Section 1: System Architecture Overview

AskTrippy is engineered to be a real-time, geo-aware, AI-enhanced travel search engine — not a chatbot — with the following principles:

- Scalable from 1 to millions with a microservice-ready, loosely coupled architecture

- Local-first development for fast iteration, low cloud cost, and tight control

- Free/open-source tech stack wherever viable to eliminate vendor lock-in and API costs

- Selective AI usage: embeddings, query understanding, summarization — not generation-heavy workloads

- On-demand scraping using a microservice powered by Trafilatura to enrich venues in real time

Core Tech Stack:

Frontend: Streamlit (MVP UI)

Backend: FastAPI, Python

DB: PostgreSQL + PostGIS + pgvector

Scraping: Trafitura

Hosting: Localhost for dev, Railway + Cloudflare for prod

Section 2: MVP Feature Set

• Semantic Recommender (pgvector): natural-language search via embedded queries

• Popularity Ranking: FSQ metadata (popularity_confidence, category_weight) as default sort

• Real-Time Scraping: triggered only when enrichment is stale or missing

• Background Pre Fetch: continuous, proactive scraping/caching job to keep high priority venues fresh (alongside real time triggers)

• Core Enrichment (MVP): schema.org parsing, targeted facts (hours/menu/contact), and LLM summary with source citations (Safe RAG)

• PostGIS filtering: radius/walk-time

• Streamlit UI: minimal MVP frontend

• Weather-aware recs: Open-Meteo API

Requirements & Constraints (MVP “done” bar)

Performance: cached responses <3s; queries that trigger scrape+enrich <10s end to end.

Freshness windows (defaults): hours 3 days; menu/contact 14 days; summary 14 days.

Triggers: on-demand scrape fires when required field is missing or past freshness window; background job continuously refreshes stale/high priority venues.

Pre fetch policy: prioritize top ~10% by popularity + any records past freshness; cap per host concurrency to avoid bans.

Scraper engine: Trafilatura microservice (HTML only); same host link finder fetches up to 3 target pages (hours/menu/contact/about/fees).

Safe RAG: enrichment must cite source URLs; LLM formats/extracts only from scraped/structured data (no invented facts).

Output shape: enrichment returns JSON serialisable facts (hours/menu/contact), summary (100–140 words), and sources (URL list).

Geo filters: PostGIS radius/walk-time must work with both cached and freshly scraped results.

Reliability: bounded retries, global + per host rate limits; no off domain fetching by link finder.

Operability: all features must run locally and in prod with the same defaults (configurable via env).

Section 3: Database Schema

The database must support **both** real-time and background crawling, while guaranteeing that all non-negotiable venue fields are stored with freshness and traceability.

Core Tables

1. venues – Baseline FSQ data + canonical venue-level facts

fsq_place_id (PK)

name

category_id / category_name

latitude, longitude (PostGIS point)

address_full, address_components (JSON: street, locality, region, postcode, country)

phone, email, website

price_range (if applicable)

popularity_confidence, category_weight

last_enriched_at (overall)

Indexes: PostGIS spatial, category, trigram on name, btree on last_enriched_at

2. scraped_pages – Every page fetched for a venue (homepage, menu, hours, contact…)

page_id (PK)

fsq_place_id (FK → venues)

url

page_type (enum: homepage, menu, hours, contact, about, fees, other)

fetched_at, valid_until (TTL)

http_status, content_type, content_hash

cleaned_text (fulltext)

raw_html (optional, compressed)

source_method (direct_url, search_api, heuristic)

Indexes: (fsq_place_id, page_type), content_hash unique

3. enrichment – Venue-level merged facts from scraping

fsq_place_id (PK, FK → venues)

hours (JSON, structured by day) + hours_last_updated

contact_details (JSON: phone/email/website/social) + contact_last_updated

description (text) + description_last_updated

menu_url + menu_items (JSON) + menu_last_updated

price_range (enum / string) + price_last_updated

features (JSON array) + features_last_updated

sources (JSON array of URLs for all facts)

Category-specific:

Accommodation: accommodation_price_range, amenities (JSON)

Attractions: fees, attraction_features (JSON)

Indexes: btree on each _last_updated field for freshness queries

4. crawl_jobs – Orchestrates both real-time and background jobs

job_id (PK)

fsq_place_id (FK → venues)

mode (realtime / background)

priority (0–10)

state (pending / running / success / fail)

started_at, finished_at, error

Indexes: priority, state, started_at

5. recovery_candidates – Optional table for inferred website discovery

candidate_id (PK)

fsq_place_id (FK → venues)

url

confidence (0–1)

method (email_domain / search / social)

is_chosen (bool)

Non-Negotiable Data Rules

Address, **contact details**, **opening hours**, and **description** must be present for all venues.

Category-specific must-haves:

Restaurants/cafés/bars: menu URL or items, price range.

Accommodation: price range, amenities.

Attractions: features, ticket/fee info.

All enriched fields must:

Include *_last_updated for freshness checks.

Carry at least one source URL from scraped_pages.

