ProxyLook

FREE EBOOK · 2026 EDITION

Scraping Without Blocks

A developer's guide to proxies, fingerprints and reliable data collection.

6
chapters, from first block to production

Python
httpx and Playwright code that runs as written

0
sign-ups, gates or affiliate links

proxylook.com/ebooks

FREE EBOOK · 2026 EDITION

Scraping Without Blocks

A developer’s guide to proxies, fingerprints and reliable data collection.

Published by ProxyLook · proxylook.com · First published November 2026

ProxyLook is an independent directory of proxy networks and antidetect browsers. This book is free. Share it with anyone, unchanged and with credit to ProxyLook.

Written and edited by the ProxyLook editorial team. The book is about collecting publicly available data responsibly. Personal data, logged-in content and terms of service raise separate legal questions.

Prices, pools and features change often. This book links to live figures on proxylook.com instead of printing numbers that go stale. Vendor-stated figures are labelled as such. The book contains no affiliate links.

Nothing here is legal advice. Laws and platform rules differ by country and by service. Check the rules that apply before acting.

© 2026 ProxyLook. Product names belong to their owners.

START HERE

How to use this book

A crawler that worked on Friday returns empty pages on Monday. The status code is 200, the proxy bill is higher and nobody changed the code. This book is about that Monday.

It is written for developers and data engineers who collect public web data. It assumes working Python and a rough idea of what a proxy is. The Proxy Playbook covers the basics.

The book follows the order a block is debugged in. It starts with how detection works, then picks proxies for the target, builds a client that fails gracefully, moves to headless browsers only where they are needed, weighs scraper APIs against building in-house, and ends with running the whole thing in production.

49	6	8	11	16
PAGES	CHAPTERS	DIAGRAMS	TABLES	CODE EXAMPLES

Every code block runs as written once the placeholders are replaced. Timeouts are set, TLS verification stays on, and nothing depends on a particular provider.

The book explains how detection works so that a client can behave well and fail clearly. It does not promise a way past any specific anti-bot product, because no honest guide can.

CONTENTS

What is inside

PART I · DIAGNOSE

01 **Anatomy of a Block** 5

One request, several observers / The detection stack / How a block presents / Reading the symptom / Why a proxy fixes only the first two layers / Scope and responsibility

02 **Picking Proxies for the Target** 11

Sorting targets by what can be observed / Which proxy type fits which tier / Rotation per request or sticky sessions / Geo-targeting for localized content / A routing ladder that starts cheap / Quick decisions

PART II · BUILD

03 **Building a Resilient Client** 17

The base client / One identity per session ID / Different failures, different responses / The retry loop / Catching soft blocks in the body / A bounded worker pool

04 **Headless Browsers Through Proxies** 25

When a browser is needed, and when it is waste / What a browser costs / A proxy per browser context / Keeping locale and time zone consistent / Blocking heavy resources / Reusing a context for sticky work

PART III · OPERATE

05 **Scraper APIs and Unblockers** 31

What these products do / Build or buy / How per-request pricing changes the math / Where an API fits in the ladder / Lock-in and observability / Decision table

06 **Running in Production** 37

Queues and idempotent jobs / Per-domain rate limits / Metrics, budgets and alerts / Data-quality checks that catch soft blocks / Logging enough to debug a block / Etiquette, law and launch

REFERENCE

A **Appendix** 44

HTTP status quick reference / Pre-flight checklist / Debugging order / Glossary / ProxyLook tools and further reading

CHAPTER 01

Anatomy of a Block

Follows one request from script to block page, names the six detection layers it passes, and shows which symptom points to which layer.

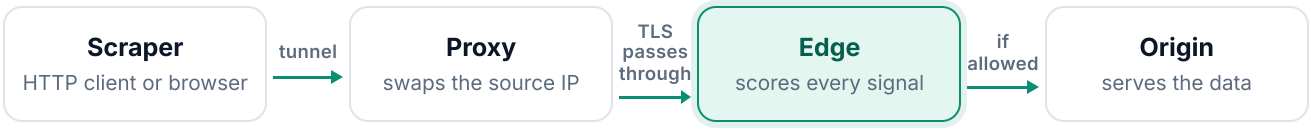
- The six layers of detection
- Mapping symptoms to layers
- How blocks present on the wire
- What a proxy cannot fix

A script sends `GET /products?page=7` and gets back a 403 with a 4 KB page that explains nothing. Somewhere between the socket opening and that response, a system decided the client was not a person in a browser. This chapter follows the request through each point where that decision can be made.

One request, several observers

A scraper rarely talks to the origin server directly. Most commercial sites sit behind an edge layer: a CDN, a web application firewall or a bot-management service. The edge sees every connection first.

The edge collects evidence in the order the protocol reveals it. The source IP arrives with the TCP handshake. The TLS ClientHello comes next, then the HTTP/2 settings, then the request headers. If a page is served, JavaScript can report on the runtime.



- The edge judges the request before the origin sees it.

Two details matter later. The proxy forwards the TLS handshake unchanged, so the edge still sees the client's own handshake. The edge can also answer on the origin's behalf, which is why a block page often looks nothing like the site.

The detection stack

Detection products differ in detail, but their signals fall into six layers. Lower layers are cheap and run on every request. Higher layers cost more and tend to run when the lower ones leave doubt.

1 · IP reputation and ASN	Who owns the address, and what it has done before.
2 · Rate and pattern	How many requests, how fast, in what order.
3 · TLS and HTTP/2 fingerprint	Which software opened the connection.
4 · Headers and their order	Whether the request matches the browser it claims to be.
5 · JavaScript and browser fingerprint	What the runtime reports when code runs in the page.
6 · Behavior	Whether the session moves like a visitor over time.

The six layers, in the order the evidence arrives.

Layers 1 and 2: the address and the tempo

Every IP belongs to an autonomous system, and the ASN says what kind of network that is. A cloud hosting ASN carries very little human browsing, so some sites refuse or challenge it on the first request. Reputation data adds history from earlier abuse.

Rate checks count requests per IP, per subnet and per session. Pattern checks look at shape. Examples are pages 1 to 500 in strict order, identical gaps between requests, or HTML fetched without its assets.

Layers 3 and 4: the software signature

A TLS ClientHello lists cipher suites, extensions and supported groups in an order chosen by the TLS library. Methods such as JA3 and JA4 hash those fields into a short fingerprint. Python's `ssl` module, Go's standard library, curl and Chrome each produce a different one. HTTP/2 adds more: the SETTINGS values a client sends and the order of its pseudo-headers also vary by implementation.

Headers are checked for agreement. A `User-Agent` that claims Chrome should arrive with Chrome's headers, in Chrome's order, over a handshake that looks like Chrome. A Chrome string on a Python handshake is a contradiction, and contradictions are cheap to detect.

Layers 5 and 6: the runtime and the session

When a page loads, a script can inspect the environment. It reads canvas and WebGL output, fonts, screen size, time zone and language. It also reads automation flags such as `navigator.webdriver`. A plain HTTP client never runs that script, and the missing result is a signal too.

Behavior is judged across a session. Real visitors pause, scroll and wander. A session that requests 300 product pages in a minute reads differently.

How a block presents

Blocks do not share one format, even on a single site.

403 Forbidden

Refused outright. The cause is usually identity: the IP, the fingerprint or the headers. Repeating it unchanged rarely helps.

429 Too Many Requests

A rate limit tripped. A `Retry-After` header sometimes says how long to wait.

CAPTCHA or challenge page

An interstitial replaces the content. The status varies: 403, 429, 503 and even 200 all occur.

Redirect

A 302 to a login page, a regional site or a challenge URL. A client that follows it silently stores the wrong page.

Soft block

Status 200 and a page that looks right but is not. Prices are missing, results are fewer, or fields are empty.

Stall or reset

The connection hangs until the timeout, or drops during the handshake.

