Redacted code excerpts: crawl and load pipelines

Vincent Gregory C. Aboy | for the Python Data Engineer application

These are short excerpts from systems I designed and built with AI coding assistants. Company, site and product names, locale details, schema names and all live data are removed or replaced with <placeholders>. There are no passwords, keys or server paths. I am glad to share more redacted samples on request.

What is in this document

Part	What it shows	Proof level
1. Crawler fetch loop	Retries with backoff, an adaptive delay that slows down on HTTP 429 and cools off after repeated 429s, one session per thread.	Code excerpt. The crawler was run end to end on a local rebuild (run 593, 2026-11-03).
2. Run log and tables	A run log, a per-attempt log and a per-product snapshot, with constraints. This is the history and provenance of every value.	Code excerpt (SQL definitions).
3. Daily load	Temporary table, empty-file guard, replace one snapshot day, all in one transaction.	Code excerpt. Read from the source; not re-run for this document.
4. Page parser	BeautifulSoup parsing, failure classification, 429/5xx retry, a robots check and saved copies of failed pages.	Code excerpt. Read from the source; not re-run for this document.

1. Crawler fetch loop (Python, requests)

Each worker thread has its own session with retry rules for GET requests. An AdaptiveDelay object makes every worker wait longer after HTTP 429 or errors, and shorter after successes. After four 429s in a row, all workers pause for 12 seconds. The loop returns the HTTP status, the parsed price fields, the last error, the bytes downloaded and the number of requests used, so each attempt can be logged.

```
def create_phase1_session() -> requests.Session:
    session = requests.Session()
    session.headers.update(
        {
            "User-Agent": USER_AGENT,
            "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
            "Accept-Language": "<locale list>",
            "Connection": "keep-alive",
        }
    )

    retry = Retry(
        total=MAX_RETRIES,
        connect=MAX_RETRIES,
        read=MAX_RETRIES,
        status=MAX_RETRIES,
        backoff_factor=BACKOFF_FACTOR,
        status_forcelist=RETRY_STATUS_CODES,
        allowed_methods={"GET", "HEAD"},
        raise_on_status=False,
        respect_retry_after_header=True,
    )
    adapter = HTTPAdapter(max_retries=retry, pool_connections=40, pool_maxsize=40)
    session.mount("http://", adapter)
    session.mount("https://", adapter)
    return session


    ...
class AdaptiveDelay:
    def __init__(self) -> None:
        self.lock = threading.Lock()
        self.factor = 1.0
        self.r429_streak = 0
        self.cooloff_until = 0.0

    def current(self, base_min: float, base_max: float) -> Tuple[float, float]:
        with self.lock:
            f = self.factor
        return base_min * f, base_max * f

    def punish(self, hard: bool = False) -> None:
        with self.lock:
            if hard:
                self.factor = min(4.0, self.factor * 1.30)
                self.r429_streak += 1
                if self.r429_streak >= 4:
```

```
                    self.cooloff_until = max(self.cooloff_until, time.time() + 12.0)
                    self.r429_streak = 2
            else:
                self.factor = min(4.0, self.factor * 1.12)

    def reward(self) -> None:
        with self.lock:
            self.factor = max(0.75, self.factor * 0.98)
            if self.r429_streak > 0:
                self.r429_streak -= 1

    def wait_if_cooling(self, stop_event: threading.Event) -> None:
        while True:
            with self.lock:
                remain = self.cooloff_until - time.time()
            if remain <= 0 or stop_event.is_set():
                return
            time.sleep(min(0.5, remain))
```

```
def fetch_page(
    url: str,
    timeout: int,
    delay_min: float,
    delay_max: float,
    adaptive: AdaptiveDelay,
    stop_event: threading.Event,
) -> Tuple[str, Dict[str, str], str, int, int]:
    session = get_thread_session()
    last_err = ""
    body_bytes = 0
    req_count = 0
    for attempt in range(1, MAX_RETRIES + 1):
        if stop_event.is_set():
            return "stopped", {"price_current": "", "price_original": "", "currency": "", "availability": ""},
                "stopped", body_bytes, req_count
        adaptive.wait_if_cooling(stop_event)
        dmin, dmax = adaptive.current(delay_min, delay_max)
        time.sleep(random.uniform(dmin, dmax))
        try:
            req_count += 1
            resp = session.get(url, timeout=timeout)
            body_bytes += len(resp.content or b"")
            if resp.status_code == 200:
                adaptive.reward()
                parsed = parse_price_from_ld_json(resp.text)
                if not (parsed.get("stock_qty") or "").strip():
                    parsed["stock_qty"] = extract_stock_qty_from_html(resp.text)
                return "200", parsed, "", body_bytes, req_count
            last_err = f"HTTP {resp.status_code}"
            adaptive.punish(hard=(resp.status_code == 429))
            if attempt < MAX_RETRIES:
                time.sleep(BACKOFF_FACTOR * (2 ** (attempt - 1)))
                continue
            return str(resp.status_code), {"price_current": "", "price_original": "", "currency": "",
                "availability": ""}, last_err, body_bytes, req_count
        except Exception as exc:
            last_err = str(exc)
            adaptive.punish(hard=False)
            if attempt < MAX_RETRIES:
                time.sleep(BACKOFF_FACTOR * (2 ** (attempt - 1)))
                continue
            return "error", {"price_current": "", "price_original": "", "currency": "", "availability": ""},
                last_err, body_bytes, req_count
    return "error", {"price_current": "", "price_original": "", "currency": "", "availability": ""}, last_err,
        body_bytes, req_count
```

