AskGWINnett

Engineering Design Doc

A production retrieval-augmented civic-information system for Gwinnett County, Georgia — grounded chat over public records, with every answer traced to its source.

Brian Foster · askgwinnett.com
August 2026 · v0.13.0

SECTION 1

What it is

AskGWINnett is a chat-first web app that answers plain-language questions about Gwinnett County — property taxes, schools, parks, traffic, voting, and the actions of the Board of Commissioners and the school board. The county publishes this information, but it is scattered across a dozen portals and PDFs. AskGWINnett routes each question to the right public data source, generates a grounded answer, and attaches the source so the reader can verify it.

The design goal is trust, not just convenience. The system never answers from the language model's own memory: every reply is generated from data retrieved at query time, verified against that data by an independent review step before it is shown, and cited. It supports four languages (English / Spanish / Korean / Vietnamese — Gwinnett has one of the largest Korean-American communities in the U.S. and Georgia's largest Vietnamese-speaking population) via a translation-pivot design covered in Section 6, and runs as a real production service with per-IP rate limiting, source citations, and a golden-test suite that gates every change.

29	3-tier	1,317	107	4
SPECIALIZED TOOLS	QUERY ROUTING	VECTORS INDEXED	GOLDEN EVAL CASES	LANGUAGES

SECTION 2

Technology stack

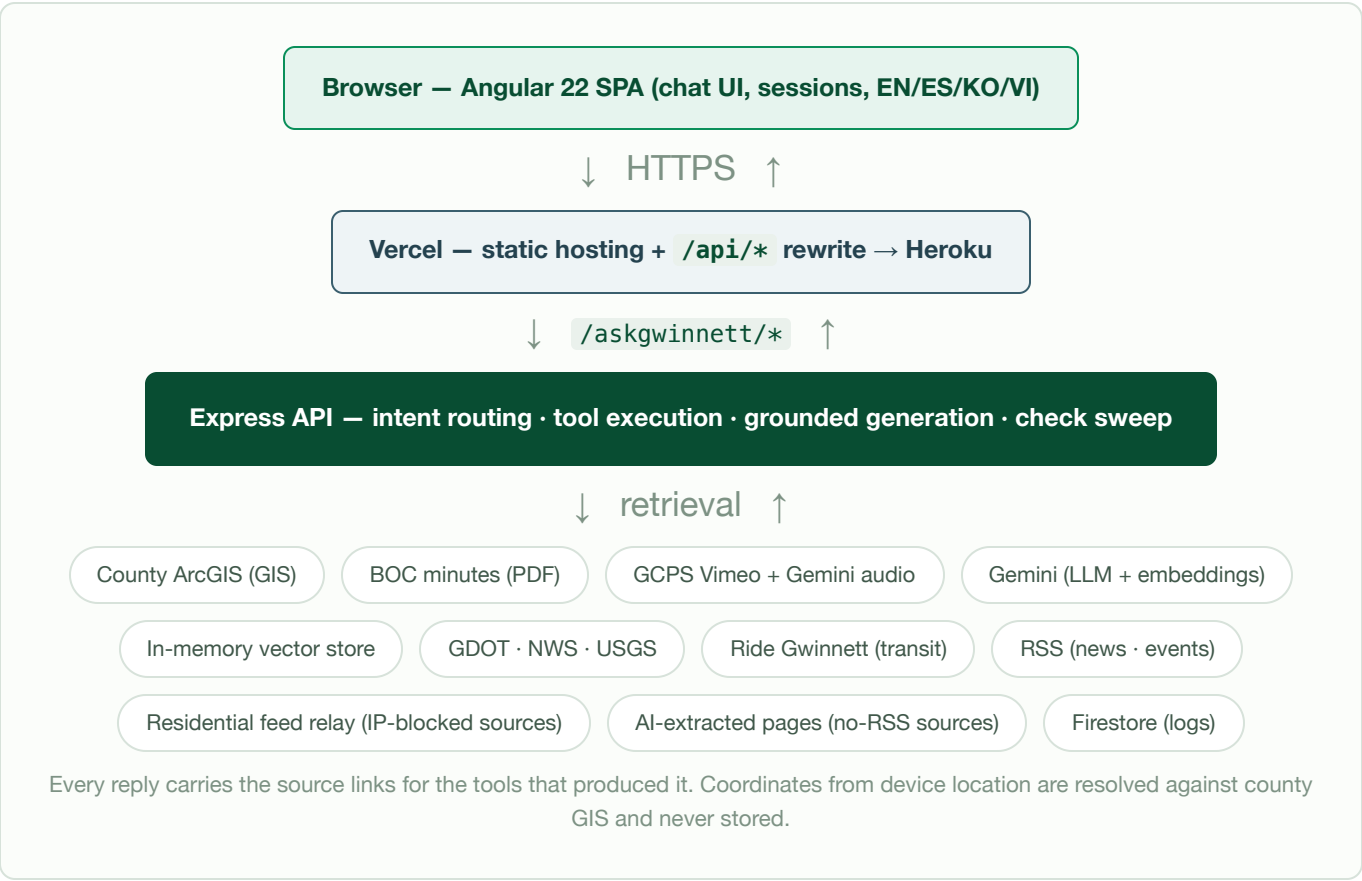
Layer	Technology	Role & rationale
Frontend	Angular 22 (standalone components, Signals), TypeScript, RxJS	Reactive chat UI with signal-based state; no NgModules. Signals drive the message thread, session store, and i18n dictionary.
Internationalization	Typed EN/ES/KO/VI dictionaries + translation pivot	A single typed dictionary object; TypeScript enforces that Spanish, Korean, and Vietnamese stay in sync with English at compile time. Chat answers in the resident's language come from the translation pivot (Section 6), not per-language pipelines.
Frontend host	Vercel (static + edge rewrites)	Serves the built SPA; rewrites `/api/*` to the backend and everything else to `index.html`. Cookieless Web Analytics.
Backend	Node.js 20, Express 4	37-module API mounted on an existing personal services app; each county capability is its own module.
AI — generation	Google Gemini (`gemini-3.1-flash-lite`), pinned	Tool-calling model, fixed server-side so behavior never depends on a client-selected model. Migrated from `gemini-2.5-flash` ahead of its Oct 2026 retirement — the model-agnostic architecture meant the swap was an eval run, and it landed ~5× faster and ~40% cheaper. Meeting ingest runs the stronger `gemini-3.5-flash`.
AI — retrieval	Gemini `gemini-embedding-2` (256-dim)	Embeds the meeting corpora for semantic search; pinned to match query and document vectors.
Data / logging	Firebase Firestore	Fire-and-forget chat logging, feedback storage, and thumbs-up/down tallies. No user accounts.
Email	Nodemailer (Gmail)	Typed feedback and 👎 reports emailed to the operator for triage.
Scraping / parsing	cheerio, pdf-parse, pdfkit, ffmpeg	HTML table scraping, minutes-PDF text extraction, downloadable report generation, meeting-audio extraction.
Security	express-rate-limit, CORS allowlist, abuse cool-down	Per-IP throttling (burst + a 75-question daily cap on chat) and origin locking so a viral post can't run up the AI bill, plus a deterministic abuse

