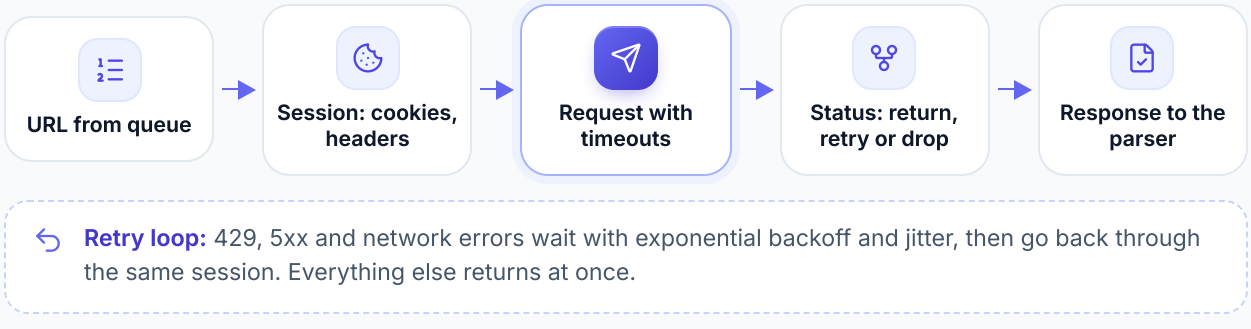


Figure 3.1 The fetcher's whole job, in five steps. Sections 3.2 to 3.4 take them one at a time.

3.2 Sessions, Cookies and Headers That Matter

A real browser sends a consistent bundle of headers, in a consistent order, and carries cookies from page to page. Anti-bot systems check every piece against what the claimed browser would send.

```
GET /p/trail-runner-4 HTTP/2                                              6
sec-ch-ua: "Google Chrome";v="NNN", "Chromium";v="NNN"                     1
sec-ch-ua-platform: "Windows"
user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) … Chrome/NNN.0.0.0   1
accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8   2
sec-fetch-site: same-origin
sec-fetch-mode: navigate
referer: https://shop.example.com/c/running                               4
accept-encoding: gzip, deflate, br, zstd
accept-language: en-US,en;q=0.9                                           3
cookie: session=8f2c…; consent=yes                                        5
```

1. **User-Agent and client hints agree.** Same browser, version and platform in both.
2. **Accept fits the resource.** HTML for pages, `application/json` for API calls.
3. **Language fits the exit.** A German IP asking for `en-US` only is a small flag.
4. **Referer tells a story.** Product pages are usually reached from a listing.
5. **Cookies carry the session.** Keep them per identity; never share them across IPs.
6. **Protocol and order count.** "Chrome" over HTTP/1.1, or in odd header order, stands out.

Figure 3.2 An abridged Chrome navigation. `NNN` is the current version; copy real values from your own browser.

Sessions and TLS, briefly

Use a client object, such as `httpx.AsyncClient`, so cookies and connections persist across requests, and treat each one as a single identity: one cookie jar, one User-Agent, one proxy session. Underneath HTTP, the TLS handshake has a fingerprint too, hashed as JA3 or JA4, and Python's TLS stack doesn't look like Chrome's. Most sites don't check it. Strict ones do, and Chapter 8 covers the fix.

Random User-Agent
Current, realistic browser strings.

User-Agent Parser
Check what a string claims to be.

cURL Builder
Build and test requests visually.

Common header mistake	What it tells the site
Leaving the library default, such as `python-httpx/0.28`	A script, announced in plain text
A Chrome User-Agent with no `sec-ch-ua` **hints**	Something that only claims to be Chrome
`Accept: */*` **on a page navigation**	Not a browser loading a page
One cookie jar shared across many exit IPs	A single user teleporting between cities

3.3 Timeouts, Retries and Backoff

Networks fail, proxies drop connections and servers have bad minutes. A fetcher that handles this calmly needs three things: timeouts on every request, a short list of failures worth retrying, and waits that grow and spread out.

Two timeouts

A short **connect** timeout (5 to 10 s) kills dead exits fast; a longer **read** timeout fits the page. httpx's 5-second default is tight for proxies.

Retry only the retryable

Retry network errors, `429` and `5xx`. Never retry a `404`, and don't hammer a `403`: the same request will fail the same way. Cap attempts at four or five.

Back off, with jitter

Double the wait after each failure, cap it, and pick a random time inside that window. Honor `Retry-After` whenever the server sends it.

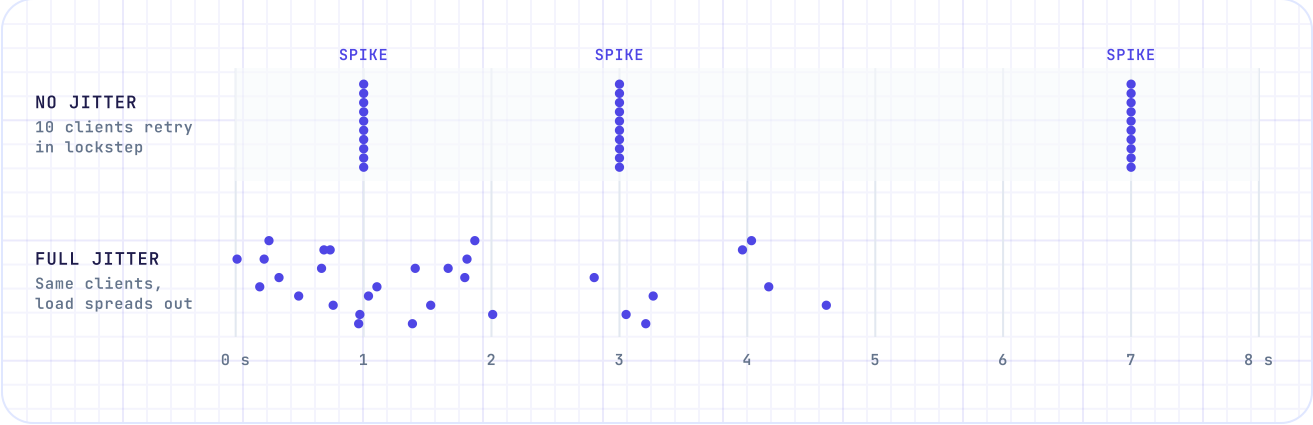


Figure 3.3 Why jitter matters. Ten clients fail at once. With plain exponential backoff they retry together at 1, 3 and 7 seconds and hit the server in waves. With full jitter the same retries land at random points inside each window.

Retry budgets and circuit breakers

Per-request retries aren't enough when a whole host goes bad. Give each host a **retry budget** (say, retries may not exceed 10% of its requests in any five minutes) and a **circuit breaker** that pauses it when errors spike. A struggling server then becomes a pause in your schedule, not a retry storm.

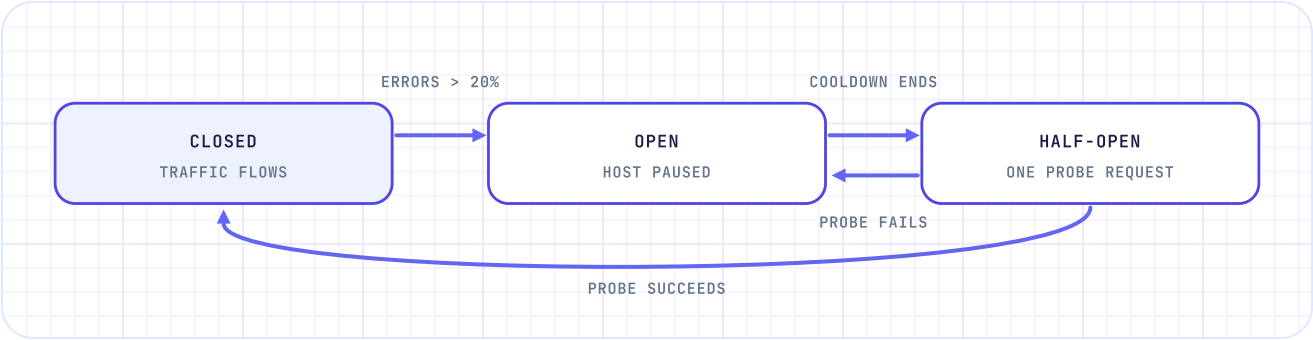


Figure 3.4 A per-host circuit breaker. The 20% threshold is an example; tune it to each host's normal error rate.

3.4 Read the Status Code

The status code is the server's one-line verdict on your request. Map every code your targets return to a single action, write that mapping down in code, and your fetcher will behave the same way at 3 a.m. as it does when you're watching.

Status	Usually means	Your fetcher should
`200` **OK**	It worked, probably	Pass it to the parser, which decides if it's real data
`301` `308` **Moved**	The URL moved for good	Follow it, and update the URL in your frontier
`302` `307` **Found**	Temporary move, or a redirect to a login or challenge	Follow it, keep the original URL, check where you landed
`304` **Not Modified**	Unchanged since your copy	Reuse the stored copy; the request cost almost nothing
`403` **Forbidden**	Blocked, or not allowed at all	Don't retry as-is; check identity, then escalate (Chapter 8)
`404` `410` **Gone**	The page doesn't exist	Mark it dead; 410 means stop scheduling it for good
`429` **Too Many**	You hit a rate limit	Honor `Retry-After` and slow this host down
`500` `502` `503` `504`	Server, gateway or proxy trouble	Retry with backoff; 503 often carries `Retry-After`
`ERR` **Timeout, reset**	No answer at all	Retry with backoff; if one exit keeps failing, rotate it

> ⚠ **A 200 isn't a success until the parser says so**
>
> Blocked requests often come back as `200 OK` with a CAPTCHA, a consent wall or an empty product grid. Count success only after validation (Chapter 5); otherwise your dashboard stays green while your data goes missing.

Conditional requests: the cheapest page is the one you don't download

If a response carries an `ETag` or `Last-Modified` header, store it with the page. On the next visit, send it back as `If-None-Match` or `If-Modified-Since`. When nothing changed, the server answers `304` with an empty body, and on per-GB proxy plans that saving goes straight to your bill. Not every site supports it, but finding out costs one request.

3.5 A Production-Ready Client

Everything from this chapter in one place. The caller decides what a 404 means.

Python · fetcher.py (httpx)

```python
import asyncio, random
import httpx
UA = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) ..."   # a real, current browser UA
RETRYABLE = {429, 500, 502, 503, 504}
def backoff(attempt, base=1.0, cap=30.0):
    return random.uniform(0, min(cap, base * 2 ** attempt))   # full jitter

async def fetch(client, url, attempts=5):
    for attempt in range(attempts):
        try:
            r = await client.get(url)
        except httpx.TransportError:                # timeouts, resets, proxy errors
            r = None
        if r is not None and r.status_code not in RETRYABLE:
            return r                                # 2xx, 3xx, 404: the caller decides
        if attempt < attempts - 1:
            wait = r.headers.get("Retry-After", "") if r is not None else ""
            await asyncio.sleep(min(int(wait), 120) if wait.isdigit() else backoff(attempt))
    raise RuntimeError(f"gave up on {url}")
async def main():
    async with httpx.AsyncClient(
        http2=True,                                 # pip install "httpx[http2]"
        proxy="http://USER:PASS@proxy.example.com:8000",
        headers={"User-Agent": UA, "Accept-Language": "en-US,en;q=0.9"},
        timeout=httpx.Timeout(30.0, connect=10.0),
        limits=httpx.Limits(max_connections=20),
        follow_redirects=True,
    ) as client:
        r = await fetch(client, "https://shop.example.com/c/running?page=1")
        print(r.status_code, r.http_version, len(r.content))

asyncio.run(main())
```

KEY TAKEAWAYS

- Default to httpx, behind one function, and send a coherent header bundle per identity.
- Set both timeouts, retry only 429, 5xx and network errors, and back off with full jitter.
- Map every status code to one action; use 304s to skip unchanged pages.

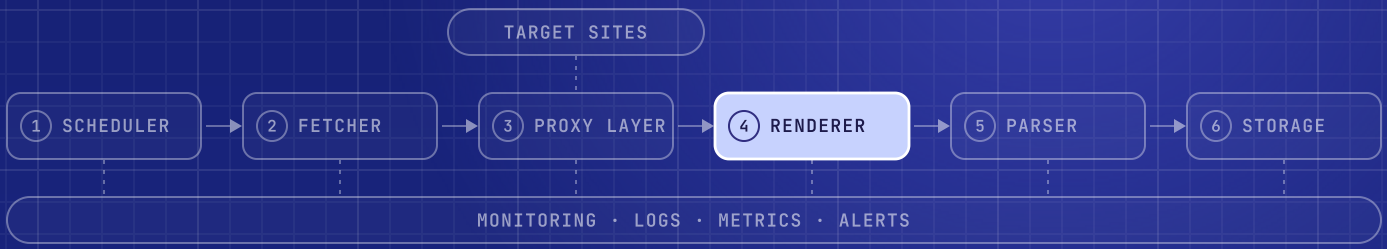
Go deeper: Web Scraping With Python · HTTP Headers for Scraping · Handling Timeouts and Errors

04

Rendering JavaScript Without Burning Money

A headless browser can scrape anything a person can see, which is exactly why teams reach for it too early. It's also the most expensive way to fetch a page. Use one only where it pays, and make every page it loads as light as possible.