Creating the run row (the run ID ties every later row to one crawl run):

```
def create_run_row(db_name: str, schema: str, run_key: str, started_at: str, workers: int, sleep_base: float,
    timeout_sec: int, autosave_every: int) -> int:
    sql = f"""
insert into {schema}.crawl_run_log (
  run_key, started_at_utc, status, workers, sleep_base, timeout_sec, autosave_every
)
values (
  {sql_literal(run_key)}, {sql_literal(started_at)}, 'running', {workers}, {sleep_base}, {timeout_sec},
      {autosave_every}
)
returning id;
"""
    out = run_psql(db_name=db_name, sql=sql, capture=True)
    lines = [ln.strip() for ln in out.splitlines() if ln.strip()]
    if not lines:
        raise RuntimeError('Failed to capture run_id from crawl_run_log insert.')
    return int(lines[0])
```

2. Database tables: run log, snapshots, attempts (PostgreSQL)

One row per crawl run, one row per product per run, and one row per attempt per product. The run ID, time, product URL and matched competitor code stay with every value, so a price can always be traced back to the run and attempt that produced it. Constraints block duplicate rows for the same run and product, bad statuses and negative stock.

```
CREATE TABLE IF NOT EXISTS <schema>.crawl_run_log (
    id BIGSERIAL PRIMARY KEY,
    run_key TEXT NOT NULL UNIQUE,
    started_at_utc TIMESTAMPTZ NOT NULL,
    ended_at_utc TIMESTAMPTZ,
    status TEXT NOT NULL,
    workers INTEGER NOT NULL,
    sleep_base NUMERIC(10,4),
    timeout_sec INTEGER,
    autosave_every INTEGER,
    max_retries INTEGER,
    retry_backoff_sec NUMERIC(10,4),
    total_target_count INTEGER,
    ok_count INTEGER,
    error_count INTEGER,
    missing_url_count INTEGER,
    note TEXT,
    source_file TEXT,
    created_at_utc TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at_utc TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    CONSTRAINT chk_crawl_run_status
        CHECK (status IN ('running', 'completed', 'completed_with_errors', 'failed', 'partial'))
);
    ...
CREATE TABLE IF NOT EXISTS <schema>.crawl_item_snapshot (
    id BIGSERIAL PRIMARY KEY,
    run_id BIGINT NOT NULL REFERENCES <schema>.crawl_run_log(id) ON DELETE CASCADE,
    match_map_id BIGINT REFERENCES <schema>.match_map(id) ON DELETE SET NULL,
    base_product_sku TEXT NOT NULL,
    base_product_name TEXT,
    matched_competitor_sku_used TEXT,
    product_url TEXT,
    http_status INTEGER,
    price_current NUMERIC(14,4),
    price_original NUMERIC(14,4),
    currency TEXT,
    availability TEXT,
    in_stock BOOLEAN,
    stock_qty TEXT,
    stock_qty_numeric INTEGER,
    stock_qty_band TEXT,
    stock_status_normalized TEXT,
    crawl_status TEXT NOT NULL,
    error TEXT,
    downloaded_kb_est NUMERIC(12,3),
    request_attempts INTEGER,
    snapshot_generated_at TIMESTAMPTZ,
    source_file TEXT,
    source_row_num INTEGER,
    created_at_utc TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at_utc TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    CONSTRAINT uq_snapshot_run_sku UNIQUE (run_id, base_product_sku),
    CONSTRAINT chk_snapshot_crawl_status
        CHECK (crawl_status IN ('ok', 'error', 'missing_url')),
    CONSTRAINT chk_stock_qty_numeric_non_negative
        CHECK (stock_qty_numeric IS NULL OR stock_qty_numeric >= 0)
);
    ...
CREATE TABLE IF NOT EXISTS <schema>.crawl_item_attempt_log (
    id BIGSERIAL PRIMARY KEY,
    run_id BIGINT NOT NULL REFERENCES <schema>.crawl_run_log(id) ON DELETE CASCADE,
    match_map_id BIGINT REFERENCES <schema>.match_map(id) ON DELETE SET NULL,
    base_product_sku TEXT NOT NULL,
    matched_competitor_sku_used TEXT,
    attempt_no INTEGER NOT NULL,
    attempt_phase TEXT NOT NULL,
    attempt_status TEXT NOT NULL,
    ...
    CONSTRAINT uq_attempt_log_run_sku_attempt UNIQUE (run_id, base_product_sku, attempt_no),
    CONSTRAINT chk_attempt_no_positive CHECK (attempt_no >= 1),
    CONSTRAINT chk_attempt_phase CHECK (attempt_phase IN ('first_pass', 'retry_1')),
```