		guard: repeated prompt-injection or abuse patterns from one IP earn a polite temporary 429 with no model call. Admin routes add a failed-auth lockout (10 misses per IP per 15 minutes) and fail closed when the key is unset. Pattern-based by design — enforcement is auditable and never rides on model judgment.
CI / automation	GitHub Actions	Weekly scrape → extract → embed → commit → auto-deploy, plus curated-data drift checks.

SECTION 3

System architecture

Two repositories, two deploy targets. The Angular app (`county-app`) is a static bundle on Vercel; the API (`bfoster-services`) is an Express app on Heroku. The browser never talks to Gemini or any county source directly — everything passes through the backend, which holds the API keys, enforces rate limits, and keeps coordinates inside county data sources.



Every reply carries the source links for the tools that produced it. Coordinates from device location are resolved against county GIS and never stored.

SECTION 4

Anatomy of one question

Every chat turn follows the same pipeline. The whole conversation is sent per request (there are no server sessions), so history travels with the message and nothing about the user is retained server-side. The diagram below is the full path through `bfoster-services` — guards, prep, the hybrid routing cascade, tools, grounded generation, and the check sweep.

1. **Client → API.** The Angular app POSTs the full message thread (plus language and optional one-time location) to `/askgwinnett/chat` via the Vercel `/api/*` rewrite.

2. **Guards.** CORS allowlist, Origin/Referer host gate, per-IP rate limits, and a deterministic abuse cool-down run before any model call. Failures return 403/429 — no LLM spend.

