ChatGPT

Continuous Apartment-Listing Discovery System Design

This proposal describes a continuous discovery system for a personal apartment-hunting tool serving San Francisco. The user cares about finding new listings quickly, keeping the dataset fresh and broad, and robustly retrying failures. The design is not aimed at enterprise-grade security but emphasises correctness, freshness, and operational reliability.

A) Architecture sketch

```
+------------------------+               +--------------------+
|       User Profiles    |               |  Observability &   |
|  (desired areas, price,|---subscribe-->|   Metrics Store    |
|  amenities)            |               | (Prometheus/Grafana)|
+-----------+------------+               +---------+----------+
            |                                      ^
            |                                      |
            v                                      |
+-----------+------------+  new listing radar      |
|  Scheduler Service      |----------enqueue--------+
|  (priority rules &      |
|  dynamic recrawl cadence|
+-----------+------------+
            |
    +-------+-------+                         +----------+-------+
    |   High-Priority|                        |  Standard        |
    |   Queue        |                        |  Queue           |
    | (high-velocity  |                        |  (PM sites &     |
    |  sources & user |                        |   slower sites)  |
    |  “radar” tasks) |                        +-----------------+
    +--------+-------+                                |
             |                                        |
            Workers (many replicas)                     |
             |                                          |
+------------+------------+                        +----+-----------+
|  Firecrawl/Fetch Layer   |  --retry & dedupe-->  | Change-Detect  |
|  (HTTP requests, rate    |                       |  Processor     |
|  limiting, proxies)      |                       | (hash diff,    |
+------------+------------+                        |  diff fields)  |
             |                                     +----+-----------+
             |                                          |
```

```
+------------v------------+                              v
|   Parse & Extract Data   |--metadata-->+-------------+------------+
|  (scrape listing details |             |    Storage & History DB    |
|   & links)               |             |   (listings, change log)  |
+-------------------------+             +-------------+------------+
                                             |         |
                                             |         +--geocoder/route batch
jobs
                                             |
                                     +-----------v----------+
                                     |  Notification Engine  |
                                     |  (detect new/changed  |
                                     |   matches for profiles|
                                     +----------------------+
```

Components

1. **Scheduler Service** – central brain that continuously decides what to crawl next. It maintains per-source metadata (last crawled time, historical change rate, failure counts) and uses dynamic recrawl policies.
2. **Queues** – separate queues for high-velocity sources (Craigslist, Facebook Marketplace, etc.) and slower property-management sites. A special *radar* queue handles user-defined high-priority queries or neighbourhoods.
3. **Workers** – stateless fetchers (async tasks or serverless functions) that pop URLs from queues, fetch pages using **Firecrawl** or a headless browser, enforce politeness, handle retries/exponential back-off, and push results to parsing.
4. **Firecrawl/Fetch Layer** – interacts with Firecrawl API (or built-in fetcher) to request pages. It sets `changeTracking` and `markdown` formats to let Firecrawl compare current pages to previous versions; Firecrawl returns `changeStatus` (`new`, `same`, `changed`, `removed`) and timestamps [1].
5. **Parse & Extract** – interprets raw HTML / Firecrawl results into structured listing records (address, price, bedrooms, description, images). New URLs discovered on listing pages are normalised and deduped before enqueuing.
6. **Change-Detection Processor** – compares the new record to the previous version. It uses field-level hashing and diffing. For each listing, compute hashes of price, text description, amenity fields and image URLs; when any hash changes, log a change event. Use tree-based hashing for structural changes to pages as described by Goel & Aggarwal: assign hash values to leaf nodes and tag values to non-leaf nodes in a DOM tree; compute bottom-up to identify modified nodes [2].
7. **Storage & History DB** – stores current listing snapshot plus a history table capturing each change event with timestamp and diff summary. Provide APIs for queries like “show price history for listing X.”
8. **Notification Engine** – monitors change events and user profiles. When a new listing or a significant change matches a user’s criteria, it sends alerts (“Your top profile has 3 new matches since 9 AM”).
9. **Observability & Metrics** – collects metrics (crawl coverage, freshness, error rates, queue sizes) and displays dashboards and alerts.