YOU ARE HERE LAYER 4 · THE RENDERER · BROWSERS, ONLY WHERE THEY PAY



IN THIS CHAPTER

4.1 Do you need a browser?
4.2 Playwright, Puppeteer or Selenium
4.3 Block, capture and isolate

4.1 Do You Need a Browser?

Ask four questions in order and stop at the first yes. Most targets never get past the third.

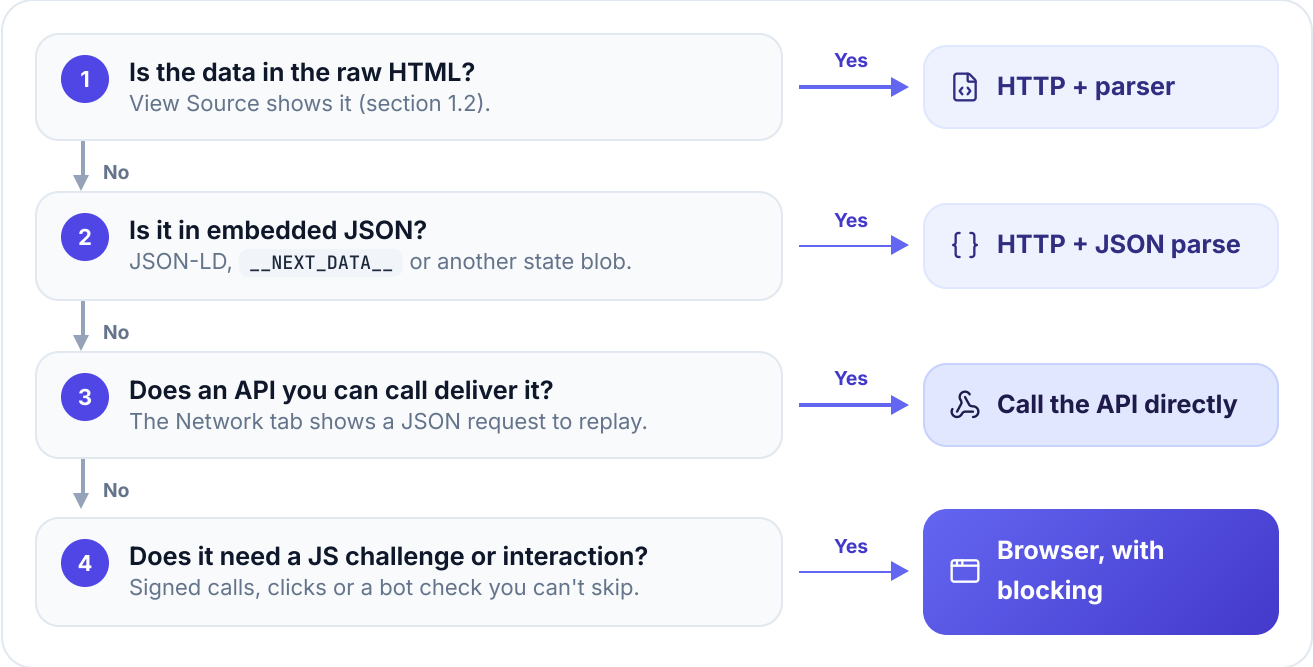


Figure 4.1 Even a "browser" answer often means one browser visit per session for cookies or tokens, then plain HTTP.

What one page costs to fetch

Say a product page weighs 2.5 MB with every asset and your proxies cost $5 per GB. Here's the bandwidth bill for 1,000 pages.

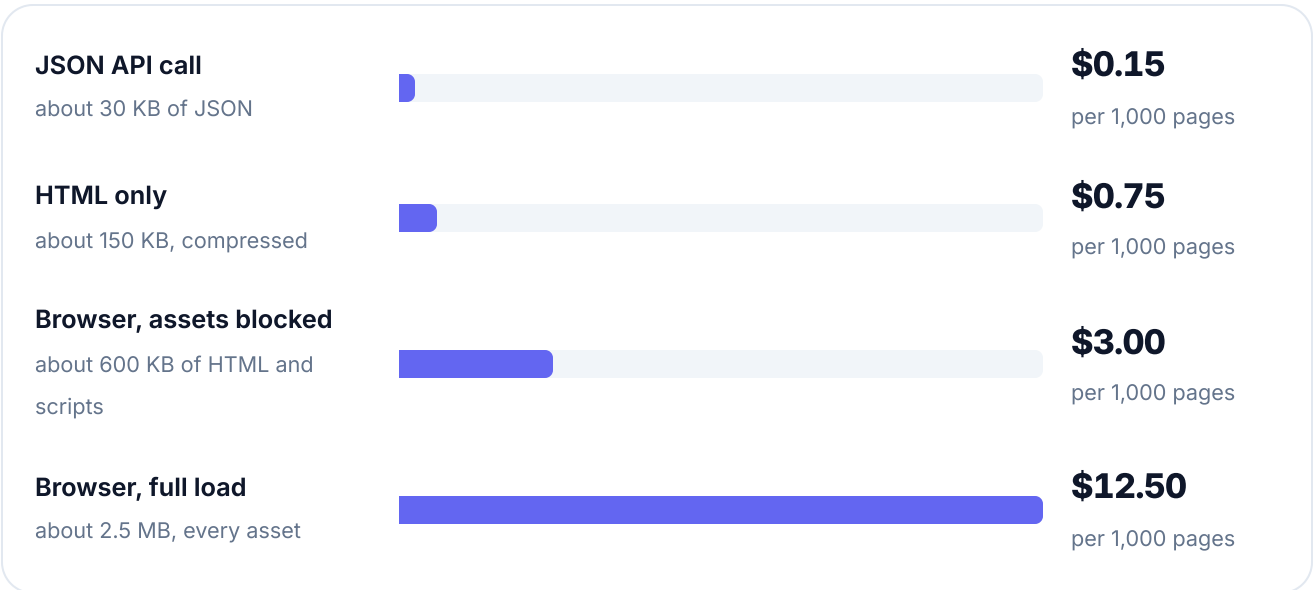


Figure 4.2 Same data, an 80-fold spread. Sizes are illustrative; browsers also cost CPU, memory and time.

4.2 Playwright, Puppeteer or Selenium

All three drive real browsers, and all three can scrape. They differ in who maintains them, which languages and browsers they support, and how much of the waiting and network control you get for free.

	Playwright	Puppeteer	Selenium
Maintained by	Microsoft	Google's Chrome team	The Selenium project
Languages	JavaScript, Python, Java, .NET	JavaScript and TypeScript	Java, Python, C#, Ruby, JavaScript
Browsers	Chromium, Firefox, WebKit	Chrome, Firefox	Chrome, Firefox, Edge, Safari
Waiting	Auto-waits on every action	Locators that wait	Explicit waits you write
Network control	Route, block and read responses	Request interception	More limited
Isolation	Cheap contexts, each with its own proxy	Browser contexts	One session per driver
Pick it when	You're starting fresh	Your team is Node and Chrome-only	You already run a Selenium grid

Our pick for new scrapers: Playwright. Cheap contexts and response capture do a lot of work for you.

Wait for the thing, not for the clock

A fixed `sleep(5)` is too long, which burns money, or too short, which loses data.

Strategy	Playwright (Python)	Use it when
Early navigation	`goto(url, wait_until="domcontentloaded")`	Always, as the starting point
Wait for an element	`locator("[data-testid=price]").wait_for()`	The data renders into the DOM
Wait for a response	`expect_response(lambda r: "/api/" in r.url)`	The data arrives as JSON
Network idle	`wait_until="networkidle"`	Rarely: Playwright's own docs discourage it

> ⚠ **Network idle may never come**
>
> It waits for 500 ms with no network traffic. Analytics beacons, chat widgets and long polling keep pages busy for far longer, so scrapers stall or time out for no good reason.

> 👁 **Headless isn't a disguise.** Automation leaves traces such as `navigator.webdriver`. Chapter 8 covers what strict sites check.

4.3 Block, Capture and Isolate

Three habits make browsers affordable: **block** heavy assets, **capture** the JSON the page fetches instead of parsing the DOM, and **isolate** each session in a disposable context.

Python · Playwright with a proxy, blocking and response capture

```python
import asyncio
from playwright.async_api import async_playwright
PROXY = {"server": "http://proxy.example.com:8000", "username": "USER", "password": "PASS"}
BLOCK = {"image", "media", "font"}           # add "stylesheet" if the page still works

async def block_heavy(route):
    if route.request.resource_type in BLOCK:
        await route.abort()
    else:
        await route.continue_()
async def scrape(browser, url):
    context = await browser.new_context(proxy=PROXY, locale="en-US")   # one identity
    await context.route("**/*", block_heavy)
    page = await context.new_page()
    try:
        async with page.expect_response(lambda r: "/api/products" in r.url and r.ok) as info:
            await page.goto(url, wait_until="domcontentloaded")
        return await (await info.value).json()     # the data itself, no DOM parsing
    finally:
        await context.close()               # cookies, storage and proxy go with it
async def main():
    async with async_playwright() as p:
        browser = await p.chromium.launch()
        data = await scrape(browser, "https://shop.example.com/c/running")
        print(len(data["items"]), "products")

asyncio.run(main())
```

Run them like cattle: about one browser per CPU core, a restart every few hundred pages, and ad and analytics domains aborted.

KEY TAKEAWAYS

- Ask four questions before launching a browser; most targets stop at the API.
- Pick Playwright for new work; wait for elements or responses, never for the clock.
- Block heavy assets, capture JSON responses and use one disposable context per identity.

Go deeper: Playwright Scraping Guide · Playwright vs Puppeteer · Playwright vs Selenium

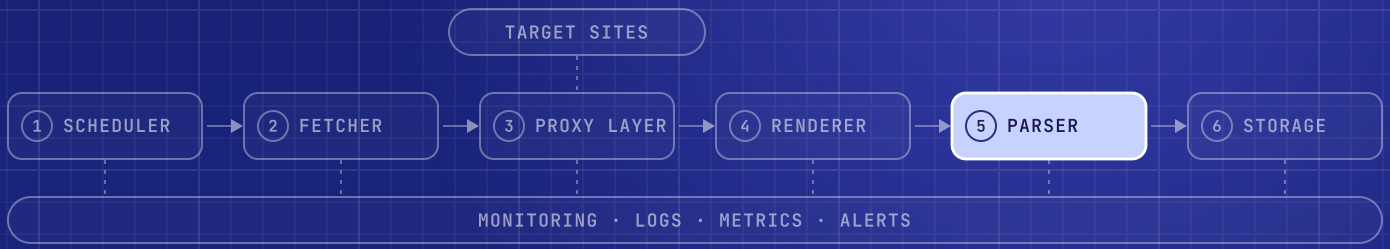
05



Parsing and Extraction That Survives Redesigns

Fetching gets you bytes. Parsing turns them into records you can trust, and it's the layer that breaks most often, usually in silence. Pick selectors that outlive redesigns, type every field and measure how full your data really is.

YOU ARE HERE LAYER 5 · PARSER AND VALIDATOR · FROM BYTES TO TYPED RECORDS



IN THIS CHAPTER

5.1 The selector hierarchy
5.2 CSS, XPath and resilient selectors
5.3 Typed extraction with validation
5.4 LLM extraction and fill rate

5.1 The Selector Hierarchy

Every field can usually be read from several places, and they don't age equally. Data published for machines changes rarely; markup styled for humans changes every time the design team ships. Read each field from the highest level that has it.

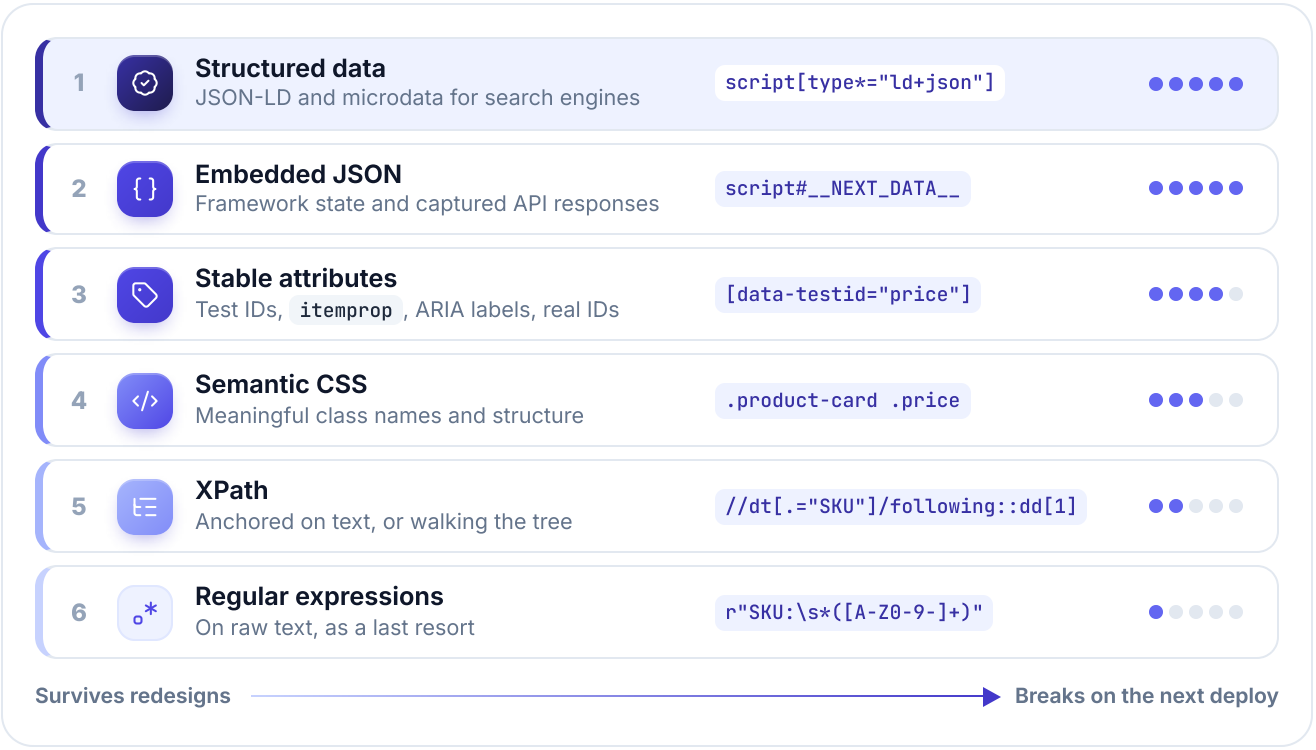


Figure 5.1 Six places to read a field from, most durable first. The dots are our editorial rating of how well each survives a site redesign.