Soft blocks deserve the most attention because nothing fails. Status monitoring stays green while the dataset degrades. Chapters 3 and 6 cover the checks that catch them.

A short probe makes the presentation visible. The script fetches one URL directly and through a proxy, then prints the status, redirect chain, final URL and body size.

PYTHON · HTTPX

```python
import httpx

PROXY = "http://USERNAME:PASSWORD@gate.example-proxy.com:7000"
URL = "https://example.com/"


def probe(label: str, proxy: str | None) -> None:
    with httpx.Client(
        proxy=proxy, timeout=20.0, follow_redirects=True
    ) as client:
        resp = client.get(URL)
    hops = [hop.status_code for hop in resp.history]
    print(label, resp.status_code, hops, resp.url, len(resp.content))
    for name in ("server", "content-type", "retry-after"):
        print("   ", name, "=", resp.headers.get(name))


probe("direct", None)
probe("proxied", PROXY)
```

Different results on the two routes point at layer 1. The same failure on both points higher up the stack. A body far smaller than the page in a desktop browser suggests a challenge or a soft block.

Reading the symptom

The table maps common symptoms to the most likely layer. It is a starting point, not a verdict: one symptom can have two causes.

Symptom	Likely layer	First check
403 on the first request from a datacenter IP	1 · IP and ASN	Repeat from a residential or ISP exit
Works for a while, then 429	2 · Rate	Lower the per-domain rate; read `Retry-After`
HTTP client fails on every IP; a browser works	3 · TLS and HTTP/2	Confirm with a real browser through the same exit
403 with default library headers only	4 · Headers	Diff the headers against a browser's request
Challenge page never clears in a headless browser	5 · Browser fingerprint	Check automation flags, locale and time zone

Symptom	Likely layer	First check
Blocks start deep into a long session	6 · Behavior	Add pauses; shorten sessions
200 with missing or altered fields	Any, often 1 or 5	Compare the body with a known-good page
Redirect to another country's site	1 · IP location	Verify the exit country

Common symptoms, the likely layer and the first check.

For the last row, the ProxyLook proxy checker reports the country and network of an exit IP. The user-agent parser shows what a header string claims to be.

Why a proxy fixes only the first two layers

A proxy changes one thing: the address the target sees. That answers layer 1 directly. A datacenter range with a poor record can be swapped for a residential, ISP or mobile address. The Proxy Playbook covers those types in detail.

A rotating pool also helps with layer 2. It spreads requests over many addresses, so no single IP crosses a per-IP limit. That help is partial. If the site groups requests by fingerprint instead of by IP, a thousand addresses with one identical handshake still look like one client.

Layers 3 to 6 are produced by the client and travel through the proxy tunnel untouched. The TLS handshake belongs to the client library. No JavaScript runs unless the client is a browser. Better IPs for a layer 3 problem cost money and change nothing.

> **KEY IDEA** **Diagnose before buying**
>
> The layer decides the fix. Layers 1 and 2 are proxy problems. Layer 3 and above are client problems, and they call for a better client, a real browser or a managed API.

Scope and responsibility

This book is about collecting publicly available data at a rate the target can absorb. It explains how detection works so that a client can behave well and fail gracefully. It does not explain how to break CAPTCHAs or get past logins and paywalls. No technique in it is promised to get through any particular anti-bot product.

WATCH OUT **Not legal advice**

Public pages, personal data, content behind a login and a site's terms of service each raise separate legal questions. The answers vary by country and by use. This book is not legal advice, and teams with real exposure should ask a lawyer first.

A sound default: decide which fields are needed, take only those, respect `robots.txt` and keep the rate modest. The ProxyLook web scraping page lists providers aimed at this work.

KEY TAKEAWAYS

- A request is judged on six layers: IP, rate, TLS and HTTP/2, headers, browser fingerprint and behavior.
- Blocks arrive as 403s, 429s, challenge pages, redirects, stalls and soft blocks that pass every status check.
- A proxy changes the source address, which addresses layer 1 and part of layer 2 only.
- Layers 3 to 6 come from the client, so they need a better client and not better IPs.
- Personal data, logged-in content and terms of service raise separate legal questions from public pages.

CHAPTER 02

Picking Proxies for the Target

02

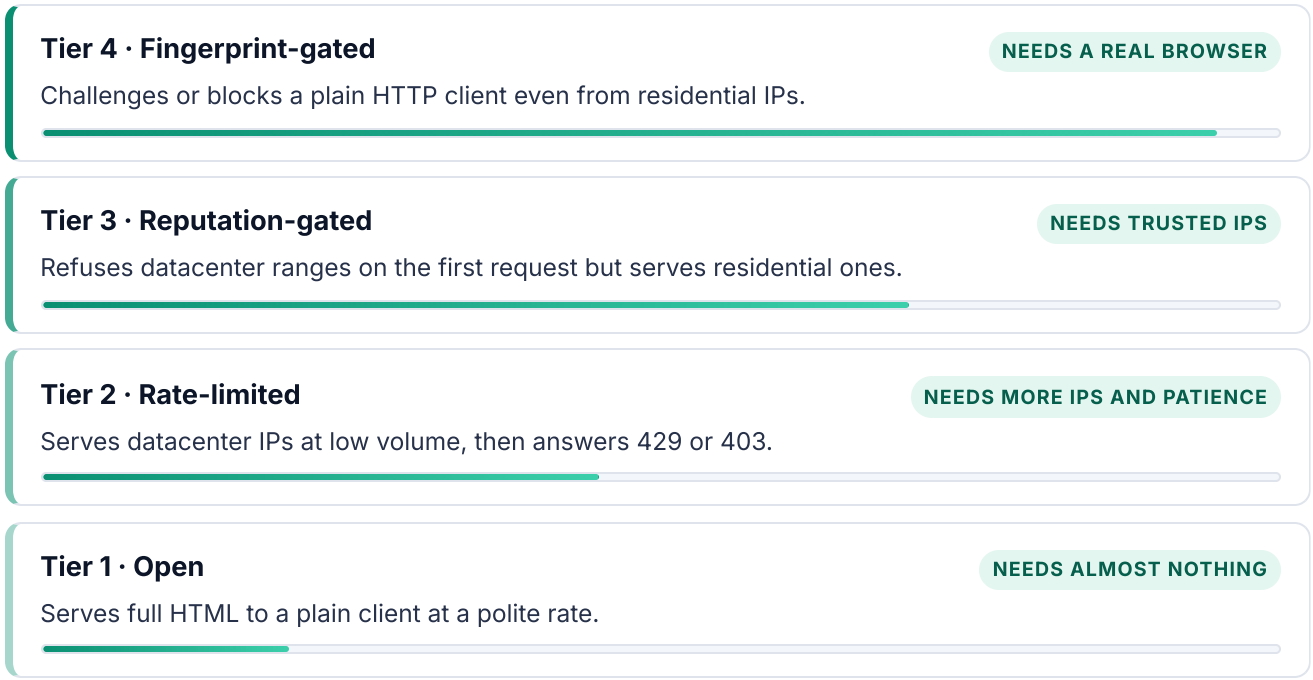
Sorts targets into four difficulty tiers by observable signs, matches a proxy type to each, and builds a routing ladder that starts cheap and escalates.

- Four target difficulty tiers
- Geo-targeting for localized content
- Rotation versus sticky sessions
- A routing ladder that escalates

A parts catalog on a small shop's own server answers plain datacenter requests all day. A search page on a large marketplace may refuse the same request before any HTML is built. Proxy choice starts with knowing which kind of target is on the list.

Sorting targets by what can be observed

Target difficulty does not need guessing. A short probe at a polite rate shows it. Fetch one page with a plain HTTP client from a datacenter IP, then from a residential IP, then in a real browser. Note where the full page first appears.



- Four target tiers, hardest first. The meter shows relative effort, not a measurement.

Tiers belong to endpoints, not to whole sites. Product pages may sit in tier 2 while search on the same domain sits in tier 4. Tiers also drift, so the probe is worth repeating whenever success rates move.

