# ATTOM Ingestion & Instant Fetch — Microservice Blueprint

Goal: **p95 < 200ms** property fetch for known addresses, with **on‑demand hydration** from ATTOM when data is missing or stale, and background pipelines to keep hot geographies fresh. Inspired by consumer RE portals’ patterns (pre‑ingest + cache + search index + SWR/stale‑while‑revalidate).

## 1) High‑Level Architecture

+-------------------+ +-------------------+  
| Next.js Frontend | -----> | API Gateway |  
+-------------------+ +-------------------+  
 | (REST)  
 v  
 +------------------+ +------------------+  
 | Property API | <-----> | Redis Cache |  
 | (search-api) | | (hot keys) |  
 +------------------+ +------------------+  
 | (read/write)  
 v  
 +------------------+ +------------------+  
 | Postgres (OLTP) | <------> | OpenSearch index |  
 | (normalized) | ETL | (search/filters) |  
 +------------------+ +------------------+  
 | (events)  
 v  
 +------------------+ +------------------+  
 | Hydrator Service | <------> | ATTOM API |  
 | (workers + SQS) | | (provider) |  
 +------------------+ +------------------+  
 |  
 v  
 +------------------+  
 | Object Store | (raw JSON snapshots)  
 +------------------+

**Key ideas** - **Cache‑first reads**: Property API checks Redis → Postgres → (optional) ATTOM via Hydrator. Returns immediately; background refresh keeps cache warm. - **Pre‑ingest + On‑demand**: Nightly/geo‑based pre‑ingest to seed hot markets; on‑demand hydration for cold misses. - **Event‑driven**: Hydrator publishes property.updated → Indexer updates OpenSearch and materialized views. - **Idempotency**: Canonical property key ensures de‑dupe across providers/sessions.

## 2) Data Flow Patterns (Zillow/Redfin/BatchLeads/Propwire‑style)

1. **Type‑ahead prefetch**: As user types an address, the frontend calls /resolve?partial=...; API predicts top matches (OpenSearch). Client fires a silent hydrate for top candidate → by the time user clicks, data is hot.
2. **SWR (stale‑while‑revalidate)**: Serve cached snapshot in <50ms; trigger background refresh if stale\_at < now.
3. **Materialized Property View**: denormalized JSON view for fast detail pages; normalized relational tables for integrity/history.
4. **Hot‑zone cron**: Pre‑ingest recent listings & top ZIPs daily/6‑hourly; cheaper than hydrating everything.
5. **Progressive enrichment**: show core facts instantly (address, beds/baths, AVM) then fold in owner, liens, permits as they arrive.

## 3) Canonical Identity & Idempotency

* **Canonical Address**: USPS/Libpostal normalize → lowercase → strip punctuation → collapse whitespace.
* **Parcel/APN** if present wins; otherwise (addr\_line1|city|state|zip) becomes property\_key.
* **Idempotency key** for jobs: sha256("attom|v1|" + property\_key + "|scope:full").

-- Uniqueness to prevent dupes (one of these must be present)  
CREATE UNIQUE INDEX IF NOT EXISTS ux\_properties\_property\_key ON properties(property\_key);  
CREATE UNIQUE INDEX IF NOT EXISTS ux\_properties\_parcel\_apn ON properties(parcel\_apn) WHERE parcel\_apn IS NOT NULL;

## 4) Postgres Schema (core)

CREATE TABLE IF NOT EXISTS properties (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 property\_key TEXT NOT NULL, -- canonical address key  
 parcel\_apn TEXT, -- if available  
 attom\_id TEXT, -- provider ID (nullable)  
 address\_line1 TEXT NOT NULL,  
 city TEXT NOT NULL,  
 state TEXT NOT NULL,  
 zip TEXT NOT NULL,  
 lat DOUBLE PRECISION,  
 lon DOUBLE PRECISION,  
 beds SMALLINT,  
 baths SMALLINT,  
 sqft INTEGER,  
 year\_built INTEGER,  
 lot\_sqft INTEGER,  
 property\_type TEXT,  
 status TEXT, -- active/offmarket/etc  
 last\_provider TEXT, -- "attom"  
 last\_fetch\_at TIMESTAMPTZ,  
 stale\_after TIMESTAMPTZ, -- SWR trigger  
 data\_version INTEGER DEFAULT 1,  
 source\_hash TEXT, -- hash of last normalized payload  
 updated\_at TIMESTAMPTZ NOT NULL DEFAULT now(),  
 created\_at TIMESTAMPTZ NOT NULL DEFAULT now()  
);  
  