Brittle and resilient, side by side

Brittle	Resilient	Why the second lasts
`div > div:nth-child(3) > span`	`[itemprop="price"]`	Meaning, not position
`.sc-AxjAm.css-1x9k2`	`[data-testid="title"]`	Generated classes change
`/html/body/div[2]/main/div[3]`	`//dt[.="SKU"]/following-sibling::dd[1]`	Anchored to a label

Two selectors per field, and log which one matched

Pair a primary from the highest level available with a fallback from a different level. When the primary stops matching but the fallback still works, a redesign just happened, and you heard about it before your data degraded.

5.2 CSS, XPath and Resilient Selectors

CSS selectors are shorter and easier to read; XPath can do things CSS can't, such as matching on text or stepping back up the tree. Use CSS by default and switch to XPath when the field is only reachable through a label or a relationship. The full side-by-side cheat sheet is in the appendix.

You need to	CSS	XPath
Match an attribute	`[data-testid="price"]`	`//*[@data-testid="price"]`
Match on visible text	*Not possible*	`//button[contains(., "Add to cart")]`
Take the value after a label	`dt + dd`, if it's adjacent	`//dt[.="SKU"]/following-sibling::dd[1]`
Step up to a parent	`:has()`, where supported	`//span[@class="price"]/..`
Pick the nth match	`li:nth-of-type(2)`	`(//li)[2]`
Read an attribute value	Use your library's attribute call	`//a/@href`

Rules that keep selectors alive

Skip generated class names

Strings like `css-1x9k2` or `sc-AxjAm` come from CSS-in-JS tools and change on every build. If a class looks like a hash, don't use it.

Avoid positions

`nth-child` and long absolute paths break the moment a banner or a new column appears. Anchor on meaning instead.

Anchor on labels

The text "SKU" or "Price" next to a value is the most stable thing on many pages. XPath's `following-sibling` was made for this.

Keep it short and unique

The shortest selector that matches exactly one element has the fewest assumptions to break. Test it against five saved pages, not one.

Parse with lxml, query with parsel or BeautifulSoup

lxml is the fast C parser underneath most Python tools. Parsel, which Scrapy uses, gives you CSS and XPath on the same object. BeautifulSoup is friendlier with broken HTML and supports CSS through its `select()` method.

Go deeper: CSS Selectors vs XPath · BeautifulSoup Tutorial

5.3 Typed Extraction With Validation

Loose dictionaries push parser bugs downstream. Make the parser build a typed model, so a missing price fails loudly at the page that caused it.

Python · lxml + pydantic v2

```
import json
from decimal import Decimal
from lxml import html
from pydantic import BaseModel, HttpUrl, field_validator

class Product(BaseModel):
    url: HttpUrl
    name: str
    price: Decimal
    currency: str
    in_stock: bool
    @field_validator("price")
    @classmethod
    def positive(cls, v):
        if v <= 0: raise ValueError("price must be positive")
        return v
def extract(url: str, body: str) -> Product:
    tree = html.fromstring(body)
    for block in tree.xpath('//script[@type="application/ld+json"]/text()'):
        data = json.loads(block)                    # 1. structured data first
        if isinstance(data, dict) and data.get("@type") == "Product":
            offer = data["offers"] if isinstance(data["offers"], dict) else data["offers"][0]
            return Product(url=url, name=data["name"], price=offer["price"],
                           currency=offer["priceCurrency"],
                           in_stock=offer.get("availability", "").endswith("InStock"))
    return Product(                                 # 2. fall back to stable attributes
        url=url,
        name=tree.xpath("string(//h1)").strip(),
        price=tree.xpath('string(//*[@itemprop="price"]/@content)'),
        currency=tree.xpath('string(//*[@itemprop="priceCurrency"]/@content)'),
        in_stock=bool(tree.xpath('//*[@itemprop="availability"][contains(@href, "InStock")]')),
    )
```

An empty price can't become a `Decimal`, so a page that lost its price raises a `ValidationError` naming the field instead of writing a blank row. Production code also walks JSON-LD lists and `@graph` blocks.

Quarantine, don't drop

Catch `ValidationError`, store the raw page with the error and the URL in a quarantine table, and keep going. Ten quarantined pages from one template is a redesign alert; one is noise.

5.4 LLM Extraction and Fill Rate

Large language models can read a page the way a person does, which makes them tempting for extraction. They earn their cost in some places and waste it in others.

Worth it when

- Hundreds of small sites share no template
- Values hide in prose, like specs in a description
- It's a one-off job and selectors cost more

Not worth it when

- Pages share a template: selectors are cheaper
- Volume is high: per-page costs add up fast
- You can't afford an invented number

Validate model output like any other input

- **Same schema, same validator.** Ask for JSON that fits your pydantic model; reject the rest.
- **Ground every value.** Check that each extracted value appears in the page text.
- **Audit a sample.** Check a few dozen records a week by hand and track the error rate.

Fill rate: the metric that catches silent breakage

Fill rate is the share of records where a field is present and valid. A redesign rarely kills a whole scraper; it quietly empties one field, and per-field fill rate shows it first.

Field	Fill rate		7-day average	Status
name	100%		100%	OK
price	99.6%		99.7%	OK
rating	71.0%		70.6%	OK
brand	38.4%		92.3%	Dropped

Figure 5.2 A worked example. Rating is low but stable (many products lack reviews); brand fell overnight: a selector broke.

KEY TAKEAWAYS

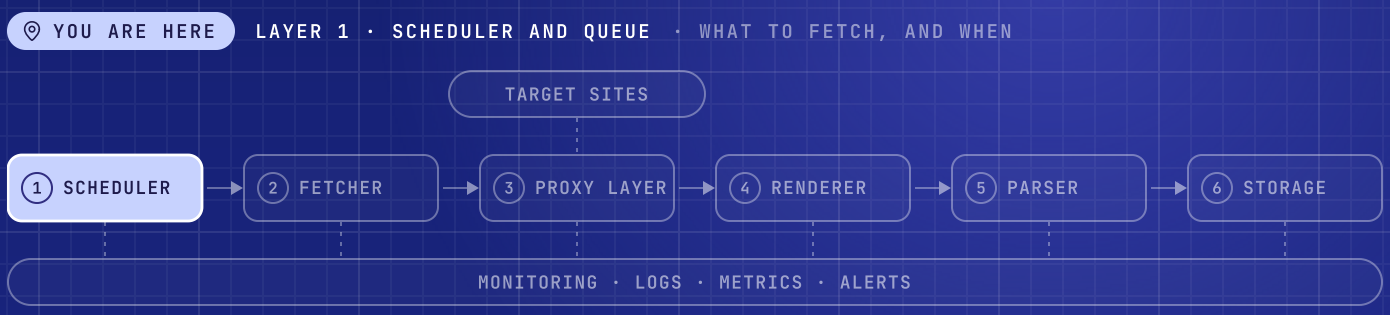
- Read fields from structured data first and regex last; keep a fallback selector per field.
- Build typed records and quarantine pages that fail validation instead of writing blanks.
- Use LLMs for varied layouts, validate and ground their output, and alert on fill-rate drops.

Go deeper: Cleaning and Storing Scraped Data · Using ChatGPT for Web Scraping · Data Extraction

06

Crawling at Scale

Fetching one page is a request. Fetching ten million is a scheduling problem: which URL comes next, which ones you've already seen, how hard each host can be pushed and how many workers it all takes. Frontiers, queues and deduplication answer all four.



IN THIS CHAPTER

6.1 The URL frontier
6.2 Canonical URLs and deduplication
6.3 Politeness and per-host limits
6.4 Workers, queues and throughput math

6.1 The URL Frontier

The frontier is the crawler's to-do list, and it does two jobs at once. It decides **what matters most**, so new products are fetched before archive pages, and it decides **when each host may be hit again**, so no site gets more traffic than it can take. Good crawlers keep those two jobs in separate queues.

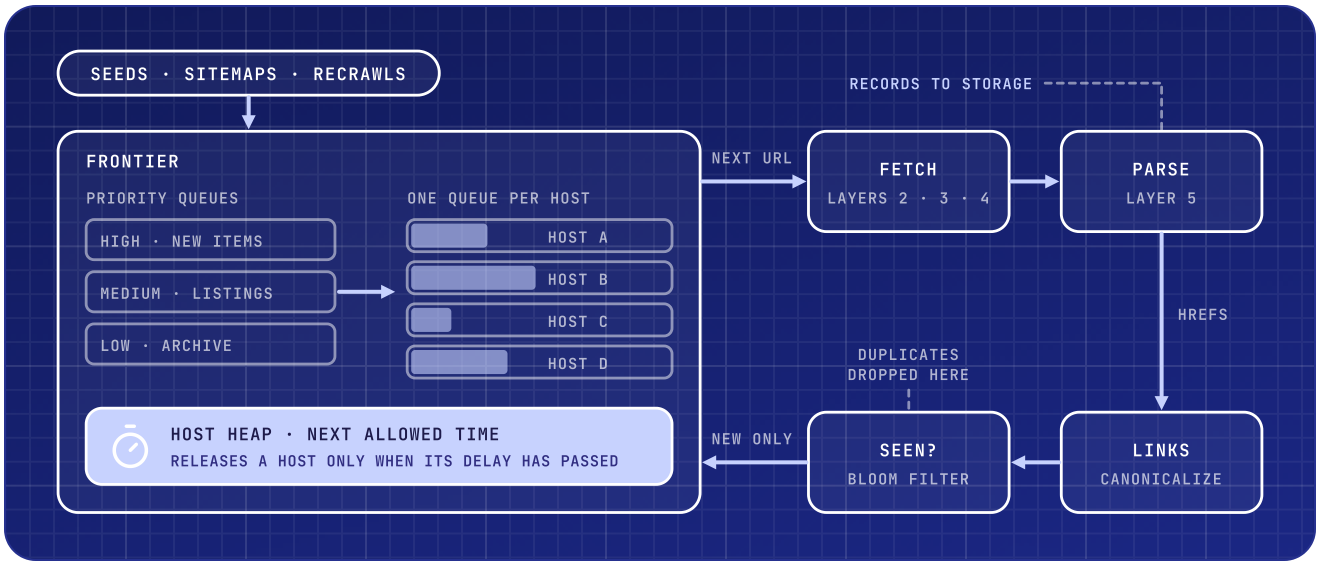


Figure 6.1 A two-tier frontier. Priority queues decide what matters; per-host queues and a heap of next-allowed times decide when each host may be visited. Only URLs the seen filter hasn't met go back in.

Setting priorities

Freshness

Pages whose `lastmod` moved, or that change often, go first. Stable pages can wait for their slot.

Value

Product pages that feed revenue decisions outrank tag pages and archives. Rank page types from your inventory.

Depth

Cap link depth. Pages ten clicks from the home page are usually calendars, filters and other crawl traps.

Crawl traps to cap before they eat your budget

Trap	Looks like	Defense
Calendars	`/events?month=2031-07`	Cap date ranges to the window you need
Faceted filters	`?color=red&size=9&sort=price`	Crawl categories; ignore filter parameters
Session IDs in URLs	`/p/123;jsessionid=8F2C`	Strip them in `canonicalize()`
Endless pagination	`?page=99999` returns page 1	Stop when a page repeats or comes back empty

6.2 Canonical URLs and Deduplication

The same page hides behind many URLs: different letter case, a default port, tracking parameters, reordered query strings, a fragment. Crawl them all and you pay for the same page five times. Canonicalize every URL before you check whether you've seen it.