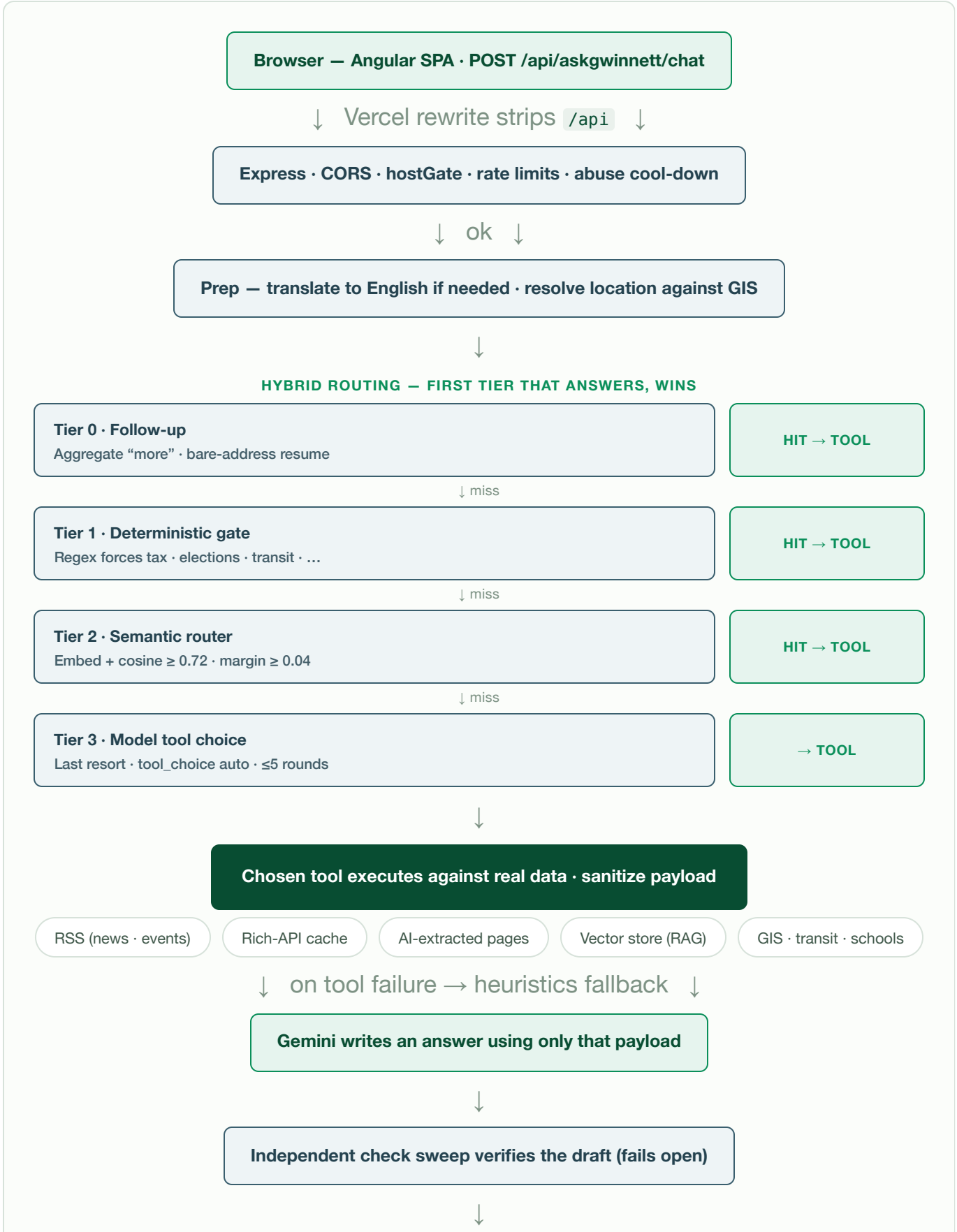
3. **Prep.** Non-English questions translate to English (pivot); optional device coordinates resolve against county GIS in parallel (nearest address + city-limits point-in-polygon).

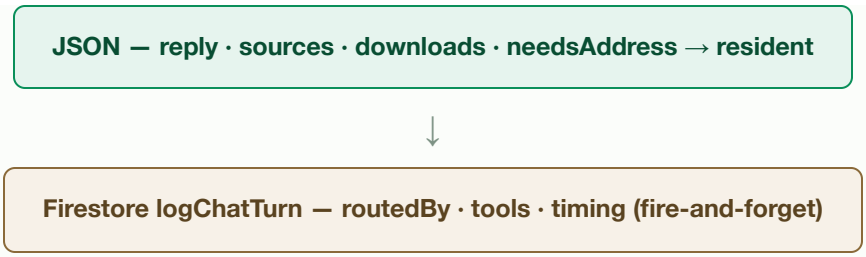
4. **Hybrid routing.** A cascade — follow-up → deterministic regex → semantic intent → LLM tool choice — stops at the first confident hit. Logged as `routedBy`.

5. **Tool execution.** The chosen tool runs against a live feed, curated data, or the vector store (≤5 rounds), returning a structured JSON payload with source URLs. On tool-router failure, a heuristics fallback still attempts an answer.

6. **Grounded generation + check.** Gemini writes a plain-language answer from *