Lean and Resilient Data Acquisition Infrastructure for Leakage-Proof Survival Modeling in High-Frequency Marketplaces

A job-oriented Docker and Airflow architecture built for high reliability, bounded infrastructure pressure, and audit-grade correctness

Author: <Your Name>
Date: January 14, 2026
Document type: Technical paper / portfolio chapter

Recruiter-oriented highlights
• Job-oriented ingestion: ephemeral containers scheduled by Airflow; no always-on crawler daemons.
• Bounded infrastructure pressure: strict non-overlap (max_active_runs=1), execution timeouts, DB pooling via PgBouncer, and adaptive rate control.
• Audit-grade correctness: append-only forensic tables, mutation guardrails, and explicit SQL verification packs.
• Operational maturity: Prometheus/Grafana observability, Pushgateway job metrics, atomic backups with retention, and nightly partition maintenance.
• Reliability claim (deployment evidence): >65,000 orchestrated runs without a recorded failure, enabled by deterministic containers and defensive orchestration patterns.

Abstract

High-frequency online marketplaces are a uniquely hostile environment for production data acquisition. Listings change state asynchronously (live, sold, removed, inactive), the target surface may throttle or intermittently fail, and downstream analytics such as survival modeling require timestamps and terminal events that are accurate and provably non-leaky. Traditional always-on crawler daemons often trade correctness for throughput, accumulate state that is difficult to debug, and impose constant background load on both the crawler infrastructure and the marketplace surface.

This paper describes a lean, job-oriented acquisition infrastructure built from containerized, single-purpose Go binaries and orchestrated as ephemeral Docker jobs by Apache Airflow. The architecture is engineered to keep infrastructure pressure bounded and predictable: runs never overlap, runtime is time-boxed, database connection pressure is normalized through PgBouncer, and crawler throughput adapts to observed latency and throttle signals. Correctness is enforced through sparse, append-only observation logs, mutation guardrails, and independent sanity auditors that continuously reconcile database state against marketplace truth without creating time-series bloat.

In a long-running deployment, the system has executed more than 65,000 orchestrated runs without a recorded failure while remaining operationally small enough to run on modest single-node infrastructure. We present the design principles, system components, operational guarantees, and verification methodology, and we contrast this approach with daemon-heavy crawler architectures.

Keywords: data acquisition, web crawling, job-oriented pipelines, Airflow, Docker, Postgres, PgBouncer, observability, survival analysis, leakage prevention

Table of Contents

0. Executive summary

This document is written as an interview-ready, engineering-focused paper. It describes a production data acquisition infrastructure for a high-frequency online marketplace that feeds a leakage-aware survival modeling system.

The core engineering claim is straightforward: by treating crawlers as short-lived, deterministic jobs (not long-running daemons), and by enforcing bounded pressure (non-overlap, timeouts, pooling, adaptive throttling), it is possible to run high-frequency acquisition workloads with extremely high reliability on modest infrastructure.

System summary
• Control plane: Apache Airflow schedules all acquisition, audit, and maintenance jobs at 15–30 minute, daily, and monthly cadences.
• Execution model: each crawler is a single-purpose Go binary built into a minimal container; jobs are ephemeral and auto-removed after completion.
• Pressure controls: strict non-overlap (max_active_runs=1), execution timeouts, PgBouncer pooling, and adaptive concurrency/RPS control loops.
• Correctness controls: terminal events derived from platform timestamps when available; sparse history (daily baseline + change events); independent sanity auditors with safe-apply guardrails.
• Operational evidence (deployment): >65,000 orchestrated runs without a recorded failure, with full observability via Prometheus/Grafana and job metrics via Pushgateway.

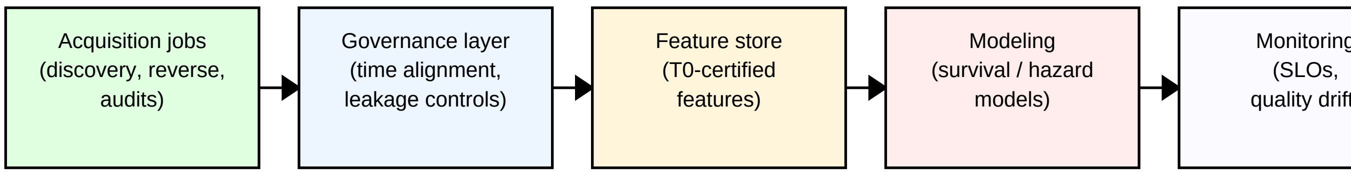

Figure 3: Where acquisition infrastructure fits in an end-to-end leakage-aware modeling stack.

0.1 Competency map

FAANG-relevant capability	Evidence demonstrated in this architecture
SRE / Reliability engineering	Non-overlap scheduling, timeouts, failure isolation via ephemeral jobs, clear SLO metrics, automated backups and maintenance.
Distributed systems and concurrency	Adaptive concurrency gates, token-bucket rate limiting, controlled fan-out, idempotent writes, advisory locks for concurrency safety.
Data engineering	End-to-end ingestion + backfills + audits; sparse event logging; partitioned Postgres with maintenance automation.
Database engineering	Connection pooling, explicit connection caps, partition strategy, index-only scans for sweepers, SQL-checkable invariants.
Observability	Metrics-first design (Prometheus pull + Pushgateway push), profiling endpoints, dashboard-driven operations.
Security and hygiene	Secrets separation strategy, least-privilege roles, container hardening blueprint, publication anonymization.