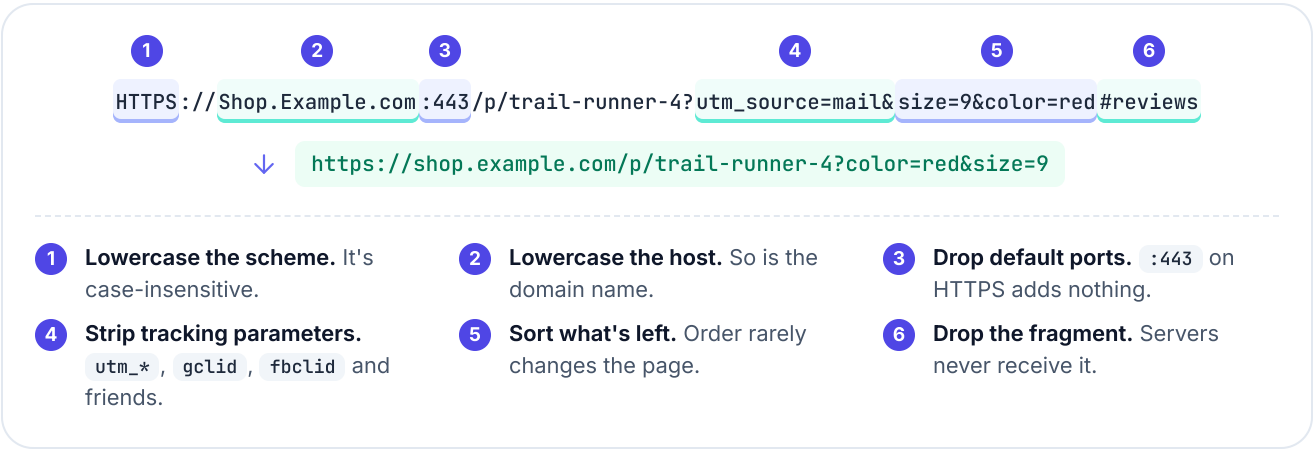


Figure 6.2 Six rules that turn many URLs into one. Add site-specific rules from recon too: trailing slashes, session IDs in paths, and the page's own `rel="canonical"`.

Python · canonicalize()

```python
from urllib.parse import urlsplit, urlunsplit, parse_qsl, urlencode

TRACKING = {"gclid", "fbclid", "msclkid", "ref"}        # plus every utm_* key
DEFAULT_PORT = {"http": 80, "https": 443}

def canonicalize(url: str) -> str:
    s = urlsplit(url)
    scheme, host = s.scheme.lower(), (s.hostname or "").lower()
    if s.port and s.port != DEFAULT_PORT.get(scheme):
        host = f"{host}:{s.port}"
    query = sorted((k, v) for k, v in parse_qsl(s.query, keep_blank_values=True)
                   if not k.lower().startswith("utm_") and k.lower() not in TRACKING)
    return urlunsplit((scheme, host, s.path or "/", urlencode(query), ""))
```

Three layers of dedup

URLs

A set of canonical URLs, or a **Bloom filter** at scale: about 9.6 bits per URL at a 1% false-positive rate, so roughly 120 MB for 100 million.

Content

Hash the normalized text of each page. Identical hashes mean identical pages; SimHash catches the near-duplicates.

Records

Key records on business identity, such as SKU plus store, so two URLs for one product become one row.

6.3 Politeness and Per-Host Limits

A global limit protects your machine; a **per-host** limit protects the site. No host should see more parallel requests than its measured ceiling (section 2.3).

Python · an asyncio crawler with a per-host limit

```python
import asyncio
from collections import defaultdict
from urllib.parse import urljoin, urlsplit
import httpx
from lxml import html
PER_HOST, DELAY, WORKERS = 4, 0.5, 32      # per host: 4 in flight, 0.5 s apart

class Crawler:
    def __init__(self, seeds):
        self.queue, self.seen = asyncio.Queue(), set()
        self.slots = defaultdict(lambda: asyncio.Semaphore(PER_HOST))
        self.allowed = {urlsplit(u).hostname for u in seeds}
        for url in seeds:
            self.enqueue(url)
    def enqueue(self, url):
        url = canonicalize(url)                              # section 6.2
        if urlsplit(url).hostname in self.allowed and url not in self.seen:
            self.seen.add(url)
            self.queue.put_nowait(url)
    async def worker(self, client):
        while True:
            url = await self.queue.get()
            try:
                async with self.slots[urlsplit(url).hostname]:   # per-host limit
                    r = await client.get(url)
                    await asyncio.sleep(DELAY)                   # spacing per slot
                if r.status_code == 200 and "html" in r.headers.get("content-type", ""):
                    handle_page(url, r)                          # hand off to layer 5
                    for href in html.fromstring(r.text).xpath("//a/@href"):
                        self.enqueue(urljoin(url, href))
            except Exception as exc:
                log_failure(url, exc)                            # retry per section 3.3
            finally:
                self.queue.task_done()
    async def run(self):
        async with httpx.AsyncClient(timeout=20, follow_redirects=True) as client:
            tasks = [asyncio.create_task(self.worker(client)) for _ in range(WORKERS)]
            await self.queue.join()
            for t in tasks:
                t.cancel()

asyncio.run(Crawler(["https://shop.example.com/"]).run())
```

> ⚠ **One shared queue blocks at the head**
>
> When most queued URLs belong to one host, workers pile up on its semaphore. At scale, switch to Figure 6.1's per-host queues.

6.4 Workers, Queues and Throughput Math

Once one machine isn't enough, the queue moves out of process, into Redis, RabbitMQ, Amazon SQS or Kafka, and workers become interchangeable. How easy that is depends on one question: does a worker need to remember anything between jobs?

	Stateless workers	Stateful workers
Remembers	Nothing between jobs	Cookies, sessions, browser contexts
Scales by	Adding workers anywhere	Routing each session to its worker
When one dies	Any worker retries the job	The session dies with it
Use for	Most HTTP scraping	Multi-step flows, long browser sessions

Make workers stateless where you can; keep cookies and session IDs in a shared store.

Worked example: three million pages a day

Say you refresh 3 million pages every 24 hours, at 1.5 seconds per response and a 10% retry rate. Little's law (in flight = rate × time per request) sizes the job.

34.7/s
pages per second
`3,000,000 ÷ 86,400`

38.2/s
requests, with retries
`34.7 × 1.1`

≈57
requests in flight
`38.2 × 1.5 s`

230K
pages a day per host
`4 in flight ÷ 1.5 s`

Fifty-seven concurrent requests is easy for one machine. The catch is the last tile: four in flight yields about 2.7 pages a second per host. If all 3 million pages live on one site, politeness, not hardware, sets the ceiling: 13 days per full pass.

Fetch less before you fetch faster

Refresh only what changed, using sitemap `lastmod` dates and 304s. If 10% of pages change daily, a 13-day pass becomes a daily job.

KEY TAKEAWAYS

- Separate priority queues from per-host politeness queues.
- Canonicalize every URL, then dedup URLs, content and records.
- Keep workers stateless; size the job with Little's law first.

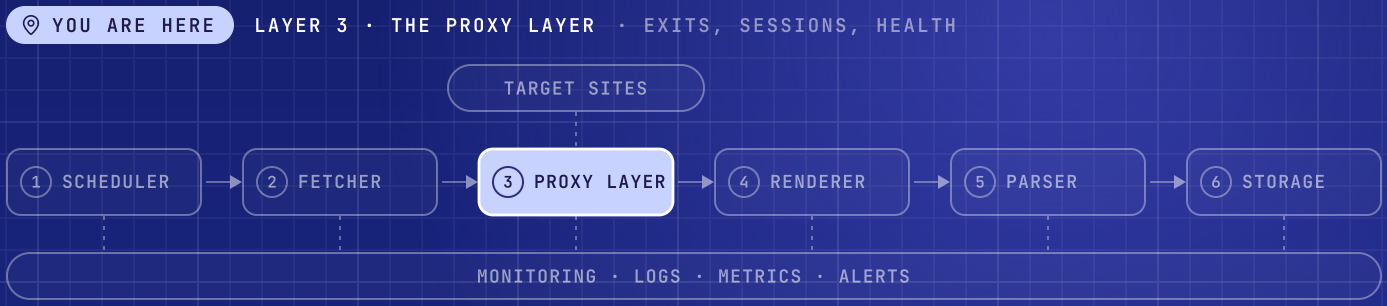
Go deeper: Web Crawler Request Queues · Stateless vs Stateful Scraping

07



The Proxy Layer

Proxies decide where your requests appear to come from, and on per-GB plans they're usually the biggest line on the bill. Use as little proxy as each target needs, keep the pool healthy, and measure cost per successful page, not per gigabyte.



IN THIS CHAPTER

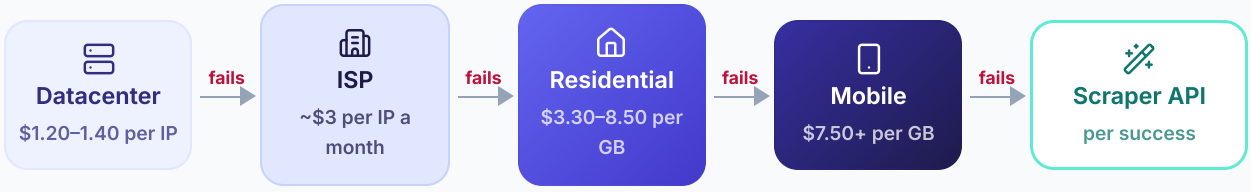
7.1 When you need proxies
7.2 Rotating, sticky and geo-targeted
7.3 Pool health and ban signals
7.4 Nine networks worth testing
7.5 Cost per 1,000 good pages

7.1 When You Need Proxies

Not every scraper needs a proxy, and buying one too early hides problems you should fix first. You need one when a single IP can't do the job politely: the site rate-limits per IP below the pace you need, it blocks cloud server ranges outright, it shows different content by country, or your volume simply outgrows one address.

> **New to proxy types?**
> This chapter assumes you know datacenter, ISP, residential and mobile proxies apart. If you don't, The Ultimate Proxy Handbook covers each in depth, free.

Escalate per target, starting at the cheapest rung



Re-test each target one rung lower every month. Defenses relax as often as they tighten.

Figure 7.1 The escalation path for one target. Entry prices across the nine providers in section 7.4, from the ProxyHorizon directory, November 2026.

Route per target, not per project

Target (example)	Settles on	Because
A documentation site	Datacenter	No bot protection; cheap, fast IPs are plenty
A retailer's listing pages	Datacenter ISP	Per-IP rate limits, but no fingerprinting
A marketplace's product pages	Residential	Challenges cloud ASNs on the first request
An app's mobile-only endpoints	Mobile	Expects carrier IPs, trusts them most

One project often spans all four rungs. A proxy **router**, one function that maps each target to its rung and provider, keeps that decision in config instead of scattered through your scrapers.

7.2 Rotating, Sticky and Geo-Targeted

Most scraping proxies sit behind a gateway that picks an exit IP for you. Two settings decide how it behaves: how long you keep an exit, and where in the world it has to be.

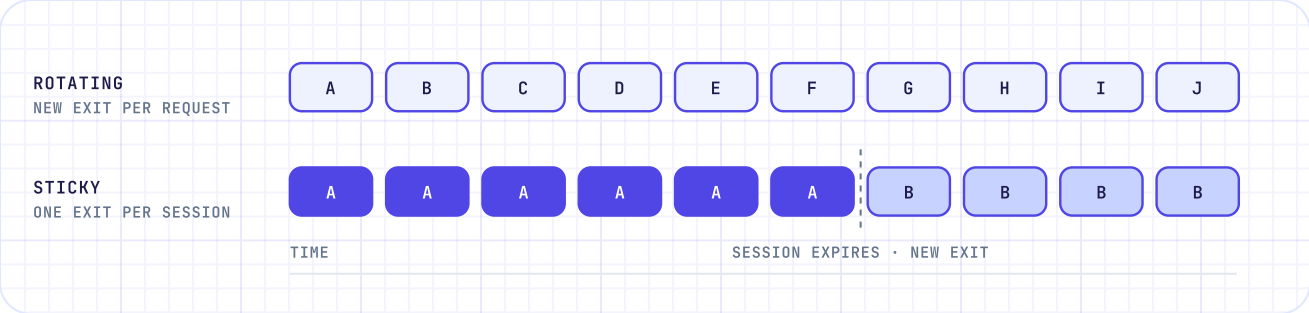


Figure 7.2 Ten requests, two policies. Letters are exit IPs. Rotation spreads load; a sticky session keeps one identity until it expires or you end it.

Policy	Use it for	Avoid it for
Rotate per request	Listings, search results, independent pages	Anything that needs cookies to persist
Sticky session	Paginating with cookies, browser sessions, multi-step flows	Bulk independent fetches; it wastes the IP
Static IP	Allowlisted APIs, long sessions, steady identities	Massive parallel crawls

Geo-targeting

Prices, stock, search results and even page layouts change by country and sometimes by city. Most gateways take the target country, region or city, plus a session ID, as flags in the proxy username. The exact format differs by provider, so build proxy URLs in one function and keep the format in config.

Python · one place to build proxy URLs (format is illustrative)

```python
def proxy_url(country="us", session=None):
    user = f"USER-country-{country}" + (f"-session-{session}" if session else "")
    return f"http://{user}:PASS@proxy.example.com:8000"    # check your provider's docs
```

Make the rest of the request agree with the exit

A German exit should ask for `de-DE`, carry euro-currency cookies and, in a browser, run on a Berlin timezone. A mismatched locale is one of the cheapest flags for a site to check.

7.3 Pool Health and Ban Signals

Exits go bad: IPs get flagged, sessions die, peers go offline. Score each exit, or each session behind a rotating gateway, on what actually happens to it.