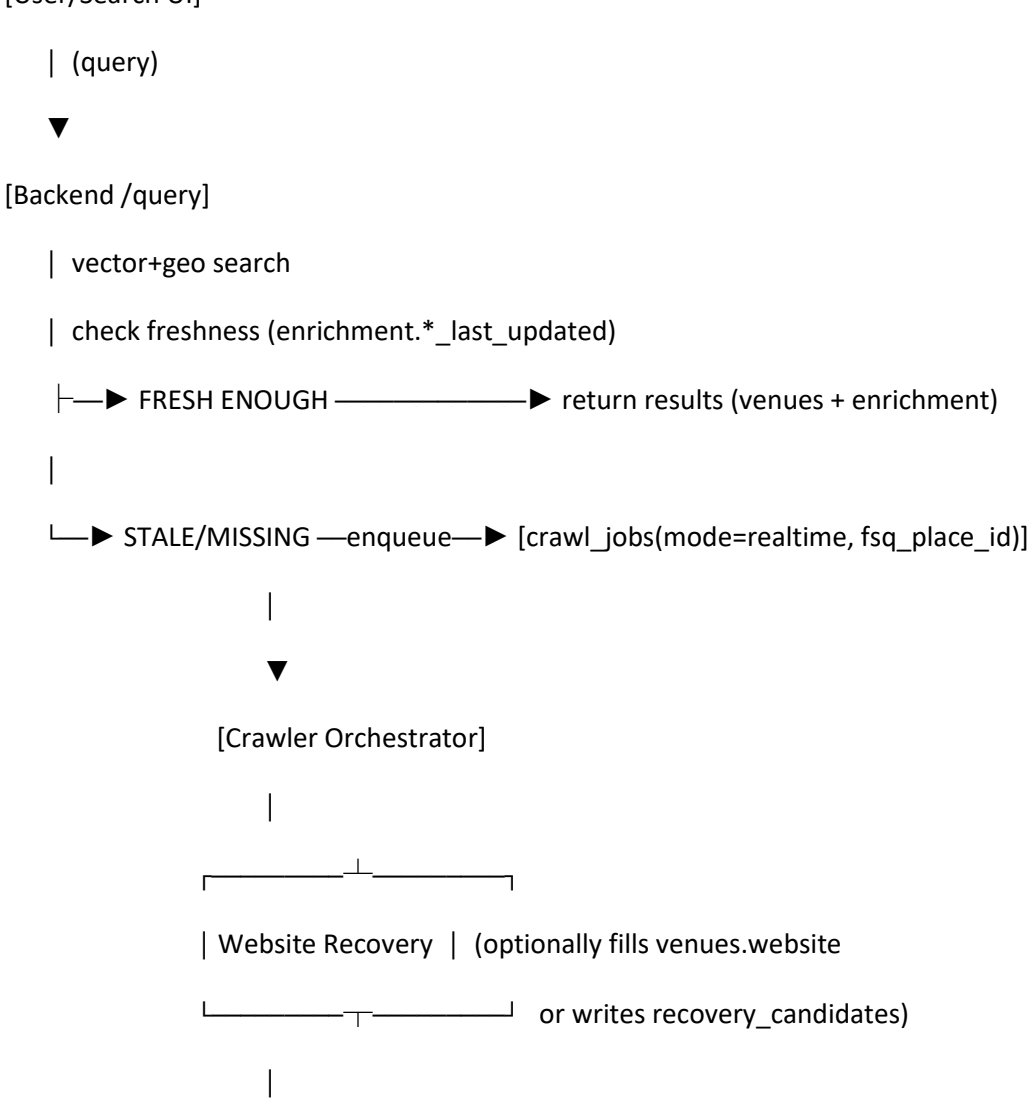
Explicitly flag not_applicable if data genuinely doesn't exist.

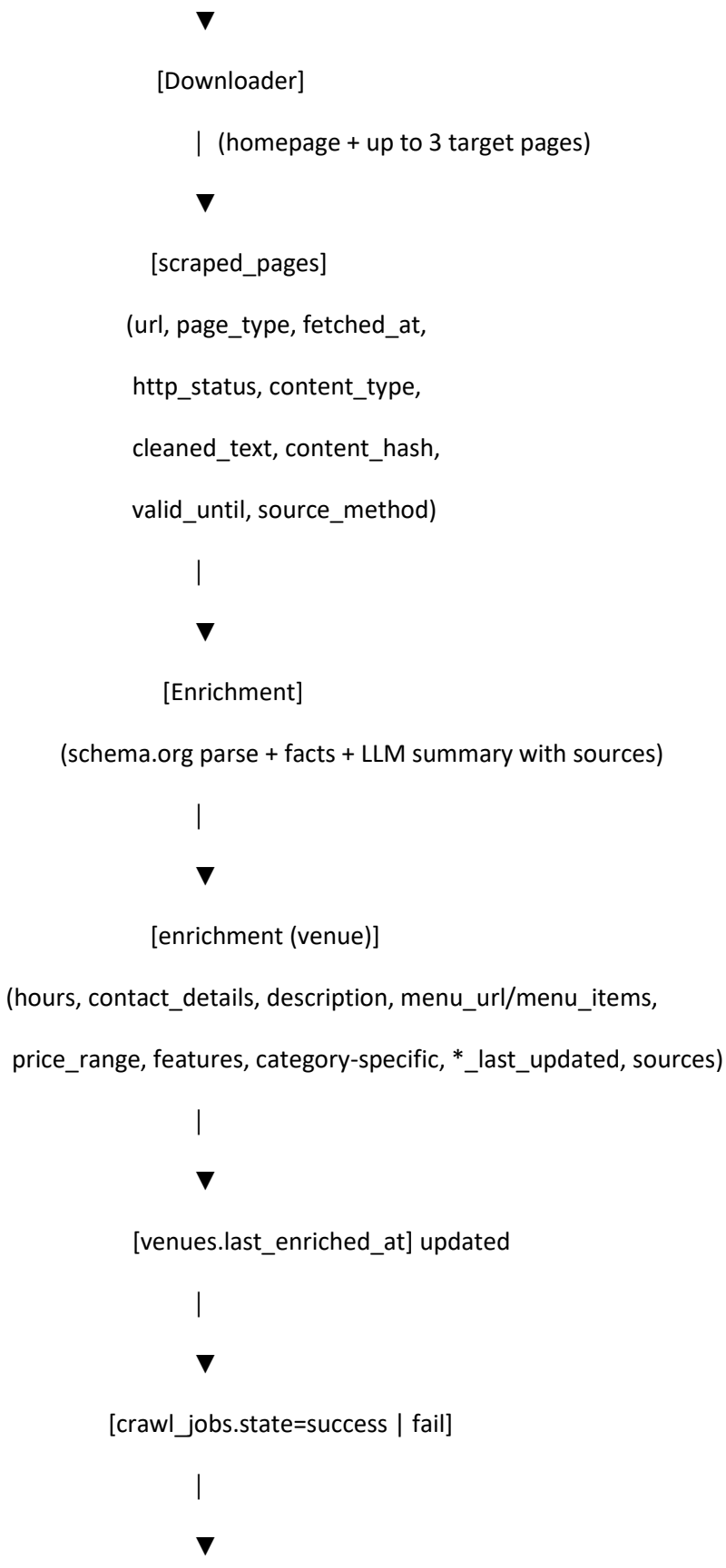
Freshness Windows (default)

hours: 3 days

menu, price_range, contact_details: 14 days

description, features, category-specific extras: 30 days




▼
[Downloader]
| (homepage + up to 3 target pages)
▼
[scraped_pages]
(url, page_type, fetched_at,
http_status, content_type,
cleaned_text, content_hash,
valid_until, source_method)
|
▼
[Enrichment]
(schema.org parse + facts + LLM summary with sources)
|
▼
[enrichment (venue)]
(hours, contact_details, description, menu_url/menu_items,
price_range, features, category-specific, *_last_updated, sources)
|
▼
[venues.last_enriched_at] updated
|
▼
[crawl_jobs.state=success | fail]
|
▼

[Backend /scrape/{job_id}] → UI poll completes

[Scheduler] —► select targets:

- enrichment fields past freshness windows
- high-popularity venues
- recent user interest areas

|

▼

enqueue ► [crawl_jobs(mode=background, priority)]

|

▼

(same pipeline as above: Recovery → Downloader → scraped_pages → Enrichment → enrichment → venues)

Key guarantees:

Traceability: every enriched field links back to one or more scraped_pages.url in enrichment.sources.