Which proxy type fits which tier

Each tier fails at a different detection layer, so each needs a different fix. The rule is to buy the cheapest type that clears the layer in question.

Tier	Start with	Why
1 · Open	Shared or rotating datacenter	Reputation is not checked, so the cheapest bytes win
2 · Rate-limited	A larger datacenter pool, or static ISP	More addresses keep each one under the limit
3 · Reputation-gated	Rotating residential or static ISP	Addresses are registered to consumer networks
4 · Fingerprint-gated	Residential plus a browser, or a scraper API	The client must change, not only the IP

- A starting proxy type for each target tier.

Datacenter proxies are fast and cheap, and their ASNs are easy to recognize. ISP proxies are hosted in datacenters but registered to consumer providers. They stay fixed, which suits sticky work. Residential proxies exit through home connections, so they are slower and less stable but hard to tell from visitors.

Mobile proxies exit through carrier networks, where many subscribers share one address. Sites tend to be cautious about refusing those addresses. Mobile is the most expensive type and fits mobile-only content, or tier 3 targets where residential still fails.

Proxy type	Billed by	Rough entry price
Rotating or shared datacenter	Traffic	Roughly $0.10 to $1.40 per GB
Dedicated datacenter	IP	Roughly $1 to $2.50 per IP per month
Static ISP	IP	Roughly $1 to $3.50 per IP per month
Rotating residential	Traffic	Roughly $1 to $5 per GB, less at volume
Rotating mobile	Traffic	Roughly $1 to $8.50 per GB

- Entry-level list prices in the ProxyLook directory. They change often.

These ranges are rough guidance only. Live figures are on the ProxyLook pricing page, and the cost calculator turns monthly traffic in GB into an estimated spend. Pool sizes on provider pages are vendor-stated and say little about success on one specific target.

Rotation per request or sticky sessions

With per-request rotation, every request leaves from a new exit IP. That fits independent URLs, such as 50,000 product pages where each response stands alone. No single address builds up a history.

A sticky session holds one exit IP for a window of time. Providers usually expose it through a session ID in the proxy username, in a form like `USERNAME-country-us-session-a1b2c3`. That syntax is illustrative. The exact parameters come from the provider's docs.

Sticky sessions fit paginated and stateful crawls. Some searches hand out a cursor token. Some listings keep their filters in a cookie. Both expect page 3 to come from the visitor who loaded page 2. A new IP mid-sequence can reset the state or draw a challenge.

> **TIP** **One visitor, one bundle**
>
> Treat a session ID, a cookie jar and a header set as one unit. Create them together and retire them together. Mixing an old cookie with a new IP produces the kind of contradiction that detection looks for.

Residential sticky sessions end when the provider's window closes or the home device goes offline. A paginated crawl should checkpoint its position and restart the sequence under a new session. Static ISP addresses hold for as long as they are rented.

Geo-targeting for localized content

Prices, currency, stock and search rankings change with the visitor's country, and sometimes with the city. A crawl for German prices needs German exits. Country targeting is set at the gateway, usually by another username parameter.

Three checks keep localized data honest. Verify the exit country with the proxy checker, because geolocation databases disagree. Send an `Accept-Language` header that matches the country. Confirm the currency or region marker in the response body. City-level targeting shrinks the usable pool, so it is worth using only when the data varies by city.

A routing ladder that starts cheap

A crawler with one proxy type overpays on easy targets or fails on hard ones. A ladder sends each domain through the cheapest route that works and moves up only on evidence.

1. **Start on datacenter.** Every new domain begins on the cheapest rung with a plain HTTP client.
2. **Widen the pool.** If 429s dominate, slow down first, then add addresses or move to ISP.
3. **Switch to residential.** If 403s arrive on the first request, datacenter ranges are being refused.
4. **Add a browser.** If residential IPs still fail with an HTTP client, the client is the problem.
5. **Hand off the remainder.** Send the hardest endpoints to a scraper API, covered in chapter 5.

The class below keeps a rung per domain. It moves a domain up one rung when the success share over a recent window drops under a floor. The floor of 0.8 is an example value to tune, not a recommendation.

PYTHON · STANDARD LIBRARY

```python
from collections import defaultdict, deque

LADDER = ("datacenter", "isp", "residential", "browser", "api")
WINDOW = 200   # recent results remembered per domain
FLOOR = 0.8    # example value: escalate below this success share


class Router:
    def __init__(self) -> None:
        self.rung: dict[str, int] = defaultdict(int)
        self.recent: dict[str, deque[bool]] = defaultdict(
            lambda: deque(maxlen=WINDOW)
        )

    def route(self, domain: str) -> str:
        return LADDER[self.rung[domain]]

    def record(self, domain: str, ok: bool) -> None:
        window = self.recent[domain]
        window.append(ok)
        if len(window) < WINDOW:
            return
        top = len(LADDER) - 1
        if sum(window) / WINDOW < FLOOR and self.rung[domain] < top:
            self.rung[domain] += 1
            window.clear()
```

Two habits make the ladder pay off. Store the rung with every result, so cost can be reported per route. Send a small sample back down a rung now and then, because defenses change in both directions.

Quick decisions

The table compresses the chapter into first moves. Each row assumes the probe from the first section has already been run.

If…	Then…
A plain client gets full pages from a datacenter IP	Stay on datacenter and spend the effort on rate limits
Requests succeed, then turn into 429s	Lower the rate, then widen the pool

If...	Then...
Datacenter IPs get 403 on the first request	Move that domain to residential or ISP
The crawl is paginated or holds state in cookies	Use a sticky session for each sequence
The data differs by country	Set the country and verify it in the response
Residential IPs fail with an HTTP client	Change the client before buying mobile

Common situations and the first move for each.

KEY TAKEAWAYS

- Probe each endpoint at a polite rate and sort it into one of four tiers.
- Buy the cheapest proxy type that clears the layer the target checks.
- Rotate per request for independent URLs and hold a sticky session for paginated or stateful sequences.
- Verify the exit country and match the language header when the content is localized.
- Route every domain through a ladder that starts on datacenter and escalates on evidence.

CHAPTER 03

Building a Resilient Client

Builds a small async httpx crawler step by step: proxy sessions, honest headers, status-aware retries with jitter, soft-block validation and a bounded worker pool.

- Timeouts, HTTP/2 and consistent headers
- Rotating the session on failure
- Treating 403, 429 and 5xx differently
- Validating bodies, bounding concurrency

A first scraper is often a loop around `get()` with a bare `except`. At the first 429 it retries at once from the same IP and turns a rate limit into a ban. This chapter replaces that loop with a client that knows why a request failed.

The base client

The six code blocks in this chapter form one file. Pasted in order, they make a small async crawler. The first block defines the headers and a factory that builds a client for one proxy session.

PYTHON · HTTPX

```python
import httpx

GATEWAY = "gate.example-proxy.com:7000"
HEADERS = {
    "User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:140.0) "
    "Gecko/20100101 Firefox/140.0",
    "Accept": "text/html,application/xhtml+xml,"
    "application/xml;q=0.9,*/*;q=0.8",
    "Accept-Language": "en-US,en;q=0.5",
    "Upgrade-Insecure-Requests": "1",
    "Sec-Fetch-Dest": "document",
    "Sec-Fetch-Mode": "navigate",
    "Sec-Fetch-Site": "none",
    "Sec-Fetch-User": "?1",
}


def make_client(session_id: str) -> httpx.AsyncClient:
    # Illustrative username syntax. The provider's docs define it.
    user = f"USERNAME-country-us-session-{session_id}"
    return httpx.AsyncClient(
        proxy=f"http://{user}:PASSWORD@{GATEWAY}",
        headers=HEADERS,
        http2=True,
        timeout=httpx.Timeout(20.0, connect=10.0),
        follow_redirects=True,
    )
```

Four settings carry most of the weight.