CREATE TABLE IF NOT EXISTS property\_owners (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 property\_id UUID NOT NULL REFERENCES properties(id) ON DELETE CASCADE,  
 owner\_full\_name TEXT,  
 mailing\_address TEXT,  
 phone\_normalized TEXT[],  
 email TEXT[],  
 updated\_at TIMESTAMPTZ NOT NULL DEFAULT now()  
);  
  
CREATE TABLE IF NOT EXISTS property\_features (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 property\_id UUID NOT NULL REFERENCES properties(id) ON DELETE CASCADE,  
 features JSONB NOT NULL,  
 updated\_at TIMESTAMPTZ NOT NULL DEFAULT now()  
);  
  
CREATE TABLE IF NOT EXISTS valuations (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 property\_id UUID NOT NULL REFERENCES properties(id) ON DELETE CASCADE,  
 avm\_value NUMERIC,  
 avm\_confidence NUMERIC,  
 last\_sold\_price NUMERIC,  
 last\_sold\_date DATE,  
 provider TEXT NOT NULL, -- attom  
 updated\_at TIMESTAMPTZ NOT NULL DEFAULT now()  
);  
  
CREATE TABLE IF NOT EXISTS provider\_raw\_snapshots (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 property\_key TEXT NOT NULL,  
 provider TEXT NOT NULL,  
 payload JSONB NOT NULL,  
 fetched\_at TIMESTAMPTZ NOT NULL DEFAULT now(),  
 payload\_sha256 TEXT NOT NULL  
);  
  
CREATE TABLE IF NOT EXISTS hydrate\_jobs (  
 id UUID PRIMARY KEY DEFAULT gen\_random\_uuid(),  
 idempotency\_key TEXT NOT NULL,  
 property\_key TEXT NOT NULL,  
 scope TEXT NOT NULL, -- e.g. "core", "owner", "full"  
 state TEXT NOT NULL, -- queued|running|succeeded|failed  
 attempts INT NOT NULL DEFAULT 0,  
 last\_error TEXT,  
 created\_at TIMESTAMPTZ NOT NULL DEFAULT now(),  
 updated\_at TIMESTAMPTZ NOT NULL DEFAULT now()  
);  
  
-- Fast lookups  
CREATE INDEX IF NOT EXISTS idx\_properties\_city\_state\_zip ON properties(city, state, zip);  
CREATE INDEX IF NOT EXISTS idx\_properties\_geo ON properties USING GIST (ll\_to\_earth(lat, lon));  
CREATE INDEX IF NOT EXISTS idx\_properties\_stale ON properties(stale\_after);

**Materialized view for fast detail** (refresh on events):

CREATE MATERIALIZED VIEW IF NOT EXISTS mv\_property\_detail AS  
SELECT p.\*,  
 (SELECT jsonb\_agg(po ORDER BY po.updated\_at DESC) FROM property\_owners po WHERE po.property\_id = p.id) AS owners,  
 (SELECT jsonb\_agg(v ORDER BY v.updated\_at DESC) FROM valuations v WHERE v.property\_id = p.id) AS valuations,  
 (SELECT jsonb\_agg(pf ORDER BY pf.updated\_at DESC) FROM property\_features pf WHERE pf.property\_id = p.id) AS features  
FROM properties p;

## 5) Redis Caching Strategy

* **Keys**: prop:{property\_key}, prop:apn:{apn} → value: compact JSON for detail page.
* **TTL**: 7 days for static facts; 24h for volatile fields (status, AVM). Use **two buckets** or embed per‑field stale\_at timestamps.
* **SWR**: if now > stale\_at, return cached payload **and** enqueue hydrate job.
* **Negative cache**: prop:miss:{property\_key} for 30–60 min to avoid stampedes.