Freshness: per-field *_last_updated in enrichment + optional per-page valid_until in scraped_pages.

De-dup: scraped_pages.content_hash avoids storing identical content.

Dual mode: same pipeline for **realtime** and **background**, only the crawl_jobs.mode/priority differs.

Section 3: Database Schema & Crawling/Enrichment Data Flow (SSOT)

This section defines what we store and how the crawler/enrichment pipeline fills it for both real time (user initiated) and background (pre fetch) modes. It guarantees that all non negotiable fields are present, fresh, and traceable to sources. It also includes the API tool interfaces for interacting with and triggering these processes.

3A. Non Negotiables (what we must have)

Global (all venues)
• Address (full, structured)
• Contact details (phone/email/website)
• Opening hours (structured by day)
• Description (concise, factual)

Category specific
• Restaurants/Cafés/Bars: menu URL or items, price range
• Accommodation: price range, amenities/features
• Attractions/Museums/Sights: features, ticket/fee info

Rules
• Each enriched field stores at least one source URL (Safe RAG).
• Per field freshness timestamps (*_last_updated).
• If something truly doesn't exist, store an explicit not_applicable flag (never silent nulls).

3B. Data Model (authoritative tables & relationships)

venues — baseline FSQ POI + canonical venue level facts
• Identity: fsq_place_id (PK), name
• Classification: category_id, category_name
• Geo: latitude, longitude, locality, region, country_code, postal_code
• Contact & address: address_full, address_components (JSON), phone, email, website (canonical, HTTPS preferred)
• Ranking hints: popularity_confidence, category_weight
• Enrichment tracking: last_enriched_at (overall venue), optional freshness JSON (per field timestamps)

scraped_pages — one row per fetched URL (supports diffs, TTLs, audit)
• Keys: page_id (PK), fsq_place_id (FK → venues)
• Page: url, page_type ∈ {homepage, hours, menu, contact, about, fees, other}
• Fetch: fetched_at, valid_until (TTL), http_status, content_type, redirect_chain (JSON)
• Content: cleaned_text (Trafilatura output), optional raw_html (compressed), content_hash (dedup)
• Provenance: source_method ∈ {direct_url, search_api, heuristic}

enrichment — venue level merged facts (the "answerable" state)
• Keys: fsq_place_id (PK, FK → venues)

• Global:
o hours (JSON) + hours_last_updated
o contact_details (JSON: phone/email/website/social) + contact_last_updated
o description (text) + description_last_updated
o features (JSON) + features_last_updated
• Category specific:
o Restaurants/Bars: menu_url, menu_items (JSON, light), price_range + menu_last_updated, price_last_updated
o Accommodation: accommodation_price_range, amenities (JSON) + freshness timestamps
o Attractions: fees (tickets/pricing), attraction_features (JSON) + freshness timestamps
• Traceability: sources (JSON array of scraped_pages.url contributing to each field)

crawl_jobs — orchestration for realtime and background modes
• Keys: job_id (PK), fsq_place_id (FK → venues)
• Mode & priority: mode ∈ {realtime, background}, priority (0–10)
• State machine: state ∈ {pending, running, success, fail}, started_at, finished_at, error (text)

recovery_candidates (optional) — website discovery audit log
• Keys: candidate_id (PK), fsq_place_id (FK → venues)
• Fields: url, confidence (0–1), method ∈ {email_domain, search, social}, is_chosen (bool)

embeddings — semantic vectors (populated after Section 3 is implemented)
• Note: Included for completeness. No embedding until the crawl/enrichment loop is working and fields are present/fresh.

3C. Field Sourcing Matrix (where each field comes from)

Global (all venues)

Field	**Primary source(s)**	**Fallbacks / notes**
Address (full)	homepage footer/contact; contact page	FSQ baseline; schema.org PostalAddress
Contact details	contact page; homepage	schema.org telephone/email; verified social “About/Contact” if

Field	Primary source(s)	Fallbacks / notes
	header/footer	nothing else
Website (canonical)	Recovery (email domain, search, social) → homepage	Prefer HTTPS, non social; strip UTM/params
Opening hours	hours page; schema.org openingHoursSpecification	homepage/contact text blocks
Description	about page; homepage hero/intro	Meta description; schema.org description
Features/Amenities	about/homepage sections; schema.org amenityFeature	Parse lists/tables to JSON

Restaurants / Cafés / Bars

Field	Primary source(s)	Fallbacks / notes
Menu URL	menu page (nav/header/footer)	PDF/menu platforms if clearly linked
Menu items	menu page text (items + prices)	Keep minimal in MVP
Price range	Prices on menu; about “£/€/$”	schema.org priceRange

Accommodation

Field	Primary source(s)	Fallbacks / notes
Price range	rooms/rates; booking/rates links	schema.org priceRange; heuristic from text
Amenities	amenities/facilities/about	schema.org amenityFeature

Attractions / Museums / Sights

Field	Primary source(s)	Fallbacks / notes
Fees/Tickets	fees/tickets/visit	about/plan your visit; schema.org offers
Features	about/exhibitions/visit sections	schema.org features

Precedence & quality

• Same host only for target pages: hours, menu, contact, about, fees|tickets|visit.
• Precedence: dedicated page → schema.org → homepage/about/contact text → FSQ baseline.
• Quality gate: visible text ≥200 chars; block placeholders ("coming soon", "under construction"); content_type = text/html; HTTP 200.
• Sources: every enriched field lists ≥1 scraped_pages.url.

3D. Crawling & Enrichment Pipeline (mechanics)

FastAPI routes:

POST /query (core search)

POST /embed (embedding API)

POST /scrape (trigger Crawl4AI)

POST /rank (reranker)

GET /health (status)

Support optional Authorization headers and pagination (limit/offset).

Triggers

• Real time (user initiated): /query detects a required field is missing or stale → enqueue crawl_jobs(mode=realtime, priority=high).
• Background (pre fetch): Scheduler selects targets: past freshness, top popularity, recently searched areas → enqueue crawl_jobs(mode=background) with priority tiers.

Stages (shared by both modes)

Website Recovery (if venues.website is null)
o Heuristics: email domain → https://domain.tld; social → homepage link; light search API (brand + locality + category).
o Log all candidates in recovery_candidates; write chosen URL to venues.website.

Downloader (Trafilatura microservice + same host link finder)
o Fetch homepage + up to 3 target pages (same host): /hours|opening, /menu|food|drinks, /contact, /about, /fees|tickets|visit.

o Persist each page to scraped_pages with fetched_at, valid_until (per page type), content_hash, provenance.