3. Daily load: temporary table, empty-file guard, replace one day (PostgreSQL via psycopg2)

The CSV is loaded into a temporary table that disappears at the end of the transaction. If the file has no rows the load stops with an error and nothing is deleted. Otherwise the snapshot day found in the file is deleted and re-inserted, so re-running the same day does not create duplicates. The long column list and the final insert are shortened here.

```
def _load_full_to_postgres(full_csv: os.PathLike, dsn: str) -> Tuple[int, Optional[str]]:
    import psycopg2

    with psycopg2.connect(dsn) as conn:
        with conn.cursor() as cur:
    ...
            with open(full_csv, "r", encoding="utf-8") as fh:
                cur.copy_expert("copy tmp_pw_full from stdin with (format csv, header true)", fh)

            cur.execute("select count(*) from tmp_pw_full;")
            full_rows = int(cur.fetchone()[0])
            if full_rows <= 0:
                raise RuntimeError("Daily full CSV is empty; aborting productwatch_full load.")
    ...
            cur.execute(
                """
                with snapshot as (
                    select
                        coalesce(
                            nullif(snapshot_date,'')::date,
                            (nullif(snapshot_date,'')::timestamptz)::date
                        ) as snapshot_date
                    from tmp_pw_full
                    where nullif(snapshot_date,'') is not null
                    limit 1
                )
                delete from productwatch_full
                where snapshot_date = (select snapshot_date from snapshot);
                """
            )
```

4. Page parser and retry (Python, BeautifulSoup)

Pages are parsed with BeautifulSoup. A failed fetch is classified (403, 429, 5xx, challenge page, selector not found); 429 and 5xx responses wait with a random pause and retry; the crawler checks robots rules first and saves the HTML of failed pages for inspection.

```
def classify_failure(status: Optional[int], html: str) -> str:
    if status == 403:
        return "403"
    if status == 429:
        return "429"
    if status in (500, 502, 503, 504):
        return "5xx"
    if is_challenge_page(html):
        return "challenge page"
    return "selector not found"


def parse_product(html: str, source_url: str) -> dict:
    soup = BeautifulSoup(html, "lxml")
    text = " ".join(soup.get_text(" ", strip=True).split())
    h1 = soup.select_one("h1")
    canonical = soup.select_one("link[rel='canonical']")
    og_image = soup.select_one("meta[property='og:image']")
    md = soup.select_one("meta[name='description']")

    title = h1.get_text(" ", strip=True) if h1 else ""
    canonical_url = canonical["href"].strip() if canonical and canonical.get("href") else ""
    image_url = og_image["content"].strip() if og_image and og_image.get("content") else ""
    meta_desc = md["content"].strip() if md and md.get("content") else ""
    page_title = soup.title.get_text(" ", strip=True) if soup.title else ""

    brand = title.split()[0] if title else ""
    rating_match = re.search(r"(\d+[.,]\d)\s*(?:/5)?", text)
    reviews_match = re.search(r"\((\d+)\)", text)
    all_prices = re.findall(r"(\d+(?:[.,]\d+)?)\s*<CURRENCY>", text, flags=re.IGNORECASE)
    current_price = all_prices[0].replace(",", ".") if all_prices else ""
    old_price = all_prices[1].replace(",", ".") if len(all_prices) > 1 else ""

    availability = ""
    for token in ("<in-stock phrase>", "<sold-out phrase>", "<unavailable phrase>"):
        if token in text.lower():
            availability = token
            break

    return {
        "source_url": source_url,
    ...
            continue
```

```
resp = None
for _ in range(SETTINGS.max_retries):
    resp = fetch_once(session, url)
    if resp is None:
        continue
    if resp.status_code in (429, 500, 502, 503, 504):
        sleep_jitter()
        continue
    break

status_code = resp.status_code if resp is not None else None
html = resp.text if resp is not None else ""

if resp is None or status_code == 403 or is_challenge_page(html):
```

What I decided and tested, and what the AI wrote

I decided, tested or approved	The AI coding assistant wrote
What to collect, how often, and what must be visible in the history (run, attempt and snapshot tables).	Most of the Python and SQL text shown above, from my written requirements.
Polite crawling rules: few workers, delays, timeouts, retries, and slowing down on 429.	The AdaptiveDelay class and the retry code, to those rules.
The empty-file stop and the replace-one-day rule for the daily load.	The temporary-table SQL and the psycopg2 calls.
Running the crawls, reading the logs, checking results against the source pages, and fixing the rules when something failed.	Drafts of fixes, which I tested before accepting.

Limits, stated plainly

• The crawler shown reads plain HTML with requests; pages that need JavaScript were handled in other scripts with Playwright.

• These excerpts were read from the source for this document. Only the crawler was run end to end (local rebuild, run 593).

• I have not worked with datasets of millions of records, and I have not used Scrapy.

• Excerpts are shortened and reformatted to fit the page; code is otherwise as written.