- **Timeouts.** The httpx default is 5 seconds per phase, which is tight for a residential exit. Each timeout covers one phase, such as connecting or waiting for the next chunk. It is not a deadline for the whole request. That deadline is added in the retry loop.
- **HTTP/2.** It needs the extra package, installed with `pip install "httpx[http2]"`. Browsers use HTTP/2 or HTTP/3 on most large sites, so an HTTP/1.1 client stands out. The `http_version` attribute on a response shows what was negotiated.
- **Redirects.** httpx does not follow them by default. The factory turns that on, and the validator later checks where each response ended up.
- **TLS verification.** It is on by default and nothing here turns it off. A password with special characters must be percent-encoded in the proxy URL.

Headers that agree with each other

The default `User-Agent` is `python-httpx` plus a version. It is honest, and many sites refuse it on sight. There are two defensible replacements. One is a declared bot string with a contact URL, which suits tier 1 targets and sites that welcome crawlers. The other is a complete header set copied from one real browser release.

The example follows Firefox because Firefox sends a short set. Chrome adds a family of `sec-ch-ua` headers that would have to match as well. The `Sec-Fetch` values describe a page opened directly. A browser changes them, and adds a `Referer`, when a link is followed. The version in the example will age, so copy the set from a current release and refresh it on a schedule. The user-agent parser shows what a string claims. `Accept-Encoding` is left to httpx, which advertises only the encodings it can decode.

> **WATCH OUT** **Consistent at layer 4 only**
>
> These headers agree with each other, but the TLS and HTTP/2 fingerprints still belong to a Python library. httpx also merges caller headers into its own defaults, so the wire order is not fully controlled. That is enough for tiers 1 to 3, while tier 4 needs the browser from chapter 4.

One identity per session ID

In httpx the proxy belongs to the client, not to the request. A new session ID therefore means a new client. That is useful, because a new client also starts with an empty cookie jar and no open connections. Nothing from a flagged session follows the scraper to its next exit IP.

PYTHON · HTTPX

```python
import secrets


class Identity:
    """One session ID: one exit IP, one cookie jar, one pool."""

    def __init__(self) -> None:
        self.client = make_client(secrets.token_hex(4))

    async def rotate(self) -> None:
        # A new client drops old cookies and connections too.
        await self.client.aclose()
        self.client = make_client(secrets.token_hex(4))

    async def close(self) -> None:
        await self.client.aclose()
```

How long the exit IP holds for one session ID depends on the provider's sticky session rules. A call to an IP echo endpoint through each client confirms that two session IDs produce two different exits, and the proxy checker shows the country and network of each one.

Different failures, different responses

A status code says which layer objected, so each one deserves its own reaction. Retrying a 403 from the same identity repeats the request that was just refused. Rotating on a 429 dodges a limit the site set on purpose.

Result	Usual meaning	Reaction
200	Served, or a soft block	Validate the body before storing
403	Identity refused	New session ID, a few tries at most
429	Too fast	Wait as told, keep the session, slow the domain
500 to 504	Origin or gateway trouble	Back off and retry
404 or 410	Page is gone	Record it and do not retry
Timeout or reset	Dead exit or a stall	New session ID, then retry

- What each result usually means and how the client reacts.

The next block turns that table into code. The wait uses exponential backoff with full jitter: a random delay below a ceiling that doubles on every attempt. Jitter keeps a pool of workers from retrying in step.

PYTHON · HTTPX

```python
import random
from enum import Enum


class Verdict(Enum):
    OK = "ok"
    BACKOFF = "backoff"   # same identity, wait longer
    ROTATE = "rotate"     # new session ID, then wait
    FATAL = "fatal"       # retrying cannot help


def classify(status: int) -> Verdict:
    if 200 <= status < 300:
        return Verdict.OK
    if status == 403:
        return Verdict.ROTATE
    if status == 429 or status >= 500:
        return Verdict.BACKOFF
    return Verdict.FATAL


def wait_for(attempt: int, resp: httpx.Response | None) -> float:
    # Full jitter: a random wait below an exponential ceiling.
    wait = random.uniform(0, min(60.0, 2.0 ** attempt))
    hint = resp.headers.get("retry-after", "") if resp is not None else ""
    return max(wait, float(hint)) if hint.isdigit() else wait
```

`Retry-After` can also carry an HTTP date, which this function ignores. Production code should cap the hint and requeue the job instead of sleeping for an hour. A 502 or 503 may also come from the proxy gateway instead of the origin. A common cause is that no exit is free in the requested country. The provider's docs list those gateway errors.

The retry loop



- Each attempt is classified before the client decides to rotate, wait or give up.

The loop wraps each attempt in `asyncio.timeout`, so a response that drips in slowly cannot hold a worker forever. Network errors and missed deadlines count as a dead exit and trigger a rotation.

PYTHON · HTTPX

```python
import asyncio

MAX_ATTEMPTS = 4
DEADLINE = 45  # seconds for one whole attempt


async def fetch(ident: Identity, url: str) -> httpx.Response | None:
    for attempt in range(MAX_ATTEMPTS):
        resp = None
        try:
            async with asyncio.timeout(DEADLINE):
                resp = await ident.client.get(url)
            verdict = classify(resp.status_code)
        except (httpx.RequestError, TimeoutError):
            verdict = Verdict.ROTATE
        if verdict is Verdict.OK and not looks_valid(resp):
            verdict = Verdict.ROTATE
        if verdict is Verdict.OK:
            return resp
        if verdict is Verdict.FATAL:
            return None
        if verdict is Verdict.ROTATE:
            await ident.rotate()
        await asyncio.sleep(wait_for(attempt, resp))
    return None
```

The attempt cap matters more than any other line. Four refusals from four exit IPs are not an IP problem. Looping harder only burns traffic, so the job should stop and move up the routing ladder from chapter 2. A rejected proxy password surfaces as `httpx.ProxyError` on HTTPS targets and fails all four attempts quickly. Logging the exception text makes that obvious.

Catching soft blocks in the body

A 200 proves that something was returned, not that it was the page. The validator checks three things. The final URL must not be an interstitial. The size must be plausible. A marker found on every good page must be present.

PYTHON · HTTPX

```python
REQUIRED = 'id="product-title"'    # found on every good page
MIN_BYTES = 5_000                  # well below any real page
CHALLENGE_HINTS = ("captcha", "challenge", "blocked")


def looks_valid(resp: httpx.Response) -> bool:
    path = resp.url.path.lower()
    if any(hint in path for hint in CHALLENGE_HINTS):
        return False   # redirected to an interstitial
    if "text/html" not in resp.headers.get("content-type", ""):
        return False
    if len(resp.content) < MIN_BYTES:
        return False
    return REQUIRED in resp.text
```

All three constants are placeholders to set per target. The positive check does most of the work. Searching the body for a word like "captcha" misfires on healthy pages that embed a CAPTCHA widget in a form. For a JSON endpoint, the same idea applies to keys: parse the body and require the expected fields.

A bounded worker pool

A semaphore caps the number of requests in flight. Each task owns its identity, so a rotation in one task never closes a client that another task is using.

PYTHON · ASYNCIO

```python
Result = tuple[str, str | None]


async def scrape(url: str, gate: asyncio.Semaphore) -> Result:
    async with gate:
        ident = Identity()
        try:
            resp = await fetch(ident, url)
        finally:
            await ident.close()
    return url, resp.text if resp is not None else None


async def crawl(urls: list[str], limit: int = 8) -> list[Result]:
    gate = asyncio.Semaphore(limit)
    return await asyncio.gather(*(scrape(u, gate) for u in urls))


if __name__ == "__main__":
    for url, body in asyncio.run(crawl(["https://example.com/"])):
        print(url, "ok" if body else "failed")
```

Run against `example.com`, the script prints "failed", because the placeholder marker is absent. That is the validator doing its job. One identity per URL matches per-request rotation. For a paginated sequence, pass the whole list to one task and reuse its identity.