Python · a rotating pool with health scores

```python
import random, time
from dataclasses import dataclass
@dataclass
class Exit:
    url: str                    # one proxy endpoint, or one sticky session
    score: float = 1.0          # 1.0 healthy, near 0 burned
    cool_until: float = 0.0

class ProxyPool:
    def __init__(self, urls):
        self.exits = [Exit(u) for u in urls]

    def pick(self) -> Exit:
        now = time.monotonic()
        live = [e for e in self.exits if e.cool_until <= now]
        if not live:
            raise RuntimeError("every exit is resting: slow down or add exits")
        return random.choices(live, weights=[max(e.score, 0.05) for e in live])[0]

    def report(self, e: Exit, ok: bool, banned: bool = False):
        if ok:
            e.score = min(1.0, e.score + 0.1)
        else:
            e.score *= 0.5 if banned else 0.8
            if banned:                                  # 403, 429 or a CAPTCHA page
                e.cool_until = time.monotonic() + 600   # rest it for ten minutes
```

Call `report()` with the validator's verdict, so soft blocks count as failures. Low scorers still get the odd try, so recovered exits can earn their way back.

Read the ban signals

Signal	What it says about the exit	What the pool does
`403` or a CAPTCHA page	This IP or session is flagged here	Halve the score, rest it for ten minutes
`429`	Usually the host's pace, not the IP	Slow the host down first (section 3.3)
`200`, empty or wrong page	A soft block the validator caught	Treat it like a 403
Timeouts, resets, `502`	A slow or dead exit	Small penalty; retry on another exit
`407`	Your credentials or plan, not the target	Stop and alert; no exit will work

7.4 Nine Networks Worth Testing

A neutral shortlist from the ProxyHorizon directory, in alphabetical order. All nine are credible for scraping; which one wins depends on your targets, locations and volume, so trial two or three on your hardest target and compare cost per 1,000 good pages (section 7.5).

Bright Data

Enterprise web data platform: proxies plus scraper APIs and ready-made datasets.

Residential · ISP · Datacenter · Mobile

RESIDENTIAL POOL	ENTRY PRICE
72M+	$5.04/GB

Decodo

Formerly Smartproxy. An easy dashboard and broad coverage across every proxy type.

Residential · ISP · Datacenter · Mobile

RESIDENTIAL POOL	ENTRY PRICE
115M+	$8.50/GB

Geonode

Developer-friendly residential proxies with unlimited-bandwidth plans.

Residential

RESIDENTIAL POOL	ENTRY PRICE
30M+	$4/GB

IPRoyal

Residential traffic that never expires, a good fit for irregular crawl schedules.

Residential · ISP · Datacenter · Mobile

RESIDENTIAL POOL	ENTRY PRICE
32M+	$7/GB

NodeMaven

Quality-filtered residential and mobile IPs, known for long sticky sessions.

Residential · Mobile

RESIDENTIAL POOL	ENTRY PRICE
30M+	$3.50/GB

Oxylabs

Premium enterprise network built for large-scale scraping, plus scraper APIs.

Residential · ISP · Datacenter · Mobile

RESIDENTIAL POOL	ENTRY PRICE
100M+	$8/GB

PYProxy

SOCKS5 per-IP pricing and an unlimited daily plan, alongside per-GB residential.

Residential · ISP · Datacenter

RESIDENTIAL POOL	ENTRY PRICE
90M+	$3.30/GB

SOAX

Clean pools with granular targeting down to city and ISP level.

Residential · Mobile

RESIDENTIAL POOL	ENTRY PRICE
155M+	$6.60/GB

Webshare

Affordable, self-serve datacenter proxies with highly customizable plans.

Datacenter · Residential

DATACENTER POOL	ENTRY PRICE
400K+	$2.99/mo

Product types, pool sizes and residential entry prices (Webshare: datacenter plans) from the ProxyHorizon directory, November 2026. Prices change often; each name links to its live directory page. Order is alphabetical and implies no ranking. Compare any two side by side at proxyhorizon.com/compare/proxies.

7.5 Cost per 1,000 Good Pages

Price per GB is on the pricing page; cost per 1,000 *good* pages is on your bill. It moves with price, success rate and megabytes per attempt.

Setup (worked example)	Price per GB	Success rate	MB per attempt	Cost per 1,000 good pages
A · cheapest plan, full pages	$3.50	70%	0.35	$1.75
B · premium plan, full pages	$8.00	96%	0.25	$2.08
C · mid plan, lean requests	$5.00	92%	0.18	**$0.98**

THE FORMULA

```
1,000 ÷ success rate × MB per attempt ÷
1,000 × price per GB
```

THE LESSON

Trimming request weight beat switching providers. Measure all three inputs.

Illustrative. Blocked attempts cost as much bandwidth as good ones; 1 GB is billed as 1,000 MB.

Cut bandwidth, not data

Tactic	Why it works
Call the JSON API, not the page	Kilobytes of data instead of a full HTML document (section 2.2)
Block images, media and fonts	Assets are most of a browser page's weight (section 4.3)
Use conditional requests	A `304` costs almost nothing on a per-GB plan (section 3.4)
Send easy targets to per-IP plans	Datacenter and ISP IPs aren't billed per gigabyte
Cap retries and crawl traps	Failed and pointless fetches cost as much as good ones

KEY TAKEAWAYS

- Use proxies only where one IP can't do the job politely, and route each target to its own rung.
- Rotate for independent pages, stick for sessions, and make locale match the exit.
- Score exits on validated outcomes and compare setups in cost per 1,000 good pages.

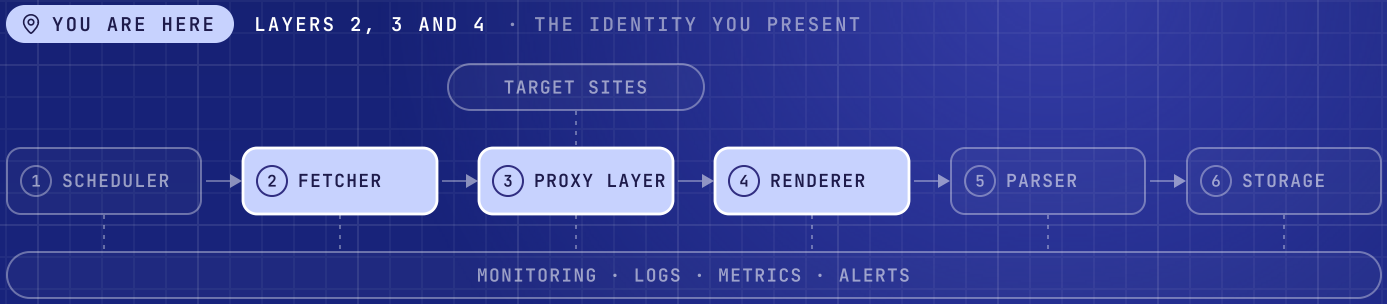
Free tools: Proxy Checker · Proxy Rotator · Proxy Format Converter · **Read:** Best Residential Proxies for Scraping

08



Staying Unblocked

Anti-bot systems don't block requests; they score them. Every layer of your request adds or removes risk, and the verdict comes from the total. Learn what gets scored, climb the escalation ladder only as far as a target forces you, and know when to stop.



IN THIS CHAPTER

8.1 How a request gets scored
8.2 The escalation ladder
8.3 Symptom, cause, fix

8.1 How a Request Gets Scored

Bot management in front of most large sites builds a risk score from many signals, starting with the first packet. Lenient sites look at the network layer and stop there. Strict ones check everything, and a request that fails early rarely gets a chance to redeem itself later.

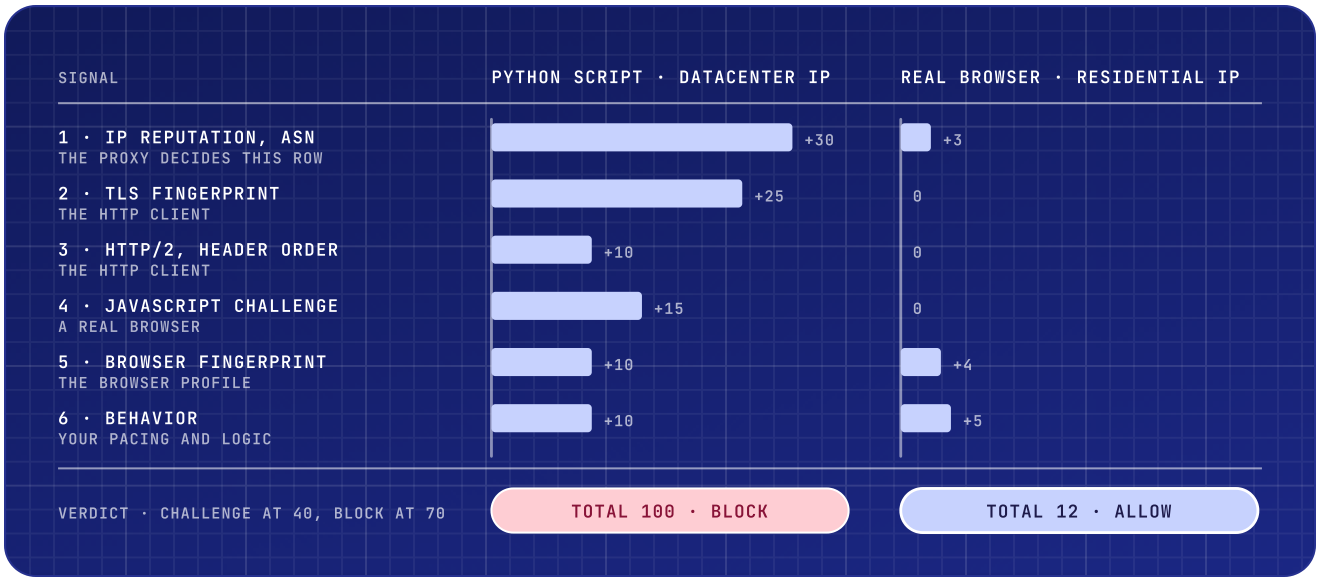


Figure 8.1 Two requests for the same page, scored. The points and thresholds are illustrative; real systems weigh far more signals with machine learning, but the shape holds: problems compound, and a good IP can't rescue a script's fingerprint.

Read the right-hand column of the figure as a to-do list. A proxy only changes row 1. Rows 2 and 3 belong to your HTTP client, row 4 needs a JavaScript engine, row 5 depends on how you configure the browser, and row 6 is your code's rhythm. That's why buying more IPs so often fails to fix a block: the points were coming from somewhere else.

See what sites see

Open ProxyHorizon's free My Fingerprint tool through your scraper's browser setup and compare it with a normal browser. Every difference is a row in Figure 8.1 that a strict site can score.

One identity, told consistently

- [ ] User-Agent and client hints name the same browser
- [ ] TLS and HTTP/2 match that browser
- [ ] Accept-Language matches the exit country
- [ ] Browser timezone matches the exit city
- [ ] WebRTC can't leak your server's real IP
- [ ] `navigator.webdriver` isn't true
- [ ] Cookies stay with one identity, never shared
- [ ] Requests arrive with human, jittered pacing

8.2 The Escalation Ladder

When a target pushes back, climb one rung at a time and stop at the first one that works. Each rung costs more than the last, and the top rung isn't a product: it's deciding that this data isn't worth taking this way.

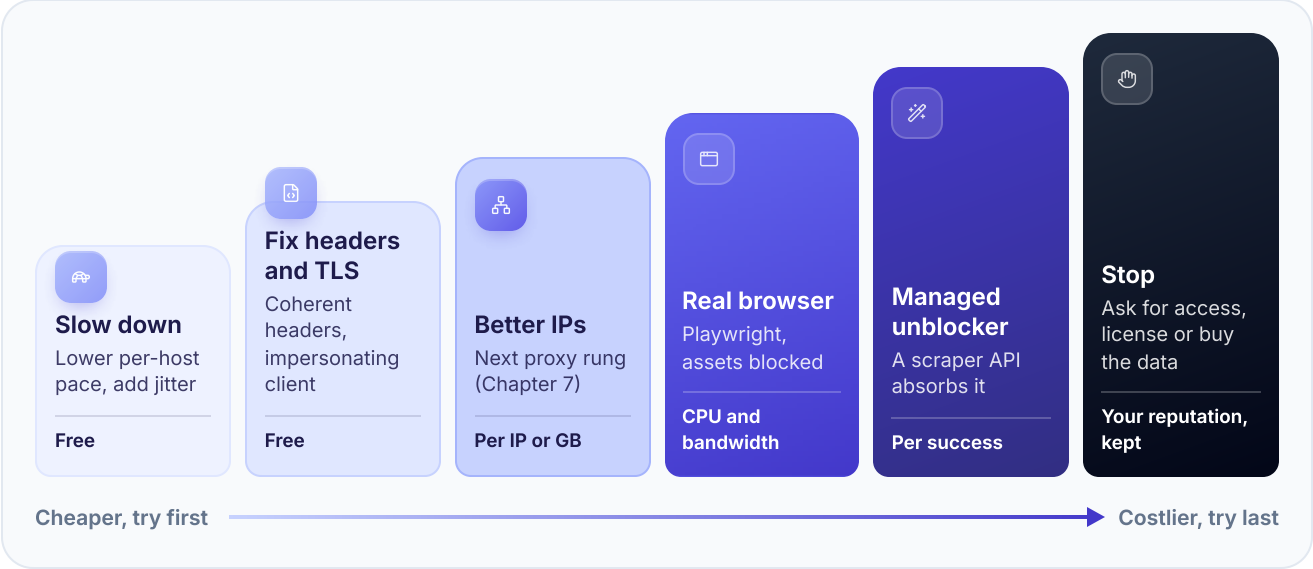


Figure 8.2 Most blocks are solved on the first two rungs, which cost nothing but attention.