Evidence and how to defend the reliability claim
• Reliability KPI: compute run success rates from the Airflow metastore (dag_run/task_instance) for a fixed time window.
• Correctness KPIs: SQL invariants for tail policies (no stale-eligible row remains live), plus daily baseline presence checks for sparse history.
• Operational KPIs: throttle ratio, p95 request latency, DB pool saturation, and history table growth rate.
• Backup KPIs: backup success gauge + backup_bytes_written to detect empty dumps.

1. Introduction

Survival analysis and other time-to-event models are only as trustworthy as their event timestamps. In a marketplace setting, the critical event is the listing terminal transition: when a listing becomes sold, when it is removed, and when it becomes inactive or otherwise unavailable. If those transitions are stamped with crawler time rather than platform time, or if terminal events are missed due to throttling, the resulting labels leak information and bias downstream models.

At the same time, acquiring these signals from a public web surface must be operationally sustainable. High-frequency probe loops can overwhelm a small database, cause uncontrolled CPU and network burn, and trigger marketplace-side throttling. The standard response is to scale up always-on crawler fleets; however, for most research and early-stage systems, the real requirement is not raw throughput but sustained correctness and reliability with bounded resource pressure.

This paper documents a production acquisition system designed around a simple thesis: when correctness and reliability are the constraints, a job-oriented architecture can outperform daemon-heavy designs. The system uses short-lived container jobs with deterministic behavior, explicit time bounds, adaptive rate control, and independent auditors. Each job can be started, inspected, and terminated without leaving residual state.

1.1 Contributions

Contribution
A job-oriented acquisition architecture for high-frequency marketplaces that is stable under throttling and modest hardware budgets.
A pressure-bounded execution model: non-overlap scheduling, execution timeouts, explicit DB connection caps via PgBouncer, and adaptive rate control.
A sparse history schema that decouples probe cadence from storage growth (daily baseline + change events + terminal events).
Audit-grade correctness loops: daily tail enforcement, monthly sanity reconciliation, and embargoed post-event enrichment with anti-bloat hashing.
An operable stack: metrics-first binaries, job metrics via Pushgateway, atomic backups with retention, and nightly partition maintenance.

Publication anonymization and non-goals
• No marketplace identifiers, URLs, selectors, or extraction patterns are included.
• All listing ids and example records are synthetic or omitted.
• Operational metrics are reported in aggregate and as time-windowed claims; platform-specific numbers are avoided.
• The paper describes respectful, throttled acquisition and does not imply unauthorized access.

2. System overview

The infrastructure is a compact stack built around Docker containers and a single Postgres instance. Apache Airflow provides the control plane: it schedules, launches, time-boxes, and monitors short-lived jobs. Each job is a single-purpose Go binary packaged in a minimal runtime image. Jobs connect to Postgres through PgBouncer, ensuring that bursts of activity do not translate into bursts of database connections.

The system also includes a monitoring plane based on Prometheus and Grafana. Long-running services expose pull-based metrics, while ephemeral jobs push completion and volume metrics via a Pushgateway. Airflow exports internal scheduling metrics through StatsD for holistic observability (scheduler backlog, task duration, and run success rates).

2.1 Technology stack

Layer	Technologies	Notes
Orchestration	Apache Airflow (CeleryExecutor)	DAGs schedule all jobs; non-overlap and timeouts are enforced in-code.
Execution	Docker (job containers)	Ephemeral containers; deterministic cleanup (auto_remove).
State	PostgreSQL (partitioned) + PgBouncer	Pooler normalizes connections; partitions keep history scalable.
Monitoring	Prometheus + Pushgateway + Grafana	Pull metrics for services; push metrics for ephemeral jobs.
Maintenance	Nightly maintenance jobs; backup runner	Lock/statement timeouts; atomic backups with retention.