## 6) Public API Contract (Property API)

GET /v1/properties/resolve?address=123+Main+St+Austin+TX+78704  
 → 200 { data, freshness: "fresh|stale|fetching", source: "cache|db|provider" }  
  
GET /v1/properties/:id  
 → 200 { mv\_property\_detail fields }  
  
POST /v1/properties/hydrate { address, scope?: "core|owner|full" }  
 → 202 { job\_id, state }  
  
GET /v1/search?q=austin%20marble%20ridge&filters=... (OpenSearch)  
 → 200 { hits: [...], facets: {...} }

**Behavior** - **Hit (fresh)**: serve from Redis (<50ms). - **Hit (stale)**: serve + enqueue refresh (SQS) → user sees up‑to‑date shortly. - **Miss**: create hydrate\_job, return {freshness:"fetching"} and a minimal skeleton (address only). Frontend polls/SSE.

## 7) Hydrator Service

**Responsibilities** - Pull jobs from SQS/Redis stream. - Call ATTOM (respect rate limits, retries, circuit breaker). - Store raw payload → S3/Object store + provider\_raw\_snapshots. - Normalize → upsert into properties, valuations, etc. within a transaction. - Compute source\_hash; if changed → publish property.updated (Kafka/SNS) for index refresh.

**Concurrency & limits** - Token bucket per‑endpoint; exponential backoff on 429/5xx. - Idempotent by idempotency\_key (unique index on hydrate\_jobs if desired).

**Pseudo (Go)**

func Handle(job Job) error {  
 if seen(job.IdemKey) { return nil }  
 raw, etag := attom.Fetch(job.Address, job.Scope)  
 storeRaw(job.PropertyKey, raw)  
 tx := db.Begin()  
 norm := Normalize(raw)  
 upsertProperty(tx, norm.Property)  
 upsertValuations(tx, norm.Valuations)  
 tx.Commit()  
 if changed(etag, norm) { publish("property.updated", job.PropertyKey) }  
 cache.Set("prop:"+job.PropertyKey, BuildDetail(norm), TTL)  
}

## 8) Indexer & Search

* **OpenSearch mappings**: text (address), keyword (city/state/zip/type), numeric (beds/baths/sqft/price), geo\_point (lat/lon).
* **Ingest pipeline** to lowercase/normalize address tokens.
* **Suggesters** for type‑ahead.
* **Event consumer** updates index on property.updated.

## 9) Frontend Fetch Pattern (Next.js)

* **Instant view**: GET /resolve on route enter; show detail skeleton; if freshness:"fetching", show loader ribbon.
* **Optimistic prefetch**: on type‑ahead select, call POST /hydrate before navigation.
* **SSE/poll**: subscribe to job\_id → swap in fresh payload on succeeded.

## 10) Folder/Repo Layout

repo/  
 apps/  
 web/ # Next.js (app router)  
 services/  
 property-api/ # Fastify/TS or Go HTTP API  
 hydrator/ # Go workers (SQS)  
 indexer/ # Node/TS or Go  
 packages/  
 shared/ # types, canonicalize, validators  
 infra/  
 docker-compose.yml # Postgres, Redis, OpenSearch, LocalStack  
 k8s/ # Deployments, HPAs, Secrets

**docker-compose (excerpt)**

services:  
 postgres:  
 image: postgres:16  
 environment:  
 POSTGRES\_PASSWORD: local  
 ports: ["5432:5432"]  
 redis:  
 image: redis:7  
 ports: ["6379:6379"]  
 opensearch:  
 image: opensearchproject/opensearch:2  
 environment: { discovery.type: single-node }  
 ports: ["9200:9200"]  
 localstack:  
 image: localstack/localstack  
 environment: { SERVICES: sqs,s3 }  
 ports: ["4566:4566"]

## 11) Example Endpoints (TypeScript Fastify)

app.get('/v1/properties/resolve', async (req, reply) => {  
 const { address } = req.query as { address: string };  
 const key = canonicalize(address);  
 const cached = await redis.get(`prop:${key}`);  
 if (cached) {  
 const payload = JSON.parse(cached);  
 if (isStale(payload)) enqueueHydrate(key, 'full');  
 return reply.send({ data: payload, freshness: isStale(payload)?'stale':'fresh', source: 'cache' });  
 }  
 // fallback DB  
 const row = await db.oneOrNone('select \* from mv\_property\_detail where property\_key=$1', [key]);  
 if (row) {  
 if (isStale(row)) enqueueHydrate(key, 'full');  
 await redis.setEx(`prop:${key}`, 7\*86400, JSON.stringify(row));  
 return reply.send({ data: row, freshness: isStale(row)?'stale':'fresh', source: 'db' });  
 }  
 // miss → enqueue and return placeholder  
 const jobId = await enqueueHydrate(key, 'full');  
 await redis.setEx(`prop:miss:${key}`, 1800, '1');  
 return reply.code(202).send({ data: { address }, freshness: 'fetching', job\_id: jobId, source: 'provider' });  
});

## 12) Normalization Rules

* Use **libpostal** or USPS API to expand/standardize.
* Lowercase, remove punctuation, normalize suffixes (st → street, ave → avenue), collapse whitespace.
* Validate ZIP/state; geocode to lat/lon if absent.
* Strip provider‑specific oddities; map to our enums.