Rung 2 without a browser

When a target fingerprints TLS, a client that impersonates a real browser's handshake and HTTP/2 settings often fixes it at a fraction of a browser's cost. In Python, curl_cffi keeps a requests-style API:

Python · curl_cffi with browser impersonation

```python
from curl_cffi import requests

r = requests.get("https://shop.example.com/p/trail-runner-4",
                 impersonate="chrome",                    # Chrome's TLS and HTTP/2 fingerprint
                 proxies={"https": "http://USER:PASS@proxy.example.com:8000"},
                 timeout=30)
```

CAPTCHAs are a symptom

A CAPTCHA means the site already distrusts you. Solving it treats the symptom at a cost per page; fixing the row that triggered it is cheaper at any real volume.

Never follow invisible links

Honeypots are links a human can't see: hidden with CSS, off-screen, or the color of the background. Following one flags you. Skip links that aren't visible.

8.3 Symptom, Cause, Fix

Blocks look alike from the outside and come from different places. Match the symptom first, fix the cause it points to, and only then reach for a higher rung.

Symptom	Likely cause	Fix
403 on the very first request, on any IP	TLS or header fingerprint	Coherent headers; impersonating client or browser
Works for a while, then 403s	IP reputation burning	Rotate sooner, slow down, climb one proxy rung
429s climbing through the day	Too fast for the host	Lower per-host concurrency; honor `Retry-After`
CAPTCHA after a few pages	Behavior or session pattern	Keep cookies per identity, add jitter, enter via listings
200 with an empty product grid	Soft block, or data loads later via XHR	Validate content; find the API (section 2.2)
Challenge page that never clears	JavaScript check fails in your client	Real browser, or a managed unblocker
Works on a laptop, fails on a server	Cloud ASN reputation	Proxies: datacenter first, then ISP or residential
Wrong prices or language	Exit, locale and cookies disagree	Match country, `Accept-Language` and currency
Blocked only on search or cart pages	Stricter limits on costly endpoints	Slow those paths; skip what robots.txt disallows

Keep a block log per target

Record the symptom, the rung that fixed it and the date. Six months later it tells you which targets tightened, which relaxed and which rung to start on.

KEY TAKEAWAYS

- Anti-bot systems add up risk across six rows; a proxy only changes the first.
- Climb the escalation ladder one rung at a time, and treat "stop" as a real option.
- Avoid CAPTCHAs instead of solving them, and never follow links a human can't see.
- Diagnose by symptom before you spend money on a higher rung.

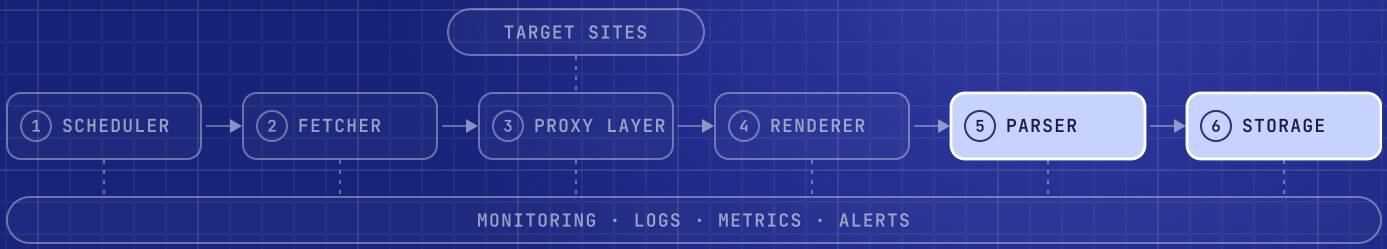
Go deeper: How Anti-Bot Systems Detect Bots · TLS Fingerprinting · Handling CAPTCHAs

09

Data Quality, Storage and Delivery

A scraper's real product isn't pages; it's a table someone trusts. Validate every record in layers, catch drift before your users do, keep enough history to answer "what changed?", and ship data with a contract that says what good means.

YOU ARE HERE LAYERS 5 AND 6 · QUALITY, STORAGE, DELIVERY



IN THIS CHAPTER

9.1 Validation layers and drift
9.2 Change detection and storage
9.3 Delivery, contracts and a dashboard

9.1 Validation Layers and Drift

No single check catches everything. Stack five cheap ones, each looking at a wider scope than the last, and a broken record has to slip past all of them to reach a dashboard.

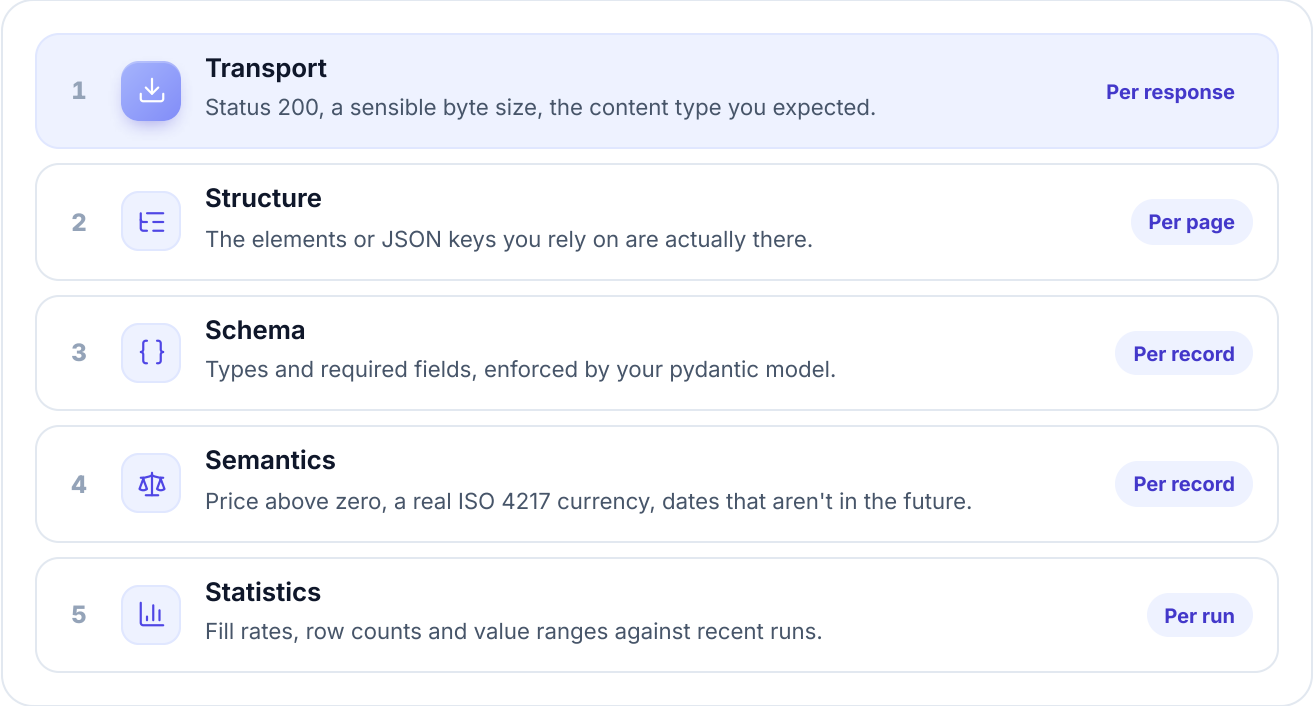


Figure 9.1 Five validation layers, from one response to a whole run. The first four catch broken records; only the fifth catches a scraper that's quietly wrong everywhere.

Schema drift: the target changed and nobody told you

Drift signal	Example	Alert when
A fill rate drops	`brand` falls from 92% to 38%	A field sits 2+ points under its 7-day average
Keys appear or vanish	The API adds `price_v2` and drops `price`	Any key set differs from yesterday's
A type changes	Price arrives as the text `"$89.00"`	One field's validation errors jump
Values shift scale	Median price jumps 100 times overnight	Values leave their normal band (cents vs dollars)
Row counts swing	A category returns 40% fewer items	Counts fall outside their usual range

Store the raw page, too

Keep compressed HTML or JSON for every fetch in cheap object storage for a few weeks. When a parser bug slips through, you re-parse from the archive instead of re-crawling, which saves proxy spend and catches the days you'd otherwise have lost.

9.2 Change Detection and Storage

Most of what a recurring scraper fetches hasn't changed since last time. Give every record a **natural key** (say, SKU plus store) and a **content hash** of its normalized fields. Then an upsert can tell new, changed and unchanged records apart in one statement, and you keep a clean history of what changed and when.

SQL · PostgreSQL upsert with a content hash

```
INSERT INTO products (sku, store, data, content_hash, first_seen, last_seen, changed_at)
VALUES ($1, $2, $3, $4, now(), now(), now())
ON CONFLICT (sku, store) DO UPDATE
SET data         = EXCLUDED.data,
    content_hash = EXCLUDED.content_hash,
    last_seen    = now(),
    changed_at   = CASE WHEN products.content_hash <> EXCLUDED.content_hash
                        THEN now() ELSE products.changed_at END;
```

Pair it with an append-only `product_changes` table written whenever the hash differs, and you can answer "when did this price change?" without storing every unchanged copy.

Where to keep it

	PostgreSQL with JSONB	Object storage + Parquet	Cloud warehouse
Best for	Current state, upserts, an API on top	Raw archives and history at low cost	Analysts running SQL over billions of rows
Scale	Comfortable into the hundreds of millions of rows	Effectively unlimited	Very large, priced by query or compute
Schema	Typed columns plus a flexible JSONB field	Columnar files, partitioned by date	Typed tables, loaded in batches
Watch for	Bloat from frequent updates; vacuum it	Many small files; compact them	Query costs that creep

Use more than one, on purpose

A common shape: raw pages and daily snapshots in Parquet on object storage, current state in Postgres for the app, and a warehouse table for analysts, all fed from the same validated records.

9.3 Delivery, Contracts and a Dashboard

Consumers pull data through an **API**, receive **exports** such as daily Parquet or CSV drops, get **webhooks** when something they care about changes, or query a shared **warehouse table**. Whichever you pick, write down what they can rely on. A data contract turns "the scraper's broken" into "the freshness promise was missed at 06:00".

competitor_prices.contract.json

```
{
  "dataset": "competitor_prices",
  "owner": "data-eng@example.com",
  "grain": "one row per SKU, store and day",
  "schema": {"sku": "string", "store": "string", "price": "decimal > 0",
             "currency": "ISO 4217", "in_stock": "boolean", "observed_at": "timestamp"},
  "freshness": "every active SKU observed within 24 hours",
  "quality": {"fill_rate_min": {"price": 0.99, "in_stock": 0.98}, "duplicate_rate_max": 0.001},
  "delivery": {"table": "analytics.competitor_prices", "webhook": "on price change"},
  "breaking_changes": "announced 14 days ahead"
}
```

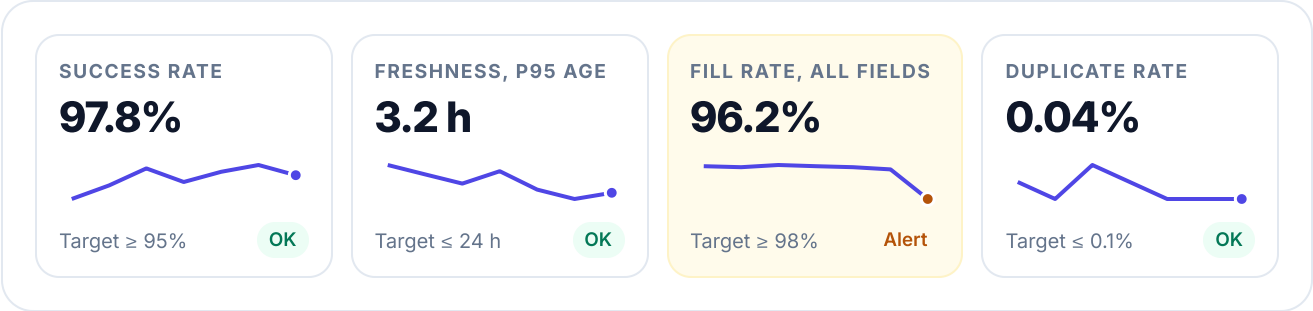


Figure 9.2 Four numbers worth a dashboard, with seven days of history each (example values). The fill-rate dip is the brand selector from Figure 5.2, caught the morning it broke.

KEY TAKEAWAYS

- Validate in five layers, from one response to a whole run, and alert on drift.
- Key every record, hash its content and upsert, so history costs little.
- Write a data contract, and watch success, freshness, fill rate and duplicates daily.