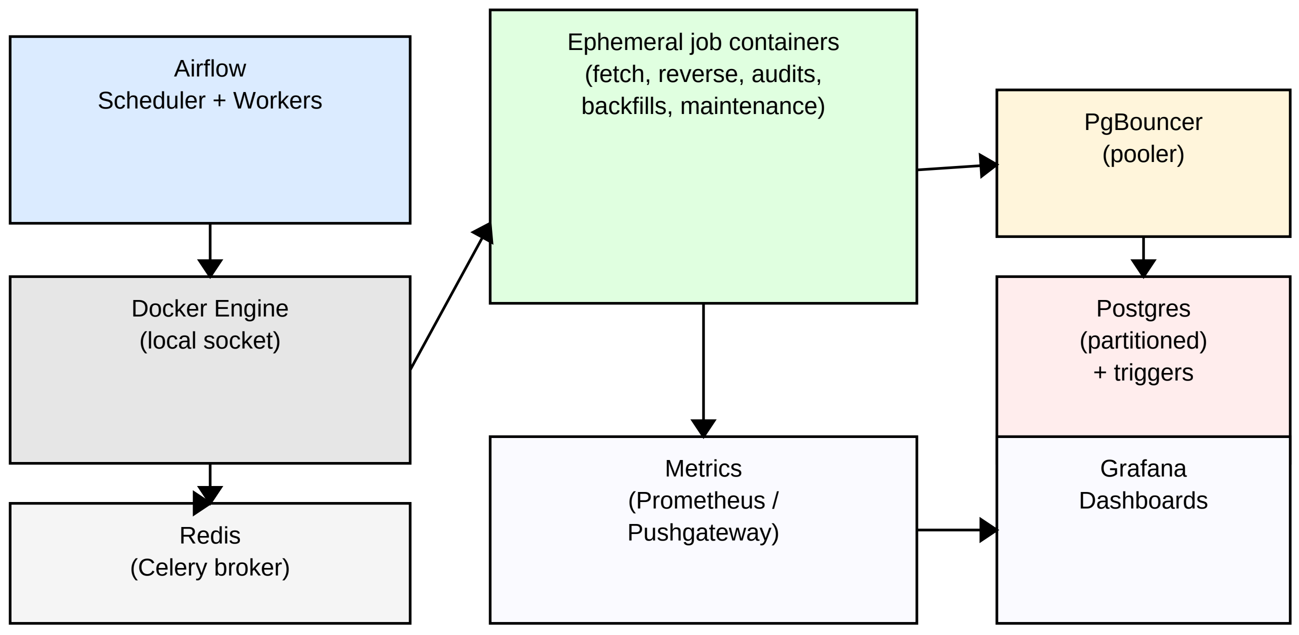

Figure 1: High-level architecture. Airflow launches ephemeral job containers via the local Docker socket. Jobs use PgBouncer to pool database connections and push job-level metrics to the monitoring stack.

2.2 Core components

Component	Role	Why it matters for efficiency and reliability
Airflow (CeleryExecutor)	Schedules and runs jobs; enforces non-overlap and timeouts.	Control-plane separation: orchestration stays stable even when crawlers evolve.
Docker job containers	Execute single-purpose acquisition or maintenance tasks and exit.	Ephemeral execution avoids drift, simplifies upgrades, and limits blast radius.
PgBouncer	Connection pooler between jobs and Postgres.	Normalizes connection pressure, prevents connection storms, and improves latency under bursts.
Postgres (partitioned)	Stores canonical listing state, sparse event logs, and forensic audit tables.	Supports strong invariants, idempotency, and efficient time-bounded queries.
Prometheus / Pushgateway / Grafana	Monitoring for both long-lived services and ephemeral jobs.	Enables SLO dashboards, alerting, and regression detection for throttling and latency.

2.3 Failure modes and defenses

Failure mode	Typical impact	Defense in this architecture
Marketplace throttling (HTTP 429 / latency spikes)	Slowdowns, missing terminal events, unstable run times	AIMD-style adaptive concurrency and RPS limiting; jitter; cool-off windows; conservative retries.
Run overlap amplification	Resource spikes, cascading failures	max_active_runs=1 per DAG; per-task execution_timeout; duty-cycled windows.
Database connection storms	Postgres saturation, lock contention, connection exhaustion	PgBouncer pooling; explicit per-job max connections; batch writes.
Long-lived process drift	Memory leaks, stale caches, unpredictable performance	Ephemeral job containers; auto_remove; restart every schedule tick.
Schema/index regression	Slow sweepers, timeouts, bloat	Nightly maintenance; partition management; SQL verification packs; canary queries.
HTML/structure drift on the surface	Parsing failures, incorrect attribute extraction	Prefer structured embedded state; fallback paths; sanity audits; safe-apply modes that avoid destructive mutations.
Disk growth / bloat	Storage exhaustion, degraded query latency	Sparse history policy; retention pruning; append-only audit tables with deduplication; scheduled cleanup jobs.

3. Core architectural principles

3.1 Single-purpose, compiled binaries

The container images follow a consistent multi-stage build pattern:

• Stage 1 (builder): a Go toolchain image compiles a static binary with trimpath and stripped symbols. • Stage 2 (runtime): a minimal base image provides only CA certificates and timezone data; the binary is copied in and run directly.

Operational advantages:

• Low cold-start overhead. This matters when jobs run every 15–30 minutes. • Reproducible builds. The runtime filesystem contains very little beyond the binary. • Smaller attack surface. Fewer packages and fewer interpreters reduce exposure.

This design also simplifies debugging and rollout: each job is a versioned image that can be rolled forward or rolled back without complex dependency graphs.

3.2 Entrypoints as thin orchestration layers

Entrypoint scripts are intentionally thin orchestration layers. They map environment variables into explicit CLI flags (worker counts, request rate, segment ranges, batch sizes) and dispatch into one of several well-defined modes (sweep, audit, backfill). This pattern prevents "configuration sprawl" and avoids turning bash scripts into a second application codebase.

Crucially, this approach keeps the behavior deterministic: a job run is fully defined by the image tag plus environment, and can be reproduced locally for debugging or performance analysis.

3.3 Job-oriented execution instead of always-on daemons

The system is duty-cycled. Discovery, reverse monitoring, audits, and backfills are executed as bounded jobs, not perpetual services. Container definitions are designed as jobs (explicit profiles, no persistent working volumes), and Airflow schedules them with strict non-overlap.

Non-overlap is the single most important infrastructure protection mechanism: it guarantees that a slow run cannot accumulate concurrent followers.