Enrichment
o Schema.org parse: openingHoursSpecification, PostalAddress, telephone, priceRange, amenityFeature, offers.
o Targeted facts extraction: rules/regex + QA spans from cleaned_text (hours/menu/contact/fees/features).
o Summary (short, neutral): LLM formats only facts observed in scraped_pages/schema.org (Safe RAG).
o Unify & write: merge into enrichment with per field *_last_updated and consolidated sources (URLs used).

Venue & job update
o Update venues.last_enriched_at (and optionally a per field freshness JSON).
o Mark crawl_jobs.state to terminal (success/fail) with error if any.

Performance Contract (hard, non negotiable)
• Per site crawl wall clock ≤ 5 seconds (from first request start to last response end).
• Multi page fetch per crawl: homepage + up to 3 target pages (same host only).
o Target order: /hours|opening, /menu|food|drinks, /contact, /about, /fees|tickets|visit.
o Stop when 3 targets fetched or the 5 second budget is reached.
• Per page timeouts: connect ≤ 1s, first byte ≤ 1s, total read ≤ 1s, ≤2 MB response size.
• Parallelism: target pages may be fetched in parallel (up to 3 in flight) as long as total per site wall clock stays ≤5s and per host limits are respected.
• Budget overrun: if 5 second budget will be exceeded, abort remaining fetches, persist what's fetched, enrich with partial data, and record reason = time_budget_exceeded.
• Measurement: orchestrator logs started_at, ended_at, duration_ms, fetched_count, aborted_count.

Concurrency & politeness
• Per host concurrency cap (e.g., 2) and global cap (e.g., 32).
• Respect robots.txt; realistic UA rotation; exponential backoff with jitter on 429/5xx; bounded retries (e.g., max 2).

3E. Freshness & Scheduling

Default freshness windows
• Hours: 3 days
• Menu / Contact / Price range: 14 days
• Description / Features / Category specific: 30 days

Page TTLs (scraped_pages.valid_until)
• hours: now + 3 days
• menu, contact, fees: now + 14 days
• about/homepage: now + 30 days

Background selection policy
• Pick venues where any enrichment.*_last_updated breaches its window.
• Always include top ~10% by popularity_confidence.
• Boost venues from recent search hotspots (geo clusters from /query).
• Fairness guard: per area/category caps to avoid starving the long tail.

3F. Reliability, Errors & Limits

Error classes
• network_timeout, dns_failure, tls_error
• robots_disallowed, blocked_by_waf
• invalid_mime, non_200_status
• thin_content (fails quality gate), duplicate_content (same content_hash)
• off_domain_link (attempted cross host; must be blocked)

Retry/backoff
• Retry only transient errors (network/5xx/429) with jittered exponential backoff; max 2 attempts.
• Do not retry robots_disallowed, invalid_mime, off_domain_link.
• Every failure leaves a clear crawl_jobs.error and/or page level reason for debugging.

Safety & scope
• Same host rule is mandatory for target pages.
• Max 3 target pages per venue per crawl cycle.
• No infinite loops: every job must hit a terminal state.

3G. Traceability & Safe RAG

• Every enriched field cites ≥1 source URL in enrichment.sources, pointing to the specific scraped_pages.url used.
• The summary is formatting only (no invented facts), derived strictly from scraped/structured content.
• Off domain links are never fetched during the target page stage.

3H. Section "Done" Checks (acceptance for Section 3)

Measured on a 100 venue EU sample:

Coverage
• ≥80% venues produce non empty, quality gated scraped_pages.cleaned_text.
• ≥70% venues have at least one enriched field among: hours or contact or menu/price/features, each with source URLs.
• Category specific: restaurants/bars have menu_url or menu_items and price_range for ≥60% where menus exist.

Performance
• Per site crawl + enrich ≤5s (homepage + ≤3 targets).
• Queries that trigger a scrape return <10s end to end; cached queries <3s.

Freshness
• enrichment.*_last_updated respects windows; background scheduler refreshes stale entries.
• scraped_pages.valid_until set per page type and honoured.

Traceability & safety
• All enriched fields have source URLs.
• No off domain fetches; robots respected; retries bounded and logged.

Operability
• Realtime and background modes both function via crawl_jobs with clear states, priorities, and error reasons.

3I. (Reference) Minimal implementation map this section implies

backend/
crawler/
recovery.py # website discovery (heuristics + light search)

downloader.py # Trafilatura fetcher + robots + budgets
link_finder.py # same-host discovery: hours/menu/contact/about/fees
pipeline.py # orchestrates; writes scraped_pages; enforces 5s budget
enrichment/
schema_org.py # parse structured data
facts_extractor.py # targeted extraction (hours/menu/contact/fees/features)
llm_summary.py # Safe RAG summary from scraped/structured facts
unify.py # merge -> enrichment + sources + *_last_updated
quality/
html_gate.py # visible text, placeholder blocks, mime/status gates
jobs/
queue.py # enqueue, per-host/global concurrency, priorities
status.py # job state machine, errors
io/
read.py # freshness checks, venue lookups
write.py # persist scraped_pages, enrichment, job updates

3J. External Data Extensions (Roadmap, non-MVP)

Purpose: allow future integration of trusted, non-publisher datasets (e.g., **OpenStreetMap**) to *augment* venue facts when sites are thin or missing details—without changing any MVP guarantees in this section.

Candidate sources (examples)

OpenStreetMap (OSM): amenity, cuisine, wheelchair, opening_hours, outdoor_seating, contact:*, website, name:xx.

Transit (GTFS/GBFS): nearest stops/stations, headway hints (for “access” summaries).

Official registries: hospitals/clinics, embassies/consulates, tourism boards, municipality POI feeds.

Safety/accessibility lists: step-free access, accessible toilets, verified 24/7.

Ingestion (separate, optional)

Batch import to **aux tables** (e.g., ext_osm_poi, ext_transit_stop, ext_official_registry) with **source, license, last_updated** columns.

No change to scraped_pages/enrichment write path; external data is **joined**, not scraped.

Join & merge rules (only when helpful)

Match strategy: (a) spatial join (≤50 m) AND (b) fuzzy name/category match; keep match score.

Priority order: publisher site (Section 3 pipeline) → **official registry** → **OSM/other**.

Write-back: if adopted, store under **namespaced keys in enrichment** (e.g., access_info.osm_wheelchair), with:

*_last_updated

not_applicable where appropriate

sources[] including dataset + URL/license.

Conflict resolution: if external fact conflicts with a publisher fact, **publisher fact wins** unless an official registry explicitly overrides (record merge_reason).