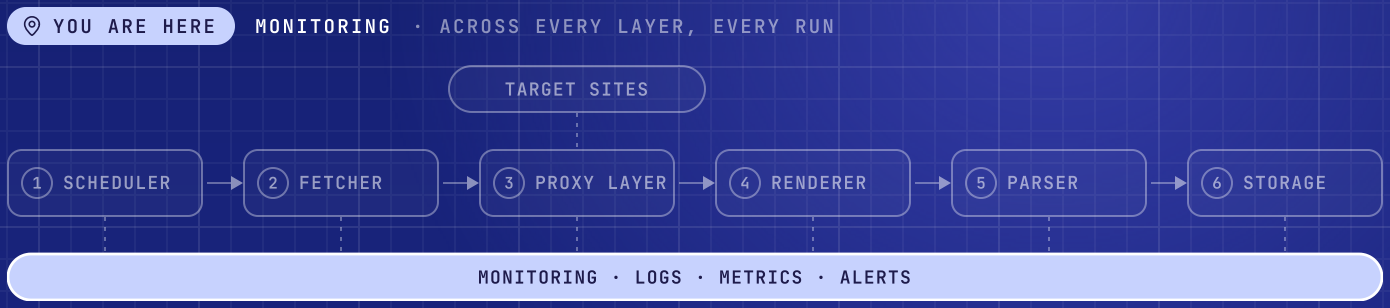
Go deeper: How to Clean and Store Scraped Data · E-commerce Price Monitoring

10



Running It in Production

A scraper in production is a promise with a schedule. Run it on something sturdier than a laptop's cron, watch the few numbers that matter, write down what to do when they slip, and know what each successful record costs.



IN THIS CHAPTER

10.1 Scheduling
10.2 SLOs, alerts and runbooks
10.3 The full cost model

10.1 Scheduling

Different data goes stale at different speeds, so one schedule rarely fits a whole project. Give each page type from your inventory (section 2.4) its own cadence, sized to how fast it really changes and how much a stale value costs.

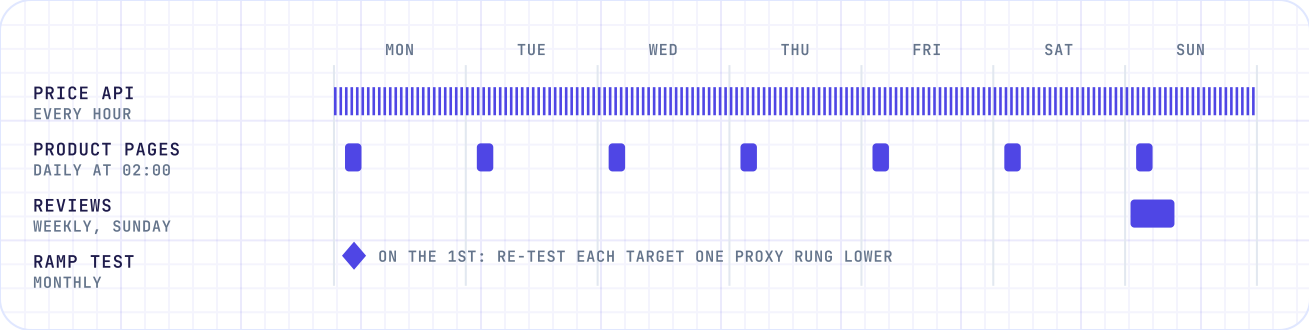


Figure 10.1 One week, four cadences (example). Volatile fields get small, frequent jobs; heavy pages run once a day in the target's quiet hours.

Run it on	Good for	Watch for
cron on one server	A handful of small jobs	Silent failures, no retries, one point of failure
Managed scheduler + serverless	Short, bursty jobs	Per-run time limits; browsers are awkward
Workflow orchestrator	Dependencies, backfills, many pipelines	More moving parts to run
Scheduler feeding a queue	Continuous crawling at volume	You build the run dashboards yourself

Airflow, Dagster and Prefect are the common orchestrators; Celery beat or a small producer feeding Redis or SQS covers the queue pattern.

Every scheduled run needs

- [ ] Idempotent writes, so a rerun can't double-count
- [ ] A lock, so two runs never overlap
- [ ] A run ID in every log line
- [ ] A heartbeat that alerts if the run never starts
- [ ] Backfill by date range, without code changes
- [ ] A hard time limit, then a clean exit

10.2 SLOs, Alerts and Runbooks

Turn the dashboard from section 9.3 into promises. A **service level objective** (SLO) states the target; an alert fires when you're on track to miss it. Page a human only for what can't wait until morning, and file tickets for the rest.

Indicator	Objective (example)	Page someone when	File a ticket when
Validated success rate, per target	≥ 95% over 24 hours	Under 80% for an hour	Under 95% for six hours
Freshness, p95 age	≤ 24 hours	Over 20 hours	Over 12 hours, twice in a week
Fill rate, per field	≥ 98%	A key field under 90%	2 points under its 7-day average
Duplicate rate	≤ 0.1%	Over 2%	Over 0.5%
Cost per 1,000 good pages	Within budget	Double last week's	30% above last month's

A runbook: success rate dropped on one target

1. **Scope it.** One target or all of them? All of them points at your proxy plan, credentials or code, not the site.
2. **Look at five failures.** Status codes, response sizes and the first lines of each body.
3. **Match the symptom** in the table in section 8.3 and note the likely cause.
4. **Apply the lowest rung that fixes it,** then confirm on 50 requests before restoring full volume.
5. **Backfill the gap** from the raw archive, or with a targeted re-crawl of the missed URLs.
6. **Write it down:** symptom, cause, fix and date in the target's block log.

> **Alert on validated outcomes, never on exit codes**
>
> A job that exits 0 after writing ten empty rows is a failure. Every alert above reads from the validator, which is why Chapter 5 asked you to quarantine instead of drop.

10.3 The Full Cost Model

Say you scrape 2 million pages a month: 1.4 million over 50 datacenter IPs at $1.30 each, 500,000 over residential at $5 per GB and 100,000 in a browser over residential, at 95% success, 0.2 MB per HTTP attempt and 0.8 MB per browser page.

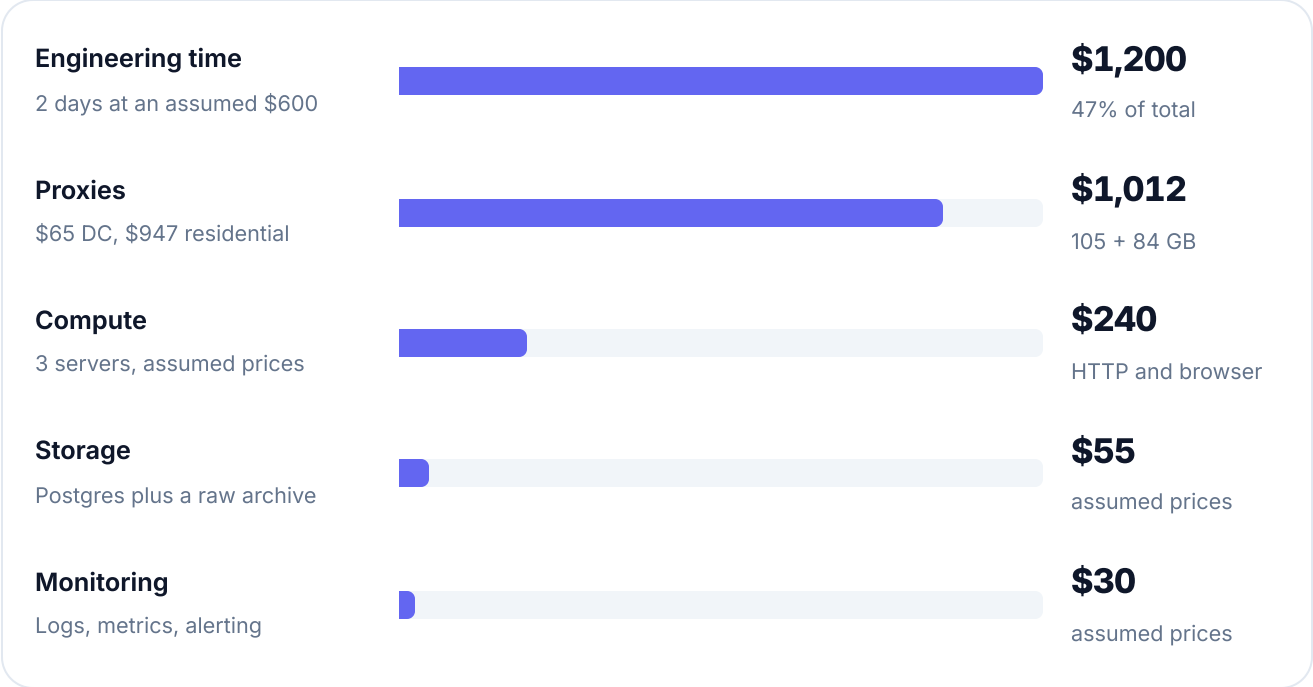


Figure 10.2 A worked model: $2,537 a month, or $1.27 per 1,000 pages. Without people's time it's $0.67, so always count it.

Build or buy, revisited

Now say a scraper API quotes $1.50 per 1,000 successful requests for these targets: $3,000 a month. DIY wins by about $460, but only because most pages are easy. Shift the mix toward hard, browser-only targets and the API wins; double the volume and DIY pulls ahead, since engineering time barely grows. The honest answer is usually a hybrid: DIY for the bulk, an API for the few targets that fight back. Buy wherever an API's price per 1,000 beats your DIY number for the same targets.

KEY TAKEAWAYS

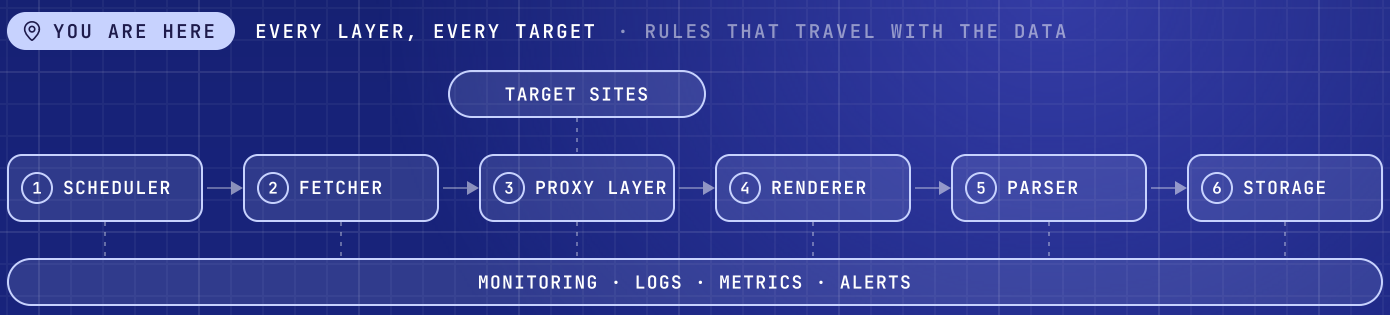
- Give each page type its own cadence; make every run idempotent, locked and observable.
- Write SLOs from validated metrics, page for what can't wait and keep a runbook.
- Price the whole system, people included, before deciding to build or buy.

Go deeper: Scheduling and Monitoring Scrapers · Cheapest Proxies for Scraping

11

Scraping Responsibly

Whether scraping is acceptable usually comes down to what you collect, how you collect it and what you do with it afterwards. Settle those questions before launch, design them into every layer, and you'll rarely have to defend a scraper you're not proud of.



IN THIS CHAPTER

11.1 Public, gated and personal data
11.2 Terms, copyright and privacy law
11.3 Etiquette and a launch checklist

11.1 Public, Gated and Personal Data

The first question isn't "can I get it?" but "what kind of data is it, and where does it sit?" Data on an open product page and data behind a login are different propositions, and data about people is different again, even when it's public.

This isn't legal advice

Laws differ by country and change often, and courts weigh the facts of each case. Use this chapter to ask better questions, and talk to a lawyer who knows your jurisdiction before anything high-stakes.

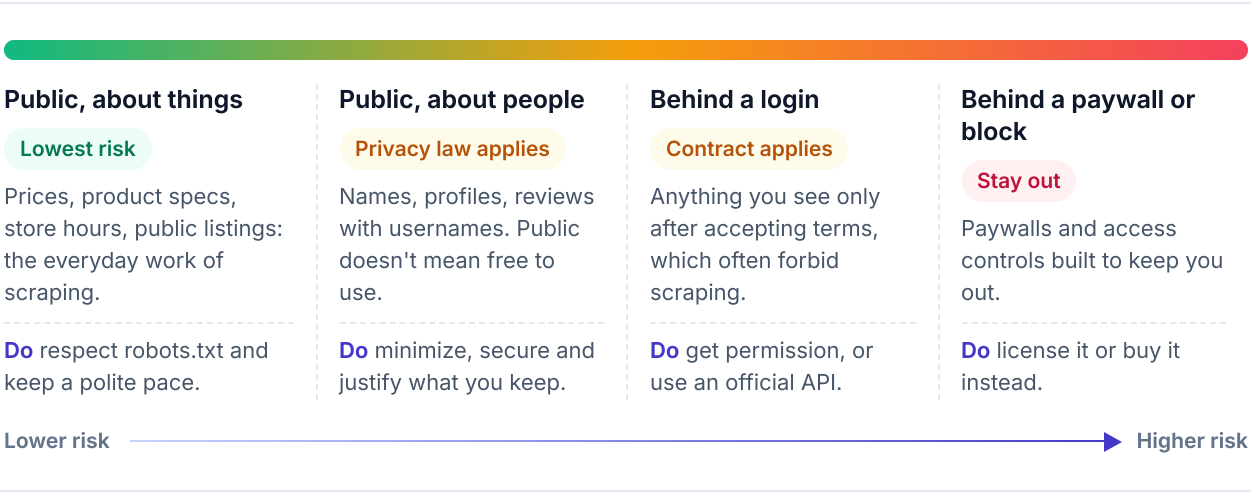


Figure 11.1 Where data sits shapes the risk of collecting it. Most commercial scraping lives in the first zone, and that's where this book's patterns are aimed.