3.4 Pressure control

PgBouncer is treated as a first-class infrastructure component, not an afterthought. All crawler jobs connect through the pooler. This creates a hard boundary between the crawler's internal concurrency and the database's connection surface. Two practical outcomes: • Burst absorption. A job may execute with high HTTP concurrency but still use a small, bounded number of database connections. • Operational simplicity. Database load becomes a function of a few explicit limits (pool size, per-job caps) rather than a function of how many jobs happen to run concurrently.

The architecture makes it possible to reason about worst-case resource usage with simple bounds: Let: • W be the maximum HTTP concurrency window (adaptive). • R be the maximum request rate (token bucket). • C be the maximum database connections per job (explicit cap). • J be the maximum number of concurrent job families (bounded by scheduling and pools). Then: • HTTP concurrency is bounded by W, independent of transient slowdowns (because W shrinks on latency/throttling). • Marketplace request pressure is bounded by R, independent of worker count. • Database connections are bounded by min(C × J, pool_size), independent of HTTP fan-out. • Job wall-clock impact is bounded by execution_timeout. These bounds are what make the system suitable for long-running, unattended operation on modest infrastructure. The system does not require capacity to "absorb" uncontrolled spikes; it prevents spikes structurally.

3.5 Scheduling and duty-cycling

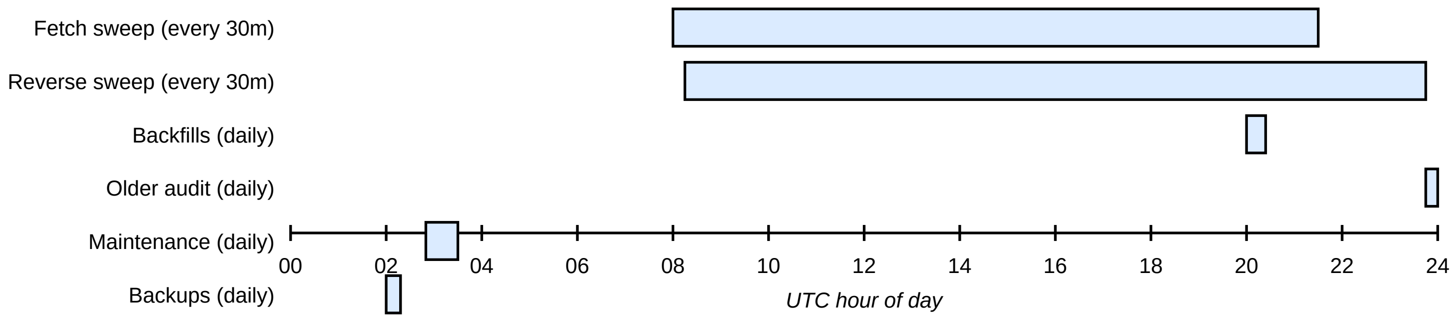

Figure 2: Example daily schedule envelope in UTC. High-frequency jobs run in bounded windows with non-overlap; audits and maintenance run in fixed daily or monthly slots.

4. Crawler and auditor design

4.1 Discovery crawler

The discovery crawler enumerates fresh listing candidates and captures a canonical snapshot of attributes required for modeling. It is optimized for deterministic execution: one job run produces one sweep worth of upserts and exits.

Implementation characteristics derived from the provided artifacts include: a Prometheus /metrics endpoint, optional pprof profiling endpoints, a negative-filter gate to avoid irrelevant candidates, and configurable worker pools for search vs. detail hydration.

Adaptive concurrency and RPS control

To remain stable under throttling and varying network conditions, the crawler combines an adaptive concurrency gate driven by p95 latency and throttle ratio with a token bucket limiter and randomized jitter.

```
# Adaptive concurrency (AIMD-style)
# Inputs: rolling p95 latency, rolling throttle ratio (e.g., HTTP 429 fraction)
# State: baseline_p95 learned online, window size W

every eval_interval:
    p95  = compute_p95(latency_window)
    r429 = throttles_in_window / requests_in_window

    if baseline_p95 not set and p95 > 0:
        baseline_p95 = p95

    base = baseline_p95 if set else default_guess
    too_slow      = p95  > base * slo_multiplier
    too_throttled = r429 > max_throttle_ratio

    if too_slow or too_throttled:
        W = max(W_min, floor(W * 0.70))    # multiplicative decrease
    else:
        W = min(W_max, W + growth_step(W)) # additive increase
```

4.2 Reverse monitor for terminal transitions and sparse history

A reverse monitor revisits listings recorded as live in the database and determines whether they remain live or have transitioned to a terminal state. It is the primary source of high-confidence terminal events for survival modeling.

To prevent history bloat, the monitor writes sparse history: one baseline per UTC day plus change events only when meaningful deltas occur. Terminal transitions always generate a terminal history record.

Bid-mode and unreliable price suppression

Some listings enter flows where the asking price is not a stable explicit value (offer-only or bidding modes). The system detects these cases using structured signals and suppresses price-history writes while bid-mode is active, preventing contamination.

4.3 Auditors as correctness control loops

Auditors run on slower cadences (daily or monthly) and provide a second line of defense for correctness. They are designed to be low-pressure and audit-grade: bounded selection, idempotent inserts, append-only evidence, and safe correction semantics.