Two limits remain. `gather` creates every task up front, which is fine for thousands of URLs and wasteful for millions. A concurrency cap is also not a rate limit: eight workers can still hit one small site eight at a time. Chapter 6 adds a queue and per-domain pacing.

KEY TAKEAWAYS

- Set explicit timeouts and add a whole-attempt deadline, because httpx timeouts cover single phases.
- Send one complete, current header set, and accept that it does not change the TLS fingerprint.
- Rotate the session on 403 and on dead exits, wait on 429 and 5xx, and never retry a 404.
- Cap attempts, because repeated refusals across fresh IPs mean the problem is not the IP.
- Validate every 200 against a marker that exists only on a good page.
- Bound concurrency with a semaphore and give each task its own identity.

CHAPTER 04

Headless Browsers Through Proxies

Shows when a real browser earns its cost, then sets up Playwright with a proxy per context, matching locale and time zone, and resource blocking.

- When a browser is waste
- Locale and time zone consistency
- A proxy per browser context
- Blocking heavy resources

A product page shows a price in the browser. The same URL fetched with httpx returns an empty `<div id="root">` and 300 KB of JavaScript. That is the honest case for a headless browser, and it comes up less often than it first appears.

When a browser is needed, and when it is waste

A browser is the most expensive way to fetch a page, so it should be the last tool tried. Three checks come first, in this order.

1. **Read the raw HTML.** Fetch the page with an HTTP client and search for a known value, such as a price. Server-rendered pages already contain it.
2. **Look for embedded JSON.** Many JavaScript sites ship their data inside a `script` tag, as JSON-LD or as framework state. A parser can read it without running anything.
3. **Watch the network tab.** If the page fills itself in from a public JSON endpoint, the HTTP client from chapter 3 can often call it directly.

The third check has a boundary. An endpoint that wants a login token or signed parameters is not public. It falls outside the scope of this book.

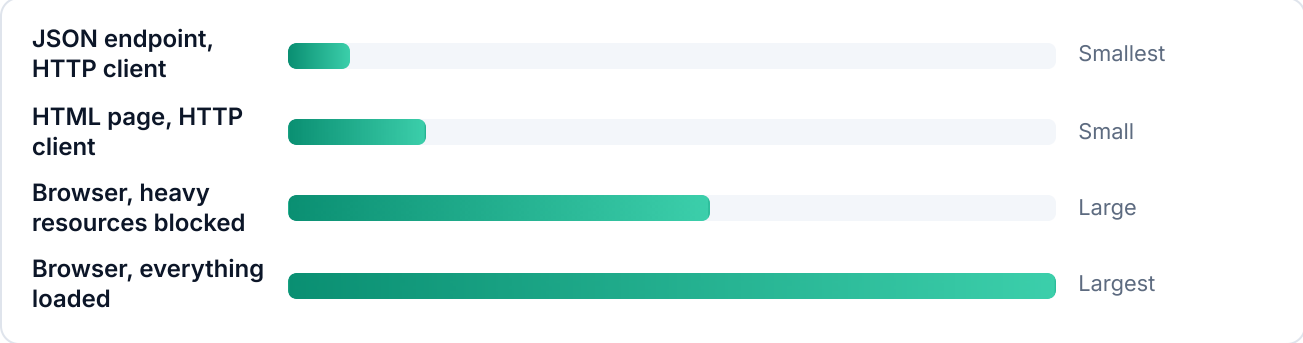
What the probe shows	Browser?	Cheaper route
Value is in the raw HTML	No	HTTP client and an HTML parser
Value is in a JSON blob in the page	No	HTTP client, then parse the blob
Page calls a public JSON endpoint	Rarely	Request the endpoint itself
Content appears only after scripts run	Yes	None
Site runs a script check before serving	Yes	None, or a scraper API
Content loads on scroll or click	Usually	The endpoint behind the scroll, if public

Signs that a browser is needed, and what to try first when it is not.

The last three rows are tier 4 targets from chapter 2. For those, a real browser sends real signals at layers 3 to 5: a browser handshake, browser headers and a JavaScript runtime.

What a browser costs

An HTTP client downloads one document. A browser downloads that document and then everything it references: scripts, stylesheets, fonts, images, video and third-party trackers. On a per-GB plan, every one of those bytes is billed, whether or not the scraper reads it.



Relative proxy traffic per page for four ways of fetching it. The widths show order, not measurements.

The real ratio depends on the site, so measure it on the target before budgeting. Multiply bytes per page by pages per month, then put the total in GB into the cost calculator. Traffic is not the only cost. A machine that runs hundreds of concurrent HTTP requests may hold only a handful of browser pages in memory. Each page also waits on rendering.

A proxy per browser context

Playwright can launch one browser and open many contexts inside it. A context is an isolated profile with its own cookies, storage and cache. It can also have its own proxy. That makes a context the browser version of the identity from chapter 3: one session ID, one exit IP, one cookie jar.

PYTHON · PLAYWRIGHT

```python
import asyncio

from playwright.async_api import Browser, BrowserContext, async_playwright

GATEWAY = "http://gate.example-proxy.com:7000"
GEO = {
    "us": ("en-US", "America/New_York"),
    "de": ("de-DE", "Europe/Berlin"),
}


async def new_identity(
    browser: Browser, country: str, session: str
) -> BrowserContext:
    locale, zone = GEO[country]
    context = await browser.new_context(
        proxy={
            "server": GATEWAY,
            "username": f"USERNAME-country-{country}-session-{session}",
            "password": "PASSWORD",
        },
        locale=locale,
        timezone_id=zone,
    )
    context.set_default_timeout(30_000)
    return context
```

The username syntax is illustrative, and the provider's docs define the real parameters. Playwright takes the credentials as separate fields, not inside the server URL. The default timeout is set in milliseconds and covers navigation and other calls that wait. Older Playwright releases needed a placeholder proxy at launch before Chromium accepted a proxy per context. The documentation for the installed version says whether that still applies.

A proxy can also be set once at launch with the same dictionary. That fits a crawl where every page shares one exit. Per-context proxies fit rotation: ten contexts in one browser process give ten exits without starting ten browsers. Each open page still holds memory, so the semaphore pattern from chapter 3 applies here as well.

Keeping locale and time zone consistent

A scraper on a cloud server usually runs in UTC with an English locale. Put a German residential exit in front of it and the page sees a visitor in Germany whose clock and language say otherwise. That is a layer 5 contradiction, and a script can read it in two lines.

The `locale` option sets `navigator.language` and the `Accept-Language` header together. The `timezone_id` option sets the zone that JavaScript dates report. Both should follow the proxy's country, which is why the example keeps them in one table beside the country code.

> **WATCH OUT** **Three leaks to check**
>
> Large countries span several time zones, so a matching zone may need city or state targeting. WebRTC can reveal the server's own address past an HTTP proxy, which the WebRTC leak test shows. Headless builds also differ from desktop browsers in ways a script can observe.

No setting removes every difference between automation and a person at a keyboard. Projects such as Camoufox, an open-source Firefox fork built for Playwright automation, narrow some of them. None can promise to pass a given detection product, and a crawler should be built to notice failure instead of assuming success.

Blocking heavy resources

Route interception lets the scraper cancel requests before they leave the machine. Images, fonts and media are the usual candidates. They make up much of a page's weight and rarely contain the data.

PYTHON · PLAYWRIGHT

```python
import random
import secrets

from playwright.async_api import Route

HEAVY = {"image", "media", "font"}


async def skip_heavy(route: Route) -> None:
    if route.request.resource_type in HEAVY:
        await route.abort()
    else:
        await route.continue_()
```

Three cautions apply. Blocking scripts or stylesheets breaks many pages, so those types stay out of the set. Playwright's documentation notes that enabling routing disables the HTTP cache, so assets that are not blocked download again on every navigation. A browser that never loads an image is also an unusual visitor. Blocking trades bytes for a less ordinary profile, and the right balance depends on the target.