B) Crawl scheduling rules

Scheduling decides what to crawl next. The **URL Frontier** should support multiple strategies:

1. **Priority scoring** – Each URL or source gets a score based on importance and likelihood of change. Use signals like:
2. **Historical change frequency** – pages that change often should be revisited sooner [3]. Track last-modified times or Firecrawl's `previousScrapeAt` and adapt.
3. **Source velocity** – classify sources as *high-velocity* (e.g., Craigslist, Zillow, Facebook groups) or *slow* (property-management sites). High-velocity sources receive shorter crawl intervals (minutes or hours). Slower sources might be recrawled daily or weekly.
4. **User interest** – if many users watch a neighbourhood or a price band, increase priority for listings in that segment (the **radar** queue). Provide "new listing radar" mode that polls specific sources/queries with high frequency during business hours.
5. **Listing age** – new pages discovered recently get higher scores for the first day, then gradually decrease as they become stale.
6. **Failure/back-off** – penalise sources with repeated errors (4xx/5xx). Use exponential back-off on error counts.
7. **Separate per-domain queues** – to respect politeness, maintain domain-level buckets so each domain is crawled independently with rate limits [4]. Limit concurrent requests per domain.
8. **Revisit policy** – compute a *next crawl time* per page based on its last change time and change frequency distribution. For example, if a listing's price changed 4 times in the last week, schedule more frequent recrawls. For pages that haven't changed in months, schedule recrawls weekly or monthly. Use time-based scheduling as recommended in incremental crawlers [5].
9. **Deduplication & normalization** – before enqueuing a URL, normalise (remove `#` fragments, resolve relative links) and check a Bloom filter or fingerprint database to avoid duplicates [6]. Use SimHash or similar to detect near-duplicate pages [7].
10. **Back-pressure & batch limits** – when queues grow, temporarily lower the crawling rate to control costs. Prioritise tasks with highest scores and drop extremely low-value tasks (e.g., old pages outside user's price bands).

Pseudo-code for scoring:

```
score(url) = w_source * source_velocity_factor + w_change *
estimated_change_prob \
            + w_interest * user_interest_factor - w_age * page_age - w_fail *
error_penalty
```

Jobs with higher scores are popped first. The scheduler continuously updates scores as new information arrives.

C) Change detection design

1. **Hash-based field diffs** – For each listing, compute a hash of each field (price, availability, bedrooms, description text, images). Store the hashes and values. On subsequent crawls, recompute and compare. If hashes differ, record a change event with `old_value`, `new_value`, timestamp, and difference type (price-drop, description change, photo update, etc.).
2. **Tree-based change detection** – For pages where structural changes matter (e.g., a property page's layout), model the DOM as a tree and assign hash values to leaf nodes and tag values to non-leaf nodes; compute bottom-up to identify modified nodes as described by Goel & Aggarwal [8]. This approach is efficient for capturing both structural and content changes.
3. **Firecrawl changeTracking** – Use Firecrawl's built-in change-tracking mode to detect whether a page is `new`, `same`, `changed`, or `removed` [9]. When the status is `changed`, Firecrawl can return a `git-diff` or `json` diff showing modified text or structured fields. Use JSON mode to focus on price and availability fields for product-like pages [10].
4. **Image change detection** – Compute perceptual hashes (pHash) of each photo. Compare new vs previous pHashes; if the Hamming distance exceeds a threshold, mark the photo as changed.
5. **Time series** – Record change events in a history table keyed by listing ID. Each entry stores the diff summary and timestamp. This allows analytics on price trajectories and helps predict future changes.

D) Freshness metrics and alerts

To ensure the dataset stays current and that high-priority profiles surface new matches quickly, the system should collect metrics:

Metric	Purpose / Description
Coverage	Number/percentage of known sources crawled in the last *N* hours (per domain, per neighbourhood). Identify coverage holes when certain areas or price ranges have not been updated recently.
Freshness score	Average age of listings (time since last crawled). Compute per-listing and overall distribution. Use as a target (e.g., 95 % of listings updated within 24 h).
Change detection rate	Ratio of crawls that resulted in a change event. High rates on a source indicate it is "hot" and may justify higher crawl frequency.
Error budget	Track HTTP errors, parsing failures, and timeout rates. Set an error budget (e.g., error rate <2 %). Alerts fire when error budgets are exceeded so engineers can investigate.
Queue lag	Time tasks spend in the queue before being processed. Long lag indicates insufficient worker capacity.
User alert metrics	Number of new/changed listings delivered to each user profile per day. Use this to provide notifications like "your top profile has 3 new matches since 9 AM".

Alerts and dashboards (e.g., using Grafana) can display these metrics and highlight anomalies. For example, if the *freshness score* for a high-velocity source exceeds 3 hours, an alert triggers to accelerate its crawl cadence.