4.4 Daily older-listing audit

The daily older-listing auditor enforces a tail policy for listings whose edited timestamp exceeds a fixed threshold (e.g., 21 days). The goal is not merely classification; the goal is to protect downstream modeling from silent tail drift and to keep the canonical state consistent with the sparse history log.

Key design choices:

• The auditor operates in bounded daily windows and uses strict non-overlap. If a run is slow, it cannot create concurrency spikes. • For live listings, it writes exactly one baseline snapshot per UTC day (a fixed bucket timestamp) plus change events. This ensures that long-lived listings remain observable without per-probe bloat. • It tracks "inactive" signals separately as edge-triggered events. The system avoids writing "heartbeat" rows; instead, it only persists a new event when the inactive flag flips. • It refuses to reclassify older/stale listings back to live unless explicitly forced. This prevents catastrophic "resurrections" from transient surface anomalies.

The enforcement mechanism is intentionally two-layered: the auditor produces audit-grade evidence in append-only tables, and a database-side sweeper (trigger-invoked or scheduled) ensures the canonical state satisfies invariants even under concurrent write workloads.

4.5 Monthly status sanity audit

A monthly status sanity audit reconciles database truth against marketplace truth. The key operational problem is that a high-frequency crawler can still miss edge cases: listings that transition between sweeps, transient removals, or platform-side glitches. A sanity audit provides a slower, more conservative control loop.

Notable properties:

• Cadence-bounded selection. A listing is eligible only if it has not been audited within a configurable window (e.g., 30 days). This controls workload and prevents re-auditing the same stable rows. • HEAD-first preflight (optional). Expensive GET hydration is skipped when a low-cost preflight indicates content is gone. • Safe-apply mode by default. In safe mode, the auditor may transition live→sold/removed/stale (with evidence) but refuses sold→live or removed→live "revivals" unless explicitly enabled. • Full forensic traceability. Every run writes a run header row (counts, parameters) and per-listing event rows describing observed vs. stored state and any applied correction.

This auditor is not a replacement for the high-frequency reverse monitor; it is an independent correctness backstop that prevents long-lived labeling drift.

4.6 Post-sold enrichment audit

A post-sold auditor enriches sold listings after a fixed embargo (e.g., 7 days after the recorded sold date). This design intentionally avoids "churn scraping" immediately after sale, when pages are most volatile and signal-to-noise is low.

Key anti-bloat design:

• Idempotent inserts. Each (listing_id, generation, day_offset) snapshot is unique; reruns are safe and cheap. • Hash-based text storage. Full text fields (title, description) are persisted only when they change, detected by a content hash. This preserves important semantic diffs without turning the audit table into a text landfill. • Evidence retention without HTML hoarding. The auditor stores structured evidence and diffs as JSON for debugging and provenance, rather than full raw HTML.

The result is a low-frequency enrichment loop that adds high value for modeling and analysis while preserving storage efficiency.

4.7 Backfills as first-class DAGs

Backfills are implemented as separate, deterministic jobs rather than as ad-hoc scripts. This matters operationally: if enrichment is a first-class DAG, it inherits all of the system's reliability properties (non-overlap, timeouts, observability) and can be rerun safely.

Examples of backfill families include:

• Attribute backfills (e.g., storage tier, condition grade) that are computed from listing payloads and written in batch. • Consistency backfills that ensure derived scores or normalized fields are present for older records.

Backfills are typically chained as sequential tasks within a DAG to prevent concurrent pressure. They are also designed to be resumable and idempotent so that a timeout results in a partial but correct state rather than inconsistent data.

5. Orchestration: Airflow as the control plane

The Airflow DAGs follow a consistent job pattern:

• catchup=False to avoid historical backfills creating unbounded concurrency. • max_active_runs=1 to prevent overlap amplification. • execution_timeout to bound worst-case runtime. • DockerOperator with auto_remove=True for deterministic cleanup. • network_mode configured to use the shared compose network so service discovery is DNS-based (pgbouncer, metrics endpoints, etc.).

These settings are small but high-leverage. They convert an unreliable "best effort" crawler schedule into an engineered control loop with predictable bounds.

5.1 Example schedules and run envelopes

Job family	Typical cadence	Non-overlap	Timeout bound	Notes
Discovery sweep	Every 30 minutes in a bounded daily window	Yes	Tens of minutes	Sequential segment sweep (e.g., G13–G17), tunable throughput.
Reverse monitor sweep	Every 30 minutes	Yes	O(1 hour)	Sparse history writes; terminal transitions always persisted.
Older-listing audit	Daily	Yes	O(2 hours)	Tail policy: never reclassify older listings back to live.
Attribute backfills	Daily	Yes	O(30 minutes)	Runs storage and condition enrichment sequentially.
DB maintenance	Daily	Yes	O(1 hour)	Vacuum, partition rotation, staging pruning with timeouts.
Sanity auditor	Monthly	Yes	O(hours)	Cadence-bounded reconciliation; safe apply mode by default.

5.2 Cross-DAG pressure caps

Non-overlap per DAG prevents self-amplification, but multiple DAGs can still fire near the same time. A natural extension is to use Airflow pools to enforce global concurrency limits for DB-heavy tasks. This preserves isolation while ensuring aggregate pressure remains within a known envelope.