Freshness & licensing

External tables carry their own **freshness windows** (default 90 days) and **license metadata**; facts are used only if license allows and data is within window.

Performance & safety

External lookups are **offline/batch**; no new latency on /query.

No change to Safe RAG: every adopted external fact must still be **cited** in enrichment.sources.

Note: This subsection is **non-MVP**. It documents how OSM/other datasets can be layered on **later** without altering the core guarantees and data flow defined in Section 3.

Section 4: Embedding & Semantic Logic (SSOT)

This section defines how both user queries and Points of Interest (POIs) are embedded and compared using semantic similarity search. It ensures embeddings remain fresh, cached where possible, and re-generated only when stale.

4A. Scope

Enable natural language search matching between queries and venues.

Support semantic ranking beyond simple keyword matching.

Ensure embeddings are stored, refreshed, and retrievable for real-time use.

4B. Embedding Targets

Queries – Embed raw search text input.

POIs – Embed category name + description (baseline), optionally including features, amenities, and menu highlights when available.

4C. Technology

Vector database: PostgreSQL with pgvector extension for similarity search.

Primary model: Hosted embedding API (e.g., text-embedding-3-small).

Fallback: Local Hugging Face model if hosted API is unavailable or cost control is required.

4D. Caching & Refresh

Store all venue embeddings in the embeddings table with a valid_until timestamp.

Cache query embeddings in memory for active sessions only.

Re-embed venues when content changes or when valid_until is exceeded.

4E. Rules

Use cosine similarity for vector comparison.

No re-embedding unless marked stale.

Embeddings must be linked to either a fsq_place_id (venue) or marked as query-only.

4F. Key Guarantees

Queries and venues are both embedded before comparison.

Search results are ranked by semantic similarity within geographic and category filters.

Fallback embedding model ensures continuity of service if the primary fails.

Section 5: UI Model (MVP)

5A. Scope & Goals

Fast, minimal Streamlit UI for local dev and demos.

Single search surface: natural-language query + radius + location.

Results list + map, with freshness-aware badges (Fresh / Stale → “Updating…”).

Zero hallucinations: every fact shows a source indicator.

5B. Core Screens & States

Search (default)

Inputs: query (text), location (autocomplete or “use my location”), radius (slider).

Actions: Search → POST /query.

Results

List + Map side-by-side (or stacked on mobile).

Each card: name, category, distance, brief description, opening status, price range, features, “Sources” link, “View on Maps” link.

Freshness badge per venue from enrichment.*_last_updated.

Stale/Missing state: show “Updating…” with spinner when a realtime crawl job was enqueued; poll /scrape/{job_id} until done, then soft-refresh the card.

Venue Detail Drawer (optional for MVP; can be a modal)

Hours (table), menu URL, features/amenities, contact info, source links.

Errors/Empty

No results, network error, robots blocked, etc. → friendly message + retry.

5C. Components

SearchBar: query input, location, radius, submit.

ResultCard: data from venues + enrichment, with source tags.

Map: OpenStreetMap (iframe is fine; better: Leaflet in Streamlit) with synced bounds.

JobStatus: per-venue polling widget (only appears when crawl enqueued).

Toast/Alert: for errors, rate limits, blocked_by_waf, etc.

5D. Data Flow (UI ↔ API)

Submit → POST /query with {query, lat/lon or place, radius}.

Backend returns: results (fresh) and/or {job_id}s for venues needing realtime update.

UI shows results immediately; for each {job_id}, poll GET /scrape/{job_id} until success|fail, then refresh only those cards.

External links:

OpenStreetMap embed for map.

Google Maps deep link (lat,lon or place_id) on “View on Maps”.

5E. UX Rules

Sub-3s for fresh hits; if scraping triggered, show partial results in <3s and background-update stale cards in-line.

Always show source count (e.g., “Sources: 2”). Clicking reveals URLs.

Open external links in new tab (your project default).

Respect small screens: map collapses below ~768px, list first.

Keyboard-friendly: focus states for inputs; Enter submits.

5F. Accessibility & i18n

WCAG AA color contrast for badges and links.

All interactive controls have labels/aria attributes.

Date/time (opening hours) localized to user locale; 24h vs 12h respected.

5G. Telemetry (minimal)

Log: query text length (not content), radius, location granularity, result count, scrape-trigger rate, time-to-first-results, time-to-fresh.

Store in Airtable/DB for learning loop (as you prefer).

5H. Performance Budgets

First paint ≤ 1s on desktop; initial query return ≤ 3s (fresh).

Incremental card refresh ≤ 1s after job success.

Map interaction under 16ms per frame (no jank).

5I. Folder/File Structure (Streamlit MVP)

```
frontend/
  app.py                # Streamlit entry
  components/
    search_bar.py
    result_card.py
    map_view.py
    job_status.py
    alerts.py
  services/
    api.py              # calls /query, /scrape/{job_id}
  styles/
    theme.py            # consistent spacing/typography
```

5J. Acceptance Criteria

I can type “late-night ramen near me”, set radius, hit Search, and see results in <3s if fresh.

Cards with stale/missing fields show “Updating…” and flip to fresh within the crawl SLA (your Section 3 budget).

Each card displays at least one source URL link.

“View on Maps” opens correct lat/lon or place_id in a new tab.

Works on mobile (list-first, map under list) without layout breaks.

Keyboard-only users can perform a full search and open a result.

Section 6: Deployment & Infrastructure (SSOT)

This section defines how the MVP runs locally and in the cloud, how configuration and secrets are managed, and the guardrails for cost, performance, and safety. It also specifies infra folders/files and the build/deploy order so this section is the single source of truth for ops.

6A. Environments & Footprint

Environments

Local Dev: Docker Compose (single host, all services).

Staging (optional): Same as prod topology on smaller instances.

Production: Railway (app containers) fronted by Cloudflare (DNS, CDN, WAF).

Core Runtime Components

FastAPI backend (stateless).

PostgreSQL (+ PostGIS + pgvector).

Crawler/Enrichment workers (same image, different command).

Streamlit UI (MVP).

Observability sidecars (logs/metrics).

Cloudflare for TLS, caching of static assets, basic WAF/rate-limits.

Data Stores

PostgreSQL local (Compose).

Cloud: managed Postgres can be **Supabase**, **Neon**, or **Timescale** (choose one per deployment).

6B. Requirements & Constraints

Performance & Reliability

P50 API latency (cached): ≤ 300 ms; P95: ≤ 1.5 s.

P50 API latency (with scrape on demand): ≤ 10 s end-to-end.

Service availability target (prod): 99.5% monthly.

Cold start limit (Railway dyno/container): < 5 s to ready.

Security & Compliance

All public traffic behind HTTPS (Cloudflare).