Reusing a context for sticky work

A paginated listing should run inside one context from first page to last. Cookies set on page 1 are still there on page 2, and the session ID keeps the exit IP steady. The block below attaches the route handler and walks the pages with a pause. It stops at the first response that is not a 200.

PYTHON · PLAYWRIGHT

```python
async def crawl_list(browser: Browser, base: str, pages: int) -> list[str]:
    context = await new_identity(browser, "de", secrets.token_hex(4))
    await context.route("**/*", skip_heavy)
    page = await context.new_page()
    bodies = []
    try:
        for number in range(1, pages + 1):
            url = f"{base}?page={number}"
            resp = await page.goto(url, wait_until="domcontentloaded")
            if resp is None or resp.status != 200:
                break
            bodies.append(await page.content())
            await page.wait_for_timeout(random.uniform(1500, 4000))
    finally:
        await context.close()
    return bodies


async def main() -> None:
    async with async_playwright() as pw:
        browser = await pw.chromium.launch(headless=True)
        html = await crawl_list(browser, "https://example.com/list", 3)
        print([len(body) for body in html])
        await browser.close()


asyncio.run(main())
```

The three blocks form one file. Closing the context in `finally` matters, because a leaked context keeps its memory and its proxy connections. A navigation timeout or a proxy failure raises Playwright's `Error`. The caller should catch it and requeue the sequence. The next sequence gets a new context and a new session ID, so no state crosses between visitors. The bodies still need the validation from chapter 3, since a browser can receive a soft block as easily as an HTTP client.

KEY TAKEAWAYS

- Check the raw HTML, embedded JSON and public endpoints before reaching for a browser.
- A browser multiplies proxy traffic and memory use, so reserve it for targets that need script execution.
- Give each browser context its own proxy session and treat it as one visitor.
- Set locale and time zone from the proxy's country so the runtime agrees with the IP.
- Block images, fonts and media to save bytes, and measure the effect on the target.
- Hold one context for a paginated sequence, then close it and start fresh.

CHAPTER 05

Scraper APIs and Unblockers

Explains what scraper APIs and unblockers do, how per-request pricing changes the cost math, and where a managed service belongs in a routing ladder.

- What managed scraping products do
- The API as a top rung
- Per-request versus per-GB math
- Lock-in and observability trade-offs

A team spends three weeks tuning a headless crawler for one marketplace, and the success rate still swings from day to day. A scraper API offers to take the URL and return the HTML for a fee per successful request. Whether that is a bargain is a matter of arithmetic.

What these products do

A managed scraping product bundles the work of chapters 2 to 4 behind one endpoint. The vendor picks the proxy, builds the headers, runs a browser when needed, retries failures and returns the result. The products come in three shapes.

Scraper API

An HTTP endpoint that takes a target URL and options such as country or rendering. It returns the page HTML.

Unblocker

The same service exposed as a proxy address. An existing client points at it like any other proxy.

Structured endpoint

An API for one site type, such as search results or product pages. It returns parsed JSON instead of HTML.

Proxy-style unblockers usually decrypt and re-encrypt traffic so they can rewrite requests. The client then has to trust the vendor's CA certificate. Install that certificate as the vendor documents. Switching verification off hides real TLS errors along with the expected one.

Two limits apply to every product in this category. None can promise success on every target, so a trial on the real URL list is the only useful test. A vendor also does not change the buyer's legal position. Responsibility for what is collected stays with the team that asked for it.

Build or buy

The choice turns on four things: target tier, volume, engineering time and the need for control. Tier 1 and tier 2 targets rarely justify a managed service, because a plain client already succeeds. Tier 4 targets are where the vendor's pooled effort is hard to match. A team of two may not have a week each month to spend on fingerprint upkeep.

Control cuts the other way. A self-built client decides its own headers, cookies, session length and pacing. An API exposes only the options its vendor chose to offer. Crawls that depend on a long sticky sequence need that support confirmed before any contract is signed.

How per-request pricing changes the math

Per-GB proxy billing charges for every byte, including failed attempts and blocked pages. Per-request billing usually charges only for successes, so failures and page weight become the vendor's problem. Rendering often carries a higher price or a credit multiplier, so the vendor's own table needs a careful read.

The fair comparison is cost per 1,000 valid pages, with engineering time included. For a self-built route, the function below does the sum.

PYTHON · STANDARD LIBRARY

```python
def cost_per_1000(
    mb_per_attempt: float,
    success_share: float,
    usd_per_gb: float,
    upkeep_usd: float,
    pages: int,
) -> float:
    gb = pages * mb_per_attempt / success_share / 1000
    total = gb * usd_per_gb + upkeep_usd
    return round(total / pages * 1000, 2)


# Invented round numbers, matching the table below.
print(cost_per_1000(2.0, 0.5, 3.0, 500.0, 100_000))   # 17.0
print(cost_per_1000(0.2, 1.0, 0.5, 50.0, 100_000))    # 0.6
```

The table runs that sum for 100,000 valid pages on a hard target and on an easy one. The hard target needs a browser at 2 MB per attempt, and half the attempts fail. The easy target needs 0.2 MB per attempt and almost never fails. An hour of engineering is priced at $50.

Target	Route	Inputs	Total	Per 1,000
Hard	Own browser, residential	400 GB at $3, plus 10 hours	$1,700	$17.00
Hard	Scraper API	$5 per 1,000 requests, plus 2 hours	$600	$6.00
Easy	Own client, datacenter	20 GB at $0.50, plus 1 hour	$60	$0.60
Easy	Scraper API	$2 per 1,000 requests, plus 1 hour	$250	$2.50

- Illustration only. Every price, page size, hour count and success share is an invented round number.

EXAMPLE **Read the shape, not the figures**

None of these numbers is a market price or a measurement, and only the pattern matters. On the hard target the API wins, because failed attempts and upkeep dominate. On the easy target it loses by a wide margin.

Rerun the sum with real inputs before deciding. Measured bytes per attempt and success share come from the crawler's own logs. Live prices are on the proxy API providers page and the pricing page. The cost calculator handles the per-GB side.

Where an API fits in the ladder

The comparison points to a mixed setup. Cheap routes take everything they can handle, and the API takes only what they cannot. In the routing ladder from chapter 2, that makes the API the top rung.



A routing ladder with a managed API as the last rung.

A thin interface keeps the rungs interchangeable. Each rung implements one method and returns the same record, whether it wraps httpx, Playwright or a vendor's API.

PYTHON · STANDARD LIBRARY

```python
from dataclasses import dataclass
from typing import Protocol


@dataclass
class Page:
    url: str
    body: str
    route: str   # the rung that produced it, for cost reports


class Fetcher(Protocol):
    name: str

    async def fetch(self, url: str) -> Page | None: ...


async def fetch_with_ladder(url: str, rungs: list[Fetcher]) -> Page | None:
    for rung in rungs:
        page = await rung.fetch(url)
        if page is not None:
            return page
    return None
```

In practice the router from chapter 2 remembers which rung works for each domain, so most URLs skip straight to it. The fall-through loop handles the exceptions. For a small team with a handful of hard targets, a ladder with one rung, the API, is a reasonable place to start.

Lock-in and observability

Every vendor has its own parameters, response envelope, error codes and credit rules. Code written against them directly is expensive to move. The interface above limits that cost to one adapter class per vendor. Storing raw HTML and keeping parsers in-house helps as well, since a vendor's parsed JSON has a schema no other vendor shares.

Observability is the larger trade. A self-built client can log every header, status and exit IP, so a block can be traced to a layer. An API reports that a request failed and little else. When its success rate drifts, the buyer can only open a ticket.