5.3 Sanitized orchestration snippet

```
# Airflow (DockerOperator) — sanitized example
with DAG(
    schedule="*/30 8-23 * * *",
    max_active_runs=1,
    default_args={"retries": 0, "execution_timeout": "60m"},
):
    DockerOperator(
        image="crawler-job:latest",
        auto_remove=True,
        network_mode="compose_default",
        environment={
            "PG_DSN": "postgres://USER:***@pgbouncer:6432/DB",
            "PG_SCHEMA": "marketplace",
            "WORKERS": "64",
            "RPS": "30",
        },
    )
```

6. Data model and correctness invariants

The acquisition layer supports leakage-aware modeling by storing a canonical state and a sparse, append-only observation log. Correctness is expressed as invariants that are testable directly in SQL.

6.1 Canonical listing table

The canonical table stores one row per listing_id and segment, with identity, timestamps, status, and derived attributes. Only controlled jobs mutate canonical status fields.

6.2 Sparse observation log

History is stored sparsely: daily baselines for live listings and event-only rows for changes and terminal transitions. This decouples probe frequency from storage growth.

6.3 Idempotency and append-only evidence

A core reliability property is idempotency: if a job is rerun, it should not corrupt state, duplicate history, or create inconsistent counters.

The system enforces idempotency using:

• unique constraints on history and audit tables, • ON CONFLICT DO NOTHING or ON CONFLICT DO UPDATE patterns, • append-only event logging for forensic tables, • run identifiers and per-run metadata tables for auditors.

Idempotency is not only a correctness property; it is an operational property. It makes timeouts and transient failures survivable because reruns converge to the same state.

6.4 Partitioning and maintenance

Partitioning is essential for keeping high-frequency acquisition tables performant over long time windows. The system partitions large append-only tables (history logs, audit logs) to:

• keep indexes small and cache-friendly, • enable cheap retention and pruning, • ensure that verification and sweeper queries remain time-bounded.

Automated partition maintenance (e.g., via a nightly job) ensures new partitions exist ahead of time, indexes are created consistently, and vacuuming keeps bloat under control. This allows sweepers and auditors to operate with index-only scans in the common case, keeping their marginal cost low even as total history grows.

6.5 Trigger-based enforcement of tail policies

A database-side sweeper enforces tail policies for stale-eligible listings. In the provided system, the sweeper is invoked by a trigger on the canonical table and can also be executed on a schedule for time-driven enforcement.

The sweeper is designed for partitioned tables and uses concurrency-safe mechanisms (advisory locks and partition-aware update paths) to avoid race conditions.

6.6 Verification queries (examples)

```
-- Invariant: no 'live' row may be older than the threshold by edited_date
SELECT COUNT(*) AS live_eligible_over_threshold
FROM marketplace.listings
WHERE status='live'
  AND edited_date IS NOT NULL
  AND edited_date <= now() - interval '21 days';

-- Invariant: stale rows must not be too new
SELECT COUNT(*) AS stale_too_new
FROM marketplace.listings
WHERE status='stale'
  AND edited_date IS NOT NULL
  AND edited_date > now() - interval '21 days';
```

7. Operations: observability, backups, and maintenance

A small infrastructure can still be production-grade if it is operable. The system therefore includes runbook-friendly features:

• Canary checks (e.g., verify a "gold" relation exists before writing backups). • Explicit timeouts (statement_timeout, lock_timeout) for maintenance tasks. • Separate metrics for job success, volume, and duration so alerting can be precise. • Append-only audit trails that support forensic debugging: an operator can answer "what happened" with SQL, not log archaeology.

These practices enable a high reliability claim to be backed by evidence rather than by anecdote.

7.1 SLO and dashboard signals

Signal	Why it matters	How it is measured
DAG run success rate	Primary reliability KPI	Airflow dag_run state distribution; per-DAG success_pct query.
Task duration (p50/p95)	Detect regressions and throttling	Airflow task_instance durations; crawler-exported timers.
Throttle ratio (e.g., 429 fraction)	Indicates marketplace pressure	Crawler counters / rolling windows.
Request latency p95	Feed into adaptive gating and SLOs	Crawler latency histograms; Prometheus.
DB pool saturation	Predict DB contention before failures	PgBouncer stats exporter + Postgres connection metrics.
History table growth rate	Guard against bloat	Daily row counts; disk usage; partition sizes.

7.2 Backup runner design

Backups run as a dedicated container under a minimal cron runner. The script includes a canary check, atomic writes, integrity validation, retention pruning, and Pushgateway metrics.

```
# Backup script (sanitized)
1) Verify a canary relation exists (schema-only dump)
2) Stream a full logical dump -> gzip -> temp file
3) Validate gzip integrity
4) Atomically rename temp -> final
5) Prune backups older than KEEP_DAYS
6) Push success + volume metrics to Pushgateway
```

7.3 Maintenance jobs

Nightly maintenance executes vacuuming and partition maintenance with explicit lock and statement timeouts. This keeps long-lived tables performant and prevents lock storms.

7.4 Reliability evidence and measurement methodology

The deployment has a reported run history exceeding 65,000 orchestrated job runs without a recorded failure. This claim is measured via Airflow metadata (dag_run/task_instance states) and is supported by the system's defensive design (non-overlap, timeouts, idempotency).