Minimize at the parser, not in a cleanup job

The cheapest personal data to protect is the data you never stored. If a review page gives you a rating, a date and a reviewer's name, and you only need the rating and the date, drop the name in the parser (layer 5) so it never reaches storage, logs or backups. Set a retention period for raw page archives too, since they hold everything the page did.

What you do with it matters as much as how you got it

Use	Why it matters
Aggregate facts into analytics	Prices, stock and rankings are facts: the lowest-risk use of scraped data
Republish content word for word	Articles, photos and reviews can be protected; terms often forbid it too
Build profiles of identifiable people	The core concern of privacy law; you need a lawful basis and a clear purpose
Train models or feed AI agents	Licenses, terms and rules are still settling, so check them for each source

11.2 Terms, Copyright and Privacy Law

Five areas of law come up again and again. None of them says "scraping is illegal," and none of them says it's always fine. Each asks a different question about what you did.

Area	The question it asks	A sensible stance
Terms of service	Did you agree to rules, for example by creating an account?	Don't scrape logged-in areas without permission; read public terms for scraping clauses
Computer misuse laws	Did you get around a barrier meant to keep you out?	Never bypass logins, paywalls or blocks aimed at you
Copyright	Did you copy creative expression rather than facts?	Prices and specs are facts; articles, photos and reviews may be protected
Database rights	In the EU, did you extract a substantial part of a protected database?	Take the fields you need, not a copy of the whole collection
Data protection	Can the data identify a living person?	Have a lawful basis, minimize, secure, set retention, honor requests

Privacy law, in brief

The EU's GDPR covers personal data of people in the EU wherever it's processed, and publicly visible data is still personal data. Its principles, including purpose limitation, data minimization and storage limitation, map directly onto layers 5 and 6, and its fines can reach 4% of global annual turnover. In the US, California's CCPA gives residents rights over personal information that businesses collect; its definition carves out some publicly available information, but the details matter, and other states have their own laws.

What the courts have shown

Disputes tend to turn on contracts and access more than on scraping itself. The long-running **hiQ Labs v. LinkedIn** case in the US is the best-known example: the Ninth Circuit doubted that scraping public pages counted as hacking, yet the case ended in 2022 with hiQ found to have breached LinkedIn's user agreement.

robots.txt is a signal worth honoring

RFC 9309 is clear that robots.txt isn't access control. It is, however, the site owner's written request, and following it is the baseline for crawling in good faith. If you ever decide to ignore a rule, decide deliberately, with a reason you could defend.

Go deeper: Is Web Scraping Legal? robots.txt Explained · Ethical Residential Proxy Sourcing

11.3 Etiquette and a Launch Checklist

Being a good guest costs little and keeps the targets you depend on healthy. Most of it is already built into the earlier chapters; this page collects it in one place.

Go gently

Stay well under each host's measured ceiling, honor `Retry-After`, back off on errors and crawl heavy jobs in off-peak hours.

Be identifiable when you can

For open crawling, a descriptive User-Agent with a contact URL lets site owners reach you before they block you.

Don't fetch twice

Conditional requests, sitemap dates and a raw archive mean fewer requests for the same data.

Stop when asked

If an owner objects, pause and talk. Licensing or an official API often follows a polite reply.

The responsible-scraping checklist

- [ ] Data type placed on the spectrum (11.1)
- [ ] robots.txt read and its rules applied
- [ ] Only the fields you need are collected
- [ ] Retention periods set for records and raw pages
- [ ] Facts extracted; creative content not republished
- [ ] No logged-in or paywalled areas without permission
- [ ] Terms of service checked for scraping clauses
- [ ] Personal data dropped or justified in writing
- [ ] Per-host limits below the measured ceiling
- [ ] A named owner who answers site complaints

KEY TAKEAWAYS

- Place the data on the spectrum first; public facts about things carry the least risk.
- Contracts, access, copyright, database rights and privacy each ask a different question.
- Minimize at the parser, pace gently, and treat robots.txt as the owner's request.

The best scraping systems are boring: they fetch less than they could, keep only what they need and run for years without anyone noticing them. Build yours that way.

APPENDIX A

Pre-Launch Checklist

Print it, walk through it layer by layer, and don't put a scraper on a schedule until every box is ticked or consciously waived. Section numbers point back to the details.

Target and recon

- [ ] robots.txt read, saved and applied (2.1)
- [ ] Sitemaps found; URL counts known (2.1)
- [ ] Best data source chosen per page type (2.2)
- [ ] Pagination pattern documented (2.3)
- [ ] Rate ceiling measured per host (2.3)
- [ ] Page-type inventory complete (2.4)
- [ ] Terms checked; no login or paywall scraping (11.2)

Fetching, rendering and proxies

- [ ] Connect and read timeouts set (3.3)
- [ ] Retries only on 429, 5xx and network errors (3.3)
- [ ] Full-jitter backoff; `Retry-After` honored (3.3)
- [ ] Coherent headers; one identity per session (3.2)
- [ ] Browser only where needed; assets blocked (4.1)
- [ ] Each target routed to its cheapest working rung (7.1)
- [ ] Pool scored on validated outcomes (7.3)

Parsing, data and storage

- [ ] Two selectors per field, highest level first (5.1)
- [ ] Typed model; failures quarantined (5.3)
- [ ] Fill rate tracked per field (5.4)
- [ ] URLs canonicalized; dedup keys defined (6.2)
- [ ] Upserts keyed and content-hashed (9.2)
- [ ] Raw archive with a retention period (9.1)
- [ ] Data contract written and shared (9.3)

Operations and responsibility

- [ ] Per-host limits in config, below the ceiling (6.3)
- [ ] Runs idempotent, locked and observable (10.1)
- [ ] SLOs and alerts on validated metrics (10.2)
- [ ] Runbook written for the top failure (10.2)
- [ ] Cost per 1,000 good pages tracked (7.5)
- [ ] Personal data minimized at the parser (11.1)
- [ ] A named owner answers site complaints (11.3)

The first week after launch

Day 1

Run at a quarter of full volume. Watch validated success and fill rate per target.

Days 2 and 3

Compare records against your saved fixtures; fix any selector that drifted.

Day 4

Raise to full volume if every SLO held, one target at a time.

Day 7

Review cost per 1,000 good pages and the block log; adjust rungs.

APPENDIX B

Cheat Sheets

HTTP status codes for scrapers

200 OK	Validate the content before you trust it.	**301 / 308**	Moved for good. Follow; update the frontier.
302 / 307	Temporary. Follow; check where you landed.	**304 Not Modified**	Unchanged. Reuse your stored copy.
401 Unauthorized	Login required. Stop unless you have permission.	**403 Forbidden**	Blocked. Diagnose (8.3), then escalate.
404 / 410	Gone. Stop scheduling; 410 is permanent.	**407 Proxy Auth**	From your proxy. Check credentials and plan.
429 Too Many	Slow down. Honor `Retry-After`.	**500 / 502 / 503 / 504**	Server or gateway trouble. Retry with backoff.

CSS and XPath, side by side

Goal	CSS	XPath
By ID	`#price`	`//*[@id="price"]`
By class	`.price`	`//*[contains(@class, "price")]` (also matches substrings)
Attribute equals	`[data-sku="123"]`	`//*[@data-sku="123"]`
Attribute starts with	`[href^="/p/"]`	`//*[starts-with(@href, "/p/")]`
Direct child	`ul > li`	`//ul/li`
Next sibling	`h2 + p`	`//h2/following-sibling::*[1][self::p]`
Third of its type	`li:nth-of-type(3)`	`//li[3]`
Contains text	*Not in CSS*	`//*[contains(text(), "In stock")]`
Parent	`:has()`, where supported	`//span[@class="price"]/..`
Attribute value	`a::attr(href)` in parsel	`//a/@href`
Text content	`h1::text` in parsel	`//h1/text()`

APPENDIX C

Glossary

Backoff with jitter

Waiting longer after each failure, at a random point inside a growing window, so retries don't arrive in waves.

Bloom filter

A compact structure that answers "seen this URL?" with no false negatives and a small, tunable rate of false positives.

Canonical URL

The one normalized form of a URL that all its variants map to before deduplication.

Circuit breaker

A per-host switch that pauses traffic when errors spike, then probes before resuming.

Content hash

A fingerprint of a page's or record's normalized content, used to detect duplicates and changes.

Crawl trap

A URL space that never ends, such as calendars or filter combinations, that eats crawl budget.

Data contract

A written promise about a dataset's schema, freshness, quality and delivery.

Fill rate

The share of records in which a field is present and valid; the first metric to drop when a selector breaks.

Frontier

The crawler's queue of URLs to visit, organized by priority and by host.

Honeypot link

A link hidden from humans; following it identifies you as a bot.

Idempotent run

A job you can run twice with the same result, because writes are keyed and upserted.

JA3 / JA4

Hashes of a client's TLS handshake that identify the software behind a request.

JSON-LD

Structured data in a script tag, written for search engines and often the most stable place to read a field.

Little's law

Requests in flight equal arrival rate times time per request; it sizes concurrency.

Politeness

Per-host limits on concurrency and pace that keep a crawler from straining a site.

Proxy router

One function that maps each target to its proxy type, provider and session policy.

Schema drift

A change in a target's structure or data types that silently breaks extraction.

SLO

A service level objective: the target value for an indicator such as freshness or success rate.

Soft block

A blocked request that returns 200 with a challenge or empty page instead of an error.

Sticky session

Keeping one exit IP for a set time so multi-step flows stay on one identity.

More terms, with longer explanations, in the ProxyHorizon Glossary at proxyhorizon.com/glossary.

APPENDIX D

Free Tools and Further Reading

Everything below is on ProxyHorizon. The browser tools need no account; the site-analysis tools need a free one.

Free tools, no account

Proxy Checker	Speed, anonymity, uptime	Proxy Rotator	Rotation scripts
Proxy Format Converter	Switch list formats	My Fingerprint	What your browser leaks
cURL Builder	Build test requests	Random User-Agent	Current UA strings
User-Agent Parser	Decode any UA	JSON Formatter	Read API responses

Site-analysis tools, free account

Robots.txt Tester	Check URLs against rules	Sitemap Visualizer	Sections and sizes
Schema Extractor	JSON-LD on any page	Technology Checker	Framework and CDN
URL Extractor	Every URL a site exposes	Link Extractor	Links on any page

Guides worth reading next

- Web Scraping With Python
- Scrapy With Proxies
- Python Requests With Proxies
- Browser Automation vs HTTP
- Rate Limiting in Scraping
- Why Scraping Needs Proxies
- Scraping Cloudflare Sites
- Node.js Scraping With Proxies

Directories and use cases

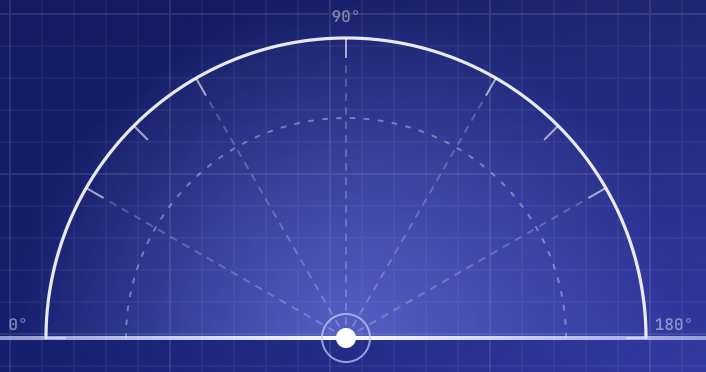
Proxy providers	Tested and rated	Compare proxies	Any two, side by side
Web scraping use case	Picks and setups	Antidetect browsers	For browser identities

ProxyHorizon

Compare confidently. Implement correctly.

ProxyHorizon is the independent directory and resource hub for proxies, VPNs and antidetect browsers: expert-tested comparisons, free tools and practical guides to choose and use them.

Compare 120+ proxy providers · Free proxy and scraping tools · In-depth, fact-checked guides



MORE FREE EBOOKS FROM PROXYHORIZON

The Ultimate Proxy Handbook
How proxies work, which type to buy and how to run them.
Free PDF

The Antidetect Browser Handbook
Browser profiles and fingerprints, managed with care.
Free PDF

Proxies for AI Agents and LLM Data
Web access for agents and data for models.
Free PDF

Lead Generation with Public Web Data
Finding and enriching leads from public sources.
Free PDF

© 2026 PROXYHORIZON · THE SCRAPING BLUEPRINT