Secrets only via env vars/secret manager (never in repo).

CORS restricted to UI origin(s).

DB network access locked to app IPs (or via platform-provided secure proxies).

PII: none stored beyond anonymous session ids (if present).

Cost Ceilings (MVP)

Infra budget: **€10–€40 / month** all-in.

No paid proprietary vector DBs; **pgvector** only.

No paid external search APIs (dropped).

6C. Networking, Domains, TLS

DNS: Cloudflare manages apex and subdomains.

TLS: Cloudflare-issued certs terminate at edge; origin certs optional.

Routes

api.<domain> → FastAPI (Railway).

app.<domain> → Streamlit (Railway).

Caching: Only static assets via Cloudflare; **no** caching of API JSON by default.

6D. Rate Limiting & Scraper Throttling (Hard Guardrails)

API Rate Limits (Cloudflare → App)

Default: 60 req/min/IP per route group.

Burst: 120 req/min/IP for /query.

429 with Retry-After on breach.

Crawler Politeness (applies in all envs)

Global concurrent site fetches: **≤ 32**.

Per-host concurrency: **≤ 2**.

Backoff: exponential with jitter on 429/5xx; max 2 retries.

Robots.txt honored; user-agent rotation allowed.

Budget: ≤ 5 s wall-clock per venue (as defined in Section 3).

Content limits: ≤ 2 MB per page, content-type text/html.

These controls are mandatory and must be configurable via env vars (see 6G).

6E. Observability & Ops

Metrics (minimum)

Request rate/latency/error by route.

Crawl jobs: queued/running/success/fail counts; median wall-clock; abort reasons.

DB health: connections, slow queries, index bloat (periodic).

Scheduler actions: items selected per rule (freshness/popularity/geo).

Logs

Structured JSON logs for API and workers.

Correlation id propagated (X-Request-ID).

Alerts (prod)

API 5xx rate > 2% for 5 min.

Crawl job failure rate > 15% over 15 min.

DB CPU > 80% for 10 min or storage > 85%.

6F. Database Lifecycle

Vendors

Local: Postgres in Compose.

Prod: **Supabase**/**Neon**/**Timescale** (choose one; must support PostGIS + pgvector).

Migrations

Linear, timestamped migrations; applied on deploy gate.

Backups:

Local: nightly dump to volume.

Prod: provider snapshots daily + PITR if available.

Restore runbook: tested quarterly on staging.

6G. Configuration & Secrets (Environment Variables)

Global

APP_ENV ∈ {local, staging, prod}

API_BASE_URL, UI_BASE_URL

LOG_LEVEL ∈ {INFO, WARN, ERROR}

REQUEST_TIMEOUT_MS (default 10000)

Database

DATABASE_URL (includes PostGIS/pgvector)

DB_POOL_MIN, DB_POOL_MAX

Crawler

CRAWL_GLOBAL_CONCURRENCY (default 32)

CRAWL_PER_HOST_CONCURRENCY (default 2)

CRAWL_BUDGET_MS (default 5000)

CRAWL_PAGE_SIZE_LIMIT_BYTES (default 2_000_000)

Scheduler / Freshness (mirrors Section 3 defaults)

FRESH_HOURS_DAYS (3)

FRESH_MENU_CONTACT_PRICE_DAYS (14)

FRESH_DESC_FEATURES_DAYS (30)

Security

ALLOWED_ORIGINS

AUTH_ENABLED ∈ {true,false}

API_KEYS_* (if needed later)

All secrets set via platform secret manager; never committed.

6H. Deployment Topologies

Local (Docker Compose)

Single compose file spins up:

api, ui, worker (crawler/enrichment), scheduler (optional), postgres.

Hot reload enabled for API/UI.

Volumes for DB and logs.

Production (Railway + Cloudflare)

Separate services:

api (autoscale 0→N), ui, worker (N replicas), scheduler (1 replica), managed postgres.

Health checks:

/health (API/UI), queue depth (workers).

Horizontal scaling on CPU/latency for api, on queue depth for worker.

6I. Folder & File Structure (Infra Only; no app code)

```
infra/
  docker/
    Dockerfile.api
    Dockerfile.ui
    Dockerfile.worker
  compose/
    docker-compose.local.yml
  deploy/
```

railway.api.toml

railway.ui.toml

railway.worker.toml

cloudflare/

dns_records.md

rules_waf.md

rate_limits.md

config/

env.local.example

env.staging.example

env.prod.example

runbooks/

rollback.md

restore_db.md

incident_checklist.md

monitors/

alerts.yml

dashboards.md

6J. Build & Release Order (No Code, Just Steps)

Local First

Start Postgres (Compose) → run DB migrations → seed minimal data.

Bring up api, worker, ui via Compose.

Verify /health, DB connectivity, and a full query → scrape → enrich loop.

Provision Cloud

Create Cloudflare zone; set DNS for api. and app..

Create Railway services (api, ui, worker, scheduler) and managed Postgres.

Set secrets/env; restrict DB access to Railway.

Deploy Staging

Deploy images → run migrations → smoke tests (/health, sample query).

Validate rate limits and WAF rules.

Go Live (Prod)

Cut DNS to prod services behind Cloudflare.

Enable autoscaling thresholds.

Turn on alerts and dashboards.

Post-Deploy Checks

Verify crawl job throughput and freshness adherence.

Confirm cost telemetry (instance size, egress, DB storage).

6K. Acceptance ("Done" for Section 6)

One-command local bring-up (Compose) with green health.

Cloud stack reachable at api.<domain> and app.<domain>, TLS OK.

DB has PostGIS + pgvector; migrations applied cleanly.

Rate limiting and crawler throttles enforced and observable.

Backups enabled; restore runbook exists.

Alerts firing thresholds tested; dashboards show SLOs.

Costs within the stated MVP budget after 48 h burn-in.

6L. Decisions & Defaults

Cloudflare in front of everything for security and simplicity.

Railway selected for zero-ops container hosting (can swap later).

Supabase/Neon/Timescale: pick one per deployment; PostGIS + pgvector are non-negotiable.

No external search APIs; all enrichment relies on Section 3 pipeline.

No code in this section; infra is declarative and documented only.

Section 7: Success Metrics (SSOT)

Purpose:
Define measurable, objective criteria to assess Voy8's performance, reliability, cost efficiency, and trustworthiness. Metrics must be **quantifiable**, **repeatable**, and **monitored continuously** from launch using Postgres-backed logging.