For external publication or interviewing, always pair the claim with a reproducible query and a fixed measurement window.

```
-- Airflow metastore: run-level success/failure counts (example)
SELECT
  dag_id,
  COUNT(*) AS total_runs,
  SUM(CASE WHEN state='success' THEN 1 ELSE 0 END) AS success_runs,
  SUM(CASE WHEN state='failed'  THEN 1 ELSE 0 END) AS failed_runs,
  ROUND(100.0 * SUM(CASE WHEN state='success' THEN 1 ELSE 0 END) / COUNT(*), 3) AS success_pct
FROM dag_run
WHERE start_date >= (now() - interval '24 months')
GROUP BY dag_id
ORDER BY total_runs DESC;
```

8. Efficiency and outperformance discussion

A job-oriented acquisition pipeline can outperform daemon-heavy designs when the objective is sustained correctness under bounded resources. Daemon-heavy systems optimize for peak throughput but introduce continuous background load and operational drift.

In contrast, this system bounds pressure structurally and makes runtime behavior predictable.

8.1 Comparative table

Dimension	Daemon-heavy crawler fleets	Job-oriented Airflow + containers (this work)
Steady-state resource use	High (always on)	Low (burst only)
Failure isolation	Lower (shared process state)	High (per-run container boundary)
Operational drift	Common without lifecycle tooling	Minimized (restart every run)
Throttling adaptation	Often global, sometimes rigid	Per-job adaptive control; safe backoff
Storage growth control	Frequently per-probe logging	Sparse history: daily baseline + change events
Auditability	Depends on implementation	First-class: append-only audit tables + SQL invariants
Small infrastructure viability	Often requires scale to be safe	Designed to run reliably on modest single-node setups

8.2 Predictable resource envelope

The architecture makes it possible to reason about worst-case resource usage with simple bounds:

Let: • W be the maximum HTTP concurrency window (adaptive). • R be the maximum request rate (token bucket). • C be the maximum database connections per job (explicit cap). • J be the maximum number of concurrent job families (bounded by scheduling and pools).

Then: • HTTP concurrency is bounded by W, independent of transient slowdowns (because W shrinks on latency/throttling). • Marketplace request pressure is bounded by R, independent of worker count. • Database connections are bounded by min(C × J, pool_size), independent of HTTP fan-out. • Job wall-clock impact is bounded by execution_timeout.

These bounds are what make the system suitable for long-running, unattended operation on modest infrastructure. The system does not require capacity to "absorb" uncontrolled spikes; it prevents spikes structurally.

8.3 Scaling without losing stability

Scaling is performed by adding segments, widening duty-cycle windows, or increasing job frequency, not by removing bounds. When additional throughput is required, add global pools and explicit per-resource budgets (DB, network, target pressure) rather than allowing uncontrolled concurrency.

Because jobs are stateless, horizontal scaling is operationally simple: increase Airflow worker capacity while keeping pools and timeouts as guardrails.

9. Security, privacy, and responsible operation

This paper intentionally describes the system in a platform-agnostic manner and avoids marketplace identifiers, URLs, selectors, or scraping patterns.

Operationally, jobs are designed to be respectful: adaptive rate control, backoff, HEAD preflights, and data minimization.

9.1 Hardening blueprint

Because the acquisition jobs are stateless by design (no persistent volumes, append-only DB writes), they are excellent candidates for strict runtime hardening. A production blueprint includes:

• Run as non-root with a fixed UID/GID. • read_only: true and tmpfs for /tmp when needed. • Drop Linux capabilities and enable no-new-privileges. • Pin egress to required destinations; deny by default. • Use separate DB roles per job family (read/write scopes). • Use Airflow connections or secret stores instead of inline DSNs.

These measures align with modern cloud security expectations and are directly compatible with the job-oriented design.

9.2 Publication anonymization checklist

• Refer to the target as a "high-frequency online marketplace".

• Avoid real URLs, selectors, or platform-specific extraction details.

• Replace listing ids and example records with synthetic examples.

• Keep operational metrics platform-agnostic; report rates and counts in aggregate and with time windows.

• Do not imply unauthorized access; describe respectful, throttled acquisition.

10. Conclusion and extensions

We presented a lean, resilient acquisition infrastructure built around compiled job binaries, Airflow orchestration, explicit pressure bounds, and audit-grade correctness mechanisms.

The architecture demonstrates that high-frequency data acquisition can be efficient and reliable without requiring heavy always-on fleets. The key is to engineer bounds (non-overlap, timeouts, pooling, adaptive throttling) and to make correctness provable (sparse history, audit trails, SQL invariants).

Future work is incremental hardening: unified secrets management, global pools, more formal SLO reporting, and a public sanitized reference implementation with synthetic fixtures.

References

[1] Apache Airflow. "Airflow Documentation". Apache Software Foundation.

[2] Docker. "Docker Documentation". Docker, Inc.

[3] PostgreSQL Global Development Group. "PostgreSQL Documentation".

[4] PgBouncer. "Lightweight connection pooler for PostgreSQL".

[5] Prometheus Authors. "Prometheus Monitoring System".

[6] Grafana Labs. "Grafana Documentation".

Note: References are intentionally generic; this document is written as a portfolio chapter and avoids platform-identifying details.

Appendix A. Sanitized configuration patterns

This appendix includes platform-agnostic configuration patterns that illustrate the system design without exposing sensitive details.