## 13) Rate Limits & Backoff

* Global concurrency cap per ATTOM endpoint.
* **429/503**: backoff (jittered exponential), dead‑letter after N attempts; operator alert.
* **Circuit breaker** to short‑circuit failing endpoints for 1–5 minutes.

## 14) Observability & SLOs

* **Metrics**: cache hit rate, p50/p95 latency per route, hydrate queue depth, ATTOM error rates, index lag.
* **Tracing**: propagate X‑Request‑ID; spans across API → hydrator → DB → OpenSearch.
* **Logs**: per‑job state transitions with user/context for billing/quotas.
* **SLO**: 99% GET /resolve < 250ms for cached; 95% hydrate completes < 8s.

## 15) Security & Governance

* Store provider keys in KMS/Secrets; never in logs.
* **Row‑level security** for user‑owned artifacts; property facts are public domain but owner data may be restricted.
* PII minimization; audit who accessed owner contact details.

## 16) Cost Controls

* Prioritize cache hit rate (>85%).
* Geo pre‑ingest only for top ZIPs (sliding window of last 30 days of traffic).
* Batch hydrations; dedupe via idempotency key.

## 17) MVP Rollout Plan (7 steps)

1. Spin up Postgres/Redis/OpenSearch (docker‑compose).
2. Implement canonicalize() and property\_key + DB schema.
3. Build Property API /resolve with Redis/DB fallbacks and SWR.
4. Add Hydrator worker hitting ATTOM “core property” endpoint; write raw+normalized; enqueue from API.
5. Create property.updated → Indexer populates OpenSearch; ship /search.
6. Frontend: type‑ahead + optimistic prefetch + SSE/poll.
7. Cron: hot‑zone pre‑ingest + dash for metrics.

## 18) Nice‑to‑Haves (Phase 2)

* **Materialized view refresh** via incremental triggers instead of full refresh.
* **Per‑field freshness** (e.g., AVM daily, tax yearly) with multi‑TTL payloads.
* **Multi‑provider arbitration** (ATTOM + backups) with confidence scores.
* **CQRS Split**: separate read model for detail page vs analytics.

## 19) Test Strategy

* Unit test canonicalizer and upserts (idempotency, conflict paths).
* Contract tests for ATTOM responses (golden JSON fixtures).
* Load test /resolve for cache hit/miss scenarios (10k RPS goal with Redis).
* Chaos tests for provider outages; verify circuit breaker + graceful degrade.

## 20) Snippets You Can Copy

**Go Hydrator: job runner (skeleton)**

type Job struct { Id string; IdemKey string; PropertyKey string; Scope string }  
  
func WorkerLoop() {  
 for job := range Dequeue() {  
 if err := Handle(job); err != nil { fail(job, err) } else { succeed(job) }  
 }  
}

**TypeScript canonicalize.ts**

export function canonicalizeAddress(s: string) {  
 const x = s.toLowerCase().trim()  
 .replace(/\./g, '')  
 .replace(/\bst\b/g, 'street')  
 .replace(/\bave\b/g, 'avenue')  
 .replace(/\s+/g, ' ');  
 return x;  
}  
export const propertyKey = (a: {line1:string, city:string, state:string, zip:string}) =>  
 `${canonicalizeAddress(a.line1)}|${a.city.toLowerCase()}|${a.state.toLowerCase()}|${a.zip}`;

**Redis SWR helper**

export function markStale(payload: any, days=7){  
 const dt = new Date(); dt.setDate(dt.getDate()+days);  
 return { ...payload, stale\_at: dt.toISOString() };  
}  
export function isStale(payload:any){  
 return payload?.stale\_at && Date.now() > Date.parse(payload.stale\_at);  
}

If you want, we can tailor this to your exact stack (Go vs TS for API), wire in your existing canonicalize helpers, and generate a ready‑to‑run starter repo skeleton.