7A. Core KPIs (MVP targets)

Metric	MVP Target	Post-MVP Target	Notes / Measurement Method
Query Latency (Cached)	< **3 s** end-to-end	< 2 s	Logged in Postgres at /query handler; measured at 50th & 95th percentile
Query Latency (With Scraping)	< **10 s** end-to-end	< 7 s	Includes realtime crawl + enrichment; measured at 95th percentile from crawl_jobs start/end
Relevance Score	> **80%**	> 90%	Manual human scoring of 100 random queries/month, using 5-point relevance rubric
Token Cost per Query	< **€0.001**	< €0.0005	Calculated from LLM usage logs stored in Postgres
Infrastructure Cost	< **€0.10** per 1K queries	< €0.05	Derived from cloud billing exports imported into Postgres
Trust Rate	> **95% verified outputs**	> 98%	Verified outputs = enriched facts with ≥1 source URL + within freshness window
Freshness Compliance	> **90%** fields within freshness window	> 95%	Checked nightly via freshness window rules from Section 3

Metric	MVP Target	Post-MVP Target	Notes / Measurement Method
Error Rate	< **5%** failed queries	< 2%	crawl_jobs + API logs tagged with error classes (Section 3F)

7B. Measurement Framework

Automated Monitoring (Postgres)

Latency: /query endpoint logs start/end timestamps for both cached and scraping flows.

Cost: Insert daily LLM token usage + infrastructure spend from cloud billing exports into metrics_cost table.

Trust Rate: Nightly Postgres job counts enriched fields with source URLs and freshness compliance.

Error Rate: Count crawl_jobs with state='fail' grouped by error class.

Human QA

Monthly random sample from Postgres query logs:

100 cached responses

100 realtime scrape responses

Manually score relevance (0–4 scale, ≥3 = relevant) and fact correctness vs. source.

Alerting

If any KPI falls below MVP threshold for >24 h, raise alert via logging hook (e.g., to Slack/email).

Critical trigger: Query Latency >15 s (scraping) or >5 s (cached) for >5% of queries in a 24 h window.

7C. Evaluation Cadence

Daily: latency, error rate, trust rate, freshness compliance.

Weekly: cost per query, infrastructure spend.

Monthly: relevance scoring (human QA), review of post-MVP stretch targets.

7D. Acceptance Criteria for MVP

All MVP target values met for **7 consecutive days** under simulated or real production load.

Human QA relevance score ≥80% for both cached and scraping modes.

No KPI exceeds MVP threshold for >24 h without remediation plan in place.

Section 8: Hugging Face Model Integration

This section defines the natural-language processing (NLP) components used in both query handling and enrichment. Models are integrated locally (no API calls to Hugging Face Hub in production) to maintain performance, cost control, and data privacy.

8A. MVP Scope (Must-Have)

Semantic Search & Embedding

Model: sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2 (384-dim multilingual sentence embeddings).

Purpose: Vectorise both user queries and POI text (category + description) for similarity search using pgvector.

Policy: Embeddings are cached and only re-embedded if source data changes (stale).

Named Entity Recognition (NER)

Models:

English: dslim/distilbert-NER

Multilingual: Davlan/xlm-roberta-base-ner-hrl

Purpose: Real-time extraction of named entities (LOC, ORG, PER) from queries to improve POI matching and enrichment lookups.

Safe RAG Alignment

LLMs are **not** used for content generation in MVP.

All answers must be grounded in scraped/structured data from Section 3.

NLP models support understanding and retrieval only — no hallucination risk.

8B. Post-MVP / Phase 2 Enhancements (Optional)

Semantic Reranking

Model: cross-encoder/ms-marco-MiniLM-L6-v2

Use: Reorder vector search results for improved precision.

Summarisation

Model: sshleifer/distilbart-cnn-12-6

Use: Generate concise venue summaries from scraped/structured content.

Intent Detection

Model: facebook/bart-large-mnli

Use: Zero-shot classification of queries to improve routing.

Translation

Model: MarianMT (e.g., Helsinki-NLP/opus-mt-mul-en)

Use: Translate non-English queries to English and optionally translate answers back to the user's language.

Personalisation

Approach: Vector-based preference scoring (cosine similarity with past interactions).

Longer-term: Sequential models (e.g., Transformers4Rec) once usage data is sufficient.

Section 9: Risk Strategy & Economic Defense

Voy8 is designed to be sustainable, affordable, and reliable from the start — even under rapid growth. The architecture deliberately avoids high-cost APIs, minimises LLM usage, and prevents runaway compute or scraping failures through strict triggers, status tracking, and performance constraints.

9A. Risk Categories & Mitigations

Risk	Mitigation
High API costs	Prefer free/low-cost datasets and APIs; use local semantic search (pgvector) instead of paid vector services.
Runaway scraping loops	Section 3 pipeline enforces explicit triggers, per-job state tracking, scrape attempt limits, and per-host rate limits.
Data staleness	Background enrichment service refreshes stale fields on schedule; freshness windows per field defined in Section 3.
LLM hallucinations	Limit LLM usage to formatting and summarisation of known facts (Safe RAG), never fact generation.
Scaling pressure	Stateless microservices deployable on scalable cloud hosts (e.g., Railway, Fly.io); Postgres supports horizontal read scaling.
Overengineering	Strict MVP scope; build only critical, real-world functionality before expanding.

9B. Cost Structure (MVP Estimates)

Component	Dev Mode	Prod Mode (Est.)
PostgreSQL	€0 (local)	~€25/month (Neon, Supabase, or Timescale)
Embeddings	€0 (local HF)	~€5–10/month (if using hosted small model)
Scraper	€0 (local/VPS)	€0–€5/month (VPS hosting)
App Hosting	€0 (local)	~€5–15/month (Railway or similar)
Total	€0	~€10–€40/month fully operational

Costs assume no high-volume commercial API calls and that embeddings are only recomputed when source data changes.

9C. Economic Defensibility

Free/low-cost data + local processing = no usage-linked cost explosion.

One-time embeddings = minimal LLM billing.

Modular design = features can be scaled, disabled, or tiered by user role without breaking the core product.

Minimal baseline infrastructure ensures predictable monthly costs even as query volume scales.

Section 10: Executive Summary

Voy8 (formerly AskTrippy) is a new class of AI-powered travel product — not a chatbot, not a booking site, but a live intelligence engine that solves problems as they happen.

Core innovation comes from combining:

Free and open structured data at scale (e.g., Foursquare OS, OpenStreetMap, official registries, other open APIs)

A high-speed, safe, scraping-based enrichment pipeline (Section 3)

Local semantic search via pgvector

A FastAPI-based modular backend

A mobile-ready MVP UI in Streamlit (local dev) with scope to migrate to a production UI framework

This yields a product that:

Responds in real time to changing conditions (closures, service disruptions, changes in opening hours)

Provides meaningful suggestions that can be acted on immediately

Runs at extremely low cost and scales cleanly to millions of users

Voy8 is not built to wow with generic AI fluff — it is engineered to help you find a nearby, trusted solution when your train is delayed, your museum is closed, or your dinner plans fall through.