A.1 Multi-stage Go build for minimal runtime images

```
# Dockerfile pattern (sanitized)
FROM golang:alpine AS build
WORKDIR /src
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 go build -trimpath -ldflags "-s -w" -o /out/crawler ./cmd/crawler

FROM alpine:stable
RUN apk add --no-cache ca-certificates tzdata
COPY --from=build /out/crawler /usr/local/bin/crawler
ENTRYPOINT ["/usr/local/bin/crawler"]
```

A.2 Entrypoint pattern: thin runtime dispatch

```
# Entrypoint pattern (sanitized)
: "${PG_DSN:=postgres://USER:***@pgbouncer:6432/DB}"
: "${WORKERS:=64}"
: "${RPS:=30}"

case "${JOB:-sweep}" in
  sweep)   crawler --mode sweep   --pg-dsn "$PG_DSN" --workers "$WORKERS" --rps "$RPS" ;;
  audit)   crawler --mode audit   --pg-dsn "$PG_DSN" --workers "$WORKERS" --rps "$RPS" ;;
  backfill) backfill_tool --dsn "$PG_DSN" --since 365 --concurrency 12 ;;
esac
```

A.3 docker-compose job definition excerpt

```
# docker-compose job pattern (sanitized)
crawler-job:
  image: crawler-job:latest
  profiles: ["job"]
  depends_on:
    pgbouncer:
      condition: service_healthy
  environment:
    PG_DSN: ${PG_DSN}
    PG_SCHEMA: marketplace
    WORKERS: "64"
    RPS: "30"
  command: ["/entrypoint.sh"]
```

Appendix B. Verification query pack

A consistent feature of the system is that operators can prove correctness properties directly in SQL. The following pack is representative.

```
-- 1) Status distribution snapshot
SELECT status, COUNT(*) AS n
FROM marketplace.listings
GROUP BY 1
ORDER BY n DESC;

-- 2) Invariant: no stale-eligible row remains marked live
SELECT COUNT(*) AS live_over_threshold
FROM marketplace.listings
WHERE status='live' AND edited_date <= now() - interval '21 days';

-- 3) Daily baseline presence for sparse history (UTC bucket)
WITH p AS (
  SELECT date_trunc('day', now() AT TIME ZONE 'UTC') + interval '5 minutes' AS bucket
)
SELECT COUNT(*) AS baselines_today
FROM marketplace.price_history ph, p
WHERE ph.source='audit-older'
  AND ph.status='live'
  AND ph.observed_at = p.bucket;
```

Appendix C. Metrics contract

A monitoring stack is only useful if metrics are standardized. The following contract is representative for crawler jobs and maintenance jobs.

Metric	Type	Description
crawler_requests_total	counter	Total HTTP requests issued by a crawler job, labeled by outcome class (2xx/3xx/4xx/5xx).
crawler_throttles_total	counter	Number of detected throttle responses (e.g., 429) and cooldown activations.
crawler_request_latency_seconds	histogram	Request latency distribution used for p95 computation and SLO gating.
crawler_concurrency_window	gauge	Current adaptive concurrency window size (AIMD gate).
crawler_rps_limit	gauge	Current token-bucket RPS limit used by the job.
job_success	gauge	1 on successful completion, 0 otherwise; pushed for ephemeral jobs.
job_duration_seconds	gauge	Wall-clock runtime for the job run.
backup_bytes_written	gauge	Size of the latest backup artifact; useful for canarying empty dumps.

Appendix D. Container hardening checklist

These hardening steps are compatible with stateless, job-oriented crawler containers.

- Run containers as non-root (USER directive; avoid UID 0).
- Set read-only root filesystem; provide tmpfs for /tmp if required.
- Drop all Linux capabilities; enable no-new-privileges.
- Disable privilege escalation and host PID/IPC namespaces.
- Pin images by digest in production; scan images for CVEs.
- Use least-privilege Postgres roles per job (read-only vs writer vs admin).
- Centralize secrets in Airflow connections or Docker secrets; avoid inline DSNs.
- Apply egress controls (network policies / firewall) to limit outbound targets.

Appendix E. Production readiness checklist

A compact stack can still be production-grade if it is operable. Use this checklist when presenting or deploying the system.

• Dashboards: success rate, task duration p95, throttle ratio, DB pool saturation, history growth.

• Alerts: job failure, missing scheduled baseline, backup failure, disk usage threshold, DB lock contention.

• Runbooks: throttle escalation, DB pressure incident, schema/index regression, restore procedure.

• Canaries: pre-backup table check; post-run sanity query to validate invariants.

• Change management: versioned images; staged rollouts; reproducible config diffs.

• Data governance: T0 alignment rules and leakage checks before feature export.

Appendix F. Glossary

• Baseline snapshot: a once-per-day record for live listings, written at a fixed UTC bucket time.

• Terminal event: a state transition that ends the listing lifecycle for modeling purposes (e.g., sold, removed, stale).

• Job-oriented pipeline: an architecture where work is performed by short-lived jobs that start, execute, and exit, rather than by long-running daemons.

• Non-overlap: an orchestration constraint that prevents concurrent runs of the same job from executing simultaneously.

• Sparse history: an observation log that records only daily baselines and meaningful changes, not every probe.