E) MVP (2-week) vs ultimate (2–3-month) roadmap

Minimal viable product (≈2 weeks)

- **Scope** – Focus on a handful of high-velocity sources (e.g., Craigslist SF apartments, Zillow, a couple of property-management sites). Ignore dynamic rendering and anti-bot circumvention for now.
- **Tech stack** – Use a serverless or simple queue/worker architecture (e.g., cron-based scheduler, Firecrawl API) running on one machine. Store listings and change history in a relational DB (SQLite/PostgreSQL). Build a minimal web UI to show current listings and change logs.
- **Scheduling** – Hard-coded crawl intervals: high-velocity sources every 30 min, slower sources once daily. Basic deduplication using a Bloom filter. Use Firecrawl's `changeTracking` to detect new/changed/removed pages [1].
- **Change detection** – Hash only key fields (price, status, description). Use Firecrawl JSON mode to track price and availability changes [10]. Log changes; send simple email notifications when a listing's price drops or a new listing matches a user's saved search.
- **Observability** – Collect simple metrics: number of crawled pages per source per day, error counts, time since last crawl. Use console logs or a lightweight dashboard.
- **Batch geocoding** – Geocode addresses nightly using a bulk geocoding API; store lat/long for mapping. Perform route distance calculations offline.

Ultimate system (≈2–3 months)

- **Scalable architecture** – Deploy the Scheduler, Workers, and Storage as microservices on Kubernetes or serverless (AWS Lambda). Use distributed message queues (Kafka/SQS). Support thousands of URLs per minute.
- **Sophisticated scheduling** – Implement dynamic scoring based on historical change frequency, user interest signals, and error rates. Use machine-learning models to predict when a listing is likely to change and adjust crawl cadence accordingly. Implement per-domain rate-limiting buckets and polite crawling [4].
- **Robust change detection** – Use tree-based hashing and Firecrawl's git-diff/JSON diff to capture small text edits, price updates, availability changes, and image updates. Use perceptual hashing for images. Maintain full time-series histories for each field.
- **Incremental crawling** – Instead of re-fetching entire pages, use HTTP `If-Modified-Since` or `ETag` headers to fetch only when content changes [11]. Only parse changed pages. Persist state and only re-scrape new entries since last run [12].
- **Broad coverage** – Expand to dozens of property-management sites, listing aggregators, and community groups. Build custom adapters for sites requiring JavaScript rendering or anti-bot measures. Use proxies and rotate IPs to avoid bans.
- **Performance & cost optimization** – Cache responses from unchanged pages; dedupe early to avoid duplicate fetches [6]. Batch geocoding and route computations; precompute features used for search and ranking. Use incremental backups and compression for storage.

- **Observability** – Implement comprehensive dashboards with coverage, freshness, change detection rate, error budgets, and queue metrics. Use logs and traces to debug failures. Provide per-neighbourhood/price band coverage heatmaps to identify holes.
- **User experience** – Build a rich front-end with search filters, map view, price history charts, and alert preferences. Use asynchronous notifications (email, SMS, push) triggered by the Notification Engine when new matches appear.

Practical cost/performance tactics

Despite an unlimited budget, good engineering practices help reduce latency and cost:

- **URL deduplication and normalization** – Use hashing (e.g., SimHash) to avoid crawling identical pages [6].
- **Incremental crawling** – Use `If-Modified-Since`/`ETag` headers to fetch only updated content [11]. For API-based sources, use `updated_since` query parameters [13] to fetch only listings that changed since a certain time.
- **Partial parsing** – For sources with feed endpoints, request only metadata (e.g., JSON) rather than full HTML [14].
- **Exponential back-off & retries** – Implement retry policies that back off on failure to reduce wasted calls [15].
- **Batch geocoding & routing** – Instead of calling the geocoder for each address individually, accumulate addresses and call the geocoding API in bulk to take advantage of volume discounts.
- **Cache & CDN** – Cache images and static resources in a CDN to speed up repeated access. Use local caches to avoid re-downloading identical pages.

Conclusion

This design provides a robust, always-on discovery system tailored for the user's apartment-hunting needs. It balances freshness and coverage by using dynamic scheduling and high-velocity radar queues, employs efficient change-detection techniques to track price and availability changes, and includes practical cost/performance optimizations. Observability and metrics ensure the system remains reliable, while the roadmap shows how to evolve from a simple MVP into a comprehensive platform over a few months.

---

1 9 10 Change tracking | Firecrawl
https://docs.firecrawl.dev/features/change-tracking

2 8 An Efficient Algorithm for Web Page Change Detection
https://www.ijcaonline.org/archives/volume48/number10/7386-0173/

3 What is a web crawler and how does it work? Full guide. | Parallel Web Systems | Web Search & Research APIs Built for AI Agents
https://parallel.ai/articles/what-is-a-web-crawler

4 6 7 15 Design a Web Crawler System Design: A Complete Guide
https://www.systemdesignhandbook.com/guides/design-a-web-crawler-system-design/

5 11 14 What strategies do incremental web crawlers use when processing web links? - Tencent Cloud
https://www.tencentcloud.com/techpedia/115223

12 13 How to perform Incremental Web scraping
https://stabler.tech/blog/how-to-perform-incremental-web-scraping