- [ ] Log the vendor's status, the final URL, the response time and whether the request was billed.
- [ ] Find out how the vendor defines success, since a soft block with status 200 may still be charged.
- [ ] Keep the body validation from chapter 3 in front of every stored page.
- [ ] Check the plan's concurrency cap against the crawl's peak rate.
- [ ] Keep a second vendor tested behind the same interface.

Decision table

The table condenses the chapter into first moves. Once the route is clear, the ProxyLook compare page sets providers side by side.

If...	Then...
Targets are tier 1 or tier 2	Build, because an API adds cost without adding success
A few tier 4 endpoints hold the value	Send only those endpoints to an API
Nobody can own crawler upkeep	Buy, and keep validation in-house
Volume on one hard target is very high	Rerun the cost sum, since a self-built route may win
Sessions, cookies or headers need exact control	Confirm the API supports it, or build
URLs or payloads are sensitive	Read the vendor's data handling terms first

Build or buy, by situation.

KEY TAKEAWAYS

- Scraper APIs and unblockers bundle proxies, headers, browsers and retries behind one endpoint.
- Compare routes by cost per 1,000 valid pages, with failed attempts and engineering time counted.
- Per-request pricing tends to win on hard targets and lose on easy ones.
- Place the API on the top rung of the ladder so it handles only what cheaper routes cannot.
- Limit lock-in with a thin adapter, stored raw HTML and in-house parsers.
- Keep logging and body validation in place, because an API hides the reason for a failure.

CHAPTER 06

Running in Production

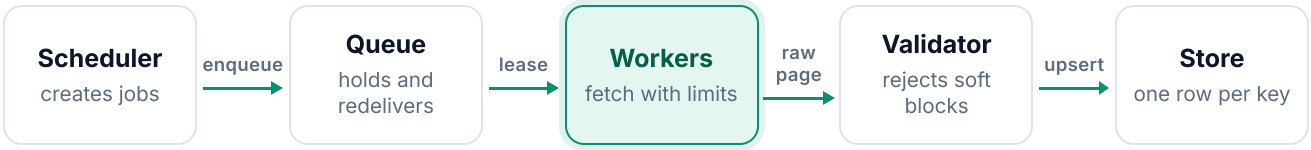
Turns a working scraper into a scheduled service: idempotent jobs, per-domain pacing, the four metrics that matter, data-quality checks, useful logs and a launch checklist.

- Queues and idempotent jobs
- Per-domain rate limits
- Metrics, budgets and quality checks
- Etiquette and a launch checklist

A crawler runs cleanly on a laptop for an afternoon and gets scheduled nightly. On the ninth night the process is killed halfway, restarts from the top and writes every row a second time. Production scraping is mostly about what happens when work stops, fails and repeats.

Queues and idempotent jobs

A production crawl is a pipeline, not a loop. A scheduler puts jobs on a queue. Workers take jobs, fetch pages and hand results to a validator. Only validated records reach the store. A job carries the URL, the route, an attempt count and the run date.



The production pipeline. Each stage can fail and restart without corrupting the next.

Queues redeliver. A worker that dies mid-job leaves that job to be handed out again, so every job must be safe to run twice. The usual method is a deterministic key and an upsert. The key below hashes a normalized URL with the run date, so two spellings of one URL collapse into one row.

PYTHON · STANDARD LIBRARY

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


def job_key(url: str, run_date: str) -> str:
    parts = urlsplit(url)
    pairs = parse_qsl(parts.query, keep_blank_values=True)
    query = urlencode(sorted(pairs))
    canonical = urlunsplit(
        (parts.scheme, parts.netloc.lower(), parts.path, query, "")
    )
    raw = f"{run_date}|{canonical}".encode()
    return hashlib.sha256(raw).hexdigest()


print(job_key("https://Example.com/p?b=2&a=1", "2026-01-15"))
print(job_key("https://example.com/p?a=1&b=2", "2026-01-15"))
```

Both lines print the same digest. Two more habits pay off early. Jobs that exhaust their retries go to a dead-letter queue for inspection instead of vanishing. Raw pages are stored compressed beside the parsed records, so a parser fix does not require a second crawl.

Per-domain rate limits

The semaphore in chapter 3 caps total concurrency. It does not stop every worker from landing on one small site at once. Pacing has to be kept per domain, whatever the number of exit IPs. A rotating pool hides the crawler's volume from per-IP counters, but the origin server still carries the full load.

PYTHON · ASYNCIO

```python
import asyncio
import time
from collections import defaultdict


class DomainLimiter:
    """Space out requests to each domain by a fixed interval."""

    def __init__(self, interval: float = 1.0) -> None:
        self.interval = interval
        self._locks: dict[str, asyncio.Lock] = defaultdict(asyncio.Lock)
        self._next: dict[str, float] = defaultdict(float)

    async def wait(self, domain: str) -> None:
        async with self._locks[domain]:
            now = time.monotonic()
            delay = self._next[domain] - now
            if delay > 0:
                await asyncio.sleep(delay)
            start = max(now, self._next[domain])
            self._next[domain] = start + self.interval
```

A worker calls `await limiter.wait(domain)` before each request. A sensible refinement is an adaptive interval. Double it for a domain after a 429 or 503, then shrink it slowly once responses recover. If `robots.txt` states a crawl delay, that value is the floor.

Metrics, budgets and alerts

Four numbers describe the health of a crawl. Each should be tracked per domain and per route, because an average across domains hides the one that broke.

Metric	Definition	What a shift suggests
Success rate	Valid records divided by attempts	A fall means a new block or a layout change
Cost per 1,000 successful records	Proxy, API and compute spend over valid records	A rise means more retries or heavier pages
p95 latency	Time under which 95% of attempts finish	A rise means throttling or slow exits

Metric	Definition	What a shift suggests
Bytes per record	Proxy traffic divided by valid records	A rise means waste; a sharp fall can mean soft blocks

The four metrics worth a dashboard, and how to read a change in each.

"Valid" means the record passed validation, not that the status was 200. Counting bare 200s makes a soft block look like a good day.

Budgets need their own alarm. Blocked pages still cost bytes on a per-GB plan, so a retry storm overnight can spend a month of traffic. Count bytes in the client, set a daily budget per domain, and alert at fixed shares of it. At the full budget, pause that domain's queue and leave the decision to a person. The provider's dashboard or usage API gives a second reading to reconcile against. The cost calculator helps set the first budget, and the pricing page has current rates.

Data-quality checks that catch soft blocks

Per-page validation catches a challenge page. It can miss a page that is well formed and wrong. Batch checks compare each run with earlier ones.

- **Record count.** Compare with the last several runs. A listing that returned 4,000 items and now returns 900 needs a look.
- **Null rate per field.** A price field that is suddenly empty in a third of rows points to a degraded page.
- **Duplicate ratio.** Many URLs returning one identical body suggests a cached or decoy response.
- **Value sanity.** Prices above zero, the expected currency, dates in range.
- **Canary URLs.** A few pages with stable, known values, checked on every run.

A batch that fails a check goes to quarantine, not to the published dataset. Stale data that is known to be stale beats fresh data that is silently wrong.

Logging enough to debug a block

When success drops, the symptom table in chapter 1 is only useful if the evidence was kept. One structured log line per attempt is enough.

What was asked

Timestamp, job key, URL, attempt number, route, proxy type, country and session ID.

What came back

Status, final URL, body size, elapsed time, and the `server`, `content-type` and `retry-after` headers.

What was decided

The verdict, the validator's reason for any rejection, and the wait before the next attempt.

Samples

A few rejected bodies per domain per hour, saved whole. One real block page explains more than a thousand status codes.

With those lines, yesterday's failures can be grouped by status, route and country in one query. Session IDs belong in logs. Proxy passwords and full proxy URLs do not.

Etiquette, law and launch

A site's `robots.txt` states what its operator asks of crawlers. Python's standard library can read it, and the fetch below keeps its timeout and TLS checks.

PYTHON · HTTPX