It works locally, works offline, works with real-world data — and is designed to scale from MVP to full production without architectural rewrites.

Section 11: Implementation Timeline & Milestones (Crawler-First SSOT)

Principle: No enrichment, embeddings, or UI work proceeds until the crawler reliably produces quality scraped_pages within the performance budget defined in Section 3.

11A. Phase 0 — Prereqs (already done)

POI baseline loaded (FSQ OS EU subset).

PostgreSQL with PostGIS + pgvector initialised.

11B. Phase 1 — Crawler Core (Weeks 1–2)

Goal: Consistently fetch **homepage + up to 3 same-host target pages** per venue within **≤5s** wall-clock and pass quality gates.

Deliverables

Website Recovery (email-domain + social → homepage) writing to venues.website and recovery_candidates.

Downloader (Trafilatura) with:

Per-page timeouts (connect ≤1s, TTFB ≤1s, read ≤1s, ≤2 MB size).

Same-host link finder for /hours|opening, /menu|food|drinks, /contact, /about, /fees|tickets|visit.

Parallel fetch (≤3 inflight) while staying under ≤5s per site.

Persistence to scraped_pages (with fetched_at, valid_until, http_status, content_type, cleaned_text, content_hash, source_method).

Job orchestration:

crawl_jobs queue + worker (states: pending → running → success|fail).

Per-host concurrency cap (2) and global cap (32).

Retries on transient errors only (max 2, jittered backoff).

Acceptance Gate (must pass before Phase 2)

On a 100-venue EU sample:

≥80% rows with non-empty, quality-gated cleaned_text.

Median per-site crawl ≤3s; 95th percentile ≤5s.

No off-domain fetches; robots respected; errors logged with classes from Section 3F.

Stop-the-line rule: If the gate fails, **do not** proceed. Fix crawler until it passes.

11C. Phase 2 — Enrichment (Week 3)

Goal: Convert scraped pages into venue-level facts in enrichment with **traceable sources** and **freshness timestamps**.

Deliverables

Schema.org parser: openingHoursSpecification, PostalAddress, telephone, priceRange, amenityFeature, offers.

Facts extractor (rules/regex + QA spans) for hours, contact, menu URL/items (light), price range, features, fees.

Unifier writing to enrichment with per-field *_last_updated, sources[] (pointing to scraped_pages.url), and not_applicable flags where appropriate.

Optional LLM formatter limited to **Safe RAG** (formatting only; no new facts).

Acceptance Gate

On the same 100-venue sample:

≥70% have **at least one** enriched field among: hours OR contact OR (menu/price/features), each with ≥1 source URL.

Restaurants/bars: menu_url or menu_items **and** price_range for ≥60% where menus exist.

All fields have *_last_updated; sources[] populated.

11D. Phase 3 — API Layer (Week 4)

Goal: Expose the pipeline safely; **/query** never blocks on long work.

Deliverables

POST /query — vector+geo search, freshness check; if stale → enqueue crawl_jobs(mode=realtime) and return partial results + job_ids.

POST /crawl — explicit trigger; enqueue jobs, return job_ids.

GET /crawl/{job_id} — poll job state; include any updated_fields.

POST /embed — internal: create/cache embeddings (only after enrichment exists).

GET /health — liveness/readiness (DB ok, queue depth, avg crawl duration).

Acceptance Gate

Cached query p95 <3s; queries that trigger realtime crawl p95 <10s end-to-end.

Responses include freshness flags and source counts where applicable.

11E. Phase 4 — UI (Week 5)

Goal: Minimal Streamlit interface for demo/testing, freshness-aware.

Deliverables

Search bar (query + location + radius).

Result cards (summary, hours, price, features, source links) + map (OSM embed/Leaflet).

Inline "Updating…" state for venues with active job_ids; poll /crawl/{job_id} and refresh card on success.

Acceptance Gate

End-to-end demo: query → (enqueue if stale) → crawl/enrich → card updates to fresh within SLA.

Mobile viewport layout acceptable; keyboard accessible.

11F. Parallel/Background Tasks (Weeks 3–5)

Background scheduler for stale fields (Section 3E).

Metrics/telemetry: latency, failure classes, freshness compliance (rolling daily).

Rate limits + WAF rules (Section 6).

11G. Definition of "MVP Done"

All Phase 1–4 gates met for **7 consecutive days** under test load.

Section 7 KPIs achieved: cached <3s, scraping <10s p95; trust rate >95% (verified outputs); error rate <5%.

Cost envelope within Section 9 estimates.

Section 12: Long-Term Vision & Scaling

AskTrippy is not just a travel search engine — it's a real-time, modular travel intelligence platform with the potential to power consumer apps, enterprise tools, and embedded travel UX in third-party systems. Its architecture is designed for longevity, flexibility, and viral growth.

🧭 Geographic Expansion Strategy

1. Europe First: Leverage strong public transport APIs, high POI density, strong mobile usage, and familiar booking flows

2. Asia-Pacific: Target digital-native travelers in Japan, South Korea, Thailand, and Australia

3. North America: Expand into urban metro markets with high local travel demand

4. Offline Countries: Support preloaded city modules for regions with poor connectivity

🧱 Platform Expansion Opportunities

- Consumers: Premium subscriptions for offline access, saved searches, mobile-first UX

- Enterprises: Real-time APIs for travel management companies and OTAs

- Local governments: Crowd-smart visitor flow tools, enrichment dashboards

- Tourism boards: Embeddable live recommendation maps

- Transport operators: Delay-aware itinerary nudges, nearby POIs

🧠 Viral Growth Vectors

- Crisis resolution: "AskTrippy saved my trip" moments are naturally shareable

- Hidden gem discovery: Users share unexpected finds driven by context-aware suggestions

- Travel influencer coverage: Solves real-world research problems in seconds

- Mobile-first UX: Built to work on the go, especially in uncertain moments

- Offline packs: Unique edge for train travelers and low-connectivity trips

🛡️ Technical Scaling

- FastAPI: Stateless, container-friendly; scale horizontally via Railway or Fly.io

- PostgreSQL: Upgrade to managed Supabase or Timescale with read replicas

- pgvector: Continues to scale with CPU or GPU-backed vector ops

- Crawl4AI: Moves to task queue + pool model for parallel scraping jobs

- Frontend: Can be rewritten in React Native or Flutter for unified mobile/web app

- Data sync: Venue data + enrichment stored in portable backup layer (daily JSON dumps or Postgres replica)