```python
from urllib.robotparser import RobotFileParser

import httpx

AGENT = "example-crawler"


def load_robots(origin: str) -> RobotFileParser:
    parser = RobotFileParser()
    resp = httpx.get(
        f"{origin}/robots.txt", timeout=10.0, follow_redirects=True
    )
    if resp.status_code == 200:
        parser.parse(resp.text.splitlines())
    elif resp.status_code in (401, 403):
        parser.disallow_all = True
    else:
        parser.allow_all = True
    return parser


robots = load_robots("https://example.com")
print(robots.can_fetch(AGENT, "https://example.com/products"))
print(robots.crawl_delay(AGENT))
```

Good manners go further than the file. Crawl small sites off-peak. Back off at the first 429 or 503. Skip pages that have not changed, using sitemaps or conditional requests where the site supports them. Take only the fields the project needs. Where a declared user agent is practical, include a contact address. An operator can then write to the crawler's owner instead of blocking a whole range.

WATCH OUT **Not legal advice**

Public pages, personal data, content behind a login and a site's terms of service raise separate legal questions. Personal data carries duties even when it is publicly visible. This book is not legal advice, and a lawyer should review any crawl with real exposure.

The launch checklist gathers the chapter into eight lines. For the proxy side of the setup, the ProxyLook web scraping page lists providers aimed at this work.

- [ ] Every job has a deterministic key and writes with an upsert.
- [ ] Failed jobs land in a dead-letter queue.

- [ ] Each domain has its own rate limit and daily traffic budget.
- [ ] Success rate, cost per 1,000 records, p95 latency and bytes per record are on a dashboard.
- [ ] Batch quality checks run before data is published.
- [ ] Each attempt writes one structured log line with no credentials in it.
- [ ] `robots.txt` is read, and its crawl delay is respected.
- [ ] Someone has reviewed the data fields for personal data and the target's terms.

KEY TAKEAWAYS

- Make every job safe to run twice with a deterministic key and an upsert.
- Limit request rate per domain, because rotation hides load from counters but not from the origin.
- Track success rate, cost per 1,000 successful records, p95 latency and bytes per record for each domain.
- Count only validated records as successes, and quarantine batches that fail quality checks.
- Log one structured line per attempt and keep sample block pages.
- Read robots.txt, pace politely, and treat personal data and terms of service as legal questions.

APPENDIX

Appendix

Reference pages for the crawl desk: a status code table, a pre-flight checklist, a debugging order, a glossary and links to ProxyLook tools.

- HTTP status quick reference
- Pre-flight checklist
- Debugging order by layer
- Glossary and further reading

A crawl that returned 40,000 records yesterday returns 6,000 today, and the dashboard is red. These pages are for that moment. They condense the book into tables and lists that can be read in a minute.

HTTP status quick reference

The table gives the usual meaning of each result for a scraper and the first reaction. Chapter 3 turns it into code.

Result	Usual meaning	First reaction
200	Served, or a soft block	Validate the body before storing
301, 302	Moved, or sent to a challenge or login	Check the final URL
304	Unchanged since the last fetch	Keep the stored copy
403	Identity refused	New session; escalate if it repeats
404, 410	Page is gone	Record it; do not retry
407	Proxy rejected the credentials	Fix the configuration
429	Rate limit reached	Wait as told; slow the domain
500	Origin error	Back off and retry
502, 503, 504	Overloaded origin, gateway fault or challenge	Back off; inspect the body

Result	Usual meaning	First reaction
Timeout or reset	Dead exit or a deliberate stall	New session, then retry

Common results, what each usually means for a scraper, and the first reaction.

Pre-flight checklist

Run through this list before pointing a crawler at a new target.

- [ ] Confirm the data is public, and review personal data and the site's terms.
- [ ] Read `robots.txt` and note any crawl delay.
- [ ] Probe one page with an HTTP client, then a browser, and assign a tier.
- [ ] Look for the value in raw HTML or embedded JSON before using a browser.
- [ ] Pick the cheapest proxy type that clears the tier.
- [ ] Verify the exit country and match the language header.
- [ ] Set timeouts, an attempt cap and a per-domain rate limit.
- [ ] Define the validation marker and minimum size for a good page.
- [ ] Set a daily traffic budget with an alert.
- [ ] Check that logs record status, final URL, route and session ID.

Debugging order

When success drops, work from the cheapest check to the most expensive. The order follows the detection layers from chapter 1, with configuration first and data last.

1 · Configuration	Credentials, gateway address, country parameter, remaining balance.
2 · Exit IP	Type, country and network of the address the target saw.
3 · Rate	Requests per domain per minute, and any Retry-After header.
4 · Client signature	HTTP version and headers compared with a current browser.
5 · Browser runtime	Locale, time zone, automation flags, blocked resources.
6 · Session behavior	Pacing, page order and session length.
7 · Data	Body validation and batch checks for soft blocks.

- The order in which to check a failing crawl, cheapest first.

> **TIP** **Change one thing at a time**
>
> Swap the proxy type, or the client, or the rate, and rerun the same ten URLs. Changing two at once fixes the crawl without showing which layer was at fault. The next block then starts from zero.

Glossary

Short definitions of terms used in this book. The ProxyLook glossary covers proxy vocabulary more widely.

Term	Meaning
ASN	The number of the network that owns an IP range
Backoff	Waiting longer after each failed attempt
Browser context	An isolated browser profile with its own cookies and proxy
Challenge page	An interstitial asking for proof of a browser or person
Dead-letter queue	A holding area for jobs that used up their retries
Exit IP	The address the target sees
Fingerprint	Client traits that together identify software or a device
Idempotent job	A job that stores the same result when run twice
JA3, JA4	Methods that hash a TLS ClientHello into a fingerprint

Term	Meaning
Jitter	Randomness added to waits so workers do not retry together
p95 latency	The time under which 95% of attempts finish
Rotating proxy	A gateway that assigns a new exit IP per request
Routing ladder	Routes ordered from cheapest to most capable
Scraper API	A service that fetches a URL and returns the page
Soft block	A 200 response with altered or missing data
Sticky session	One exit IP held for a sequence of requests
Unblocker	A scraper API exposed as a proxy endpoint

Terms used in this book.

ProxyLook tools and further reading

ProxyLook is an independent directory that lists more than 200 proxy providers. Its free tools cover three of the checks above.

- The proxy checker shows the country and network of an exit IP.
- The user-agent parser shows what a header string claims to be.
- The cost calculator turns traffic estimates into a budget.

Two companion books sit beside this one. The Proxy Playbook is the flagship. It covers proxy types, protocols and basic setup. The Multi-Account Handbook is for teams that run several accounts. Multi-accounting can breach a platform's terms of service, and the operator carries that risk.

KEY TAKEAWAYS

- Read the status code as a clue to the layer, then validate the body anyway.
- Run the pre-flight list before the first request to a new target.
- Debug from configuration upward and change one variable per test.
- Keep the legal questions about personal data, logins and terms separate from the technical ones.

KEEP GOING

Where to go next

Method goes in a book. Current prices and provider details belong on a page that can be updated.

Everything below is free and needs no account.

Web scraping proxies

How ProxyLook matches proxy types to scraping targets.

`proxylook.com/use-case/web-scraping`

Scraper APIs and unblockers

Providers that handle proxies, browsers and retries behind one endpoint.

`proxylook.com/use-case/proxy-api-providers`

Live pricing

Current entry prices by proxy type, side by side.

`proxylook.com/pricing`

Proxy checker

Look up a proxy IP's country, ASN, ISP and network type before a run.

`proxylook.com/tools/proxy-checker`

The Proxy Playbook

The flagship guide to choosing, setting up and running proxies.

`proxylook.com/ebooks/the-proxy-playbook`

The Multi-Account Handbook

Antidetect browsers, cloud phones and proxies, working together.

`proxylook.com/ebooks/multi-account-handbook`

ProxyLook

Independent research on proxy networks and antidetect browsers. Free to read, free to share.

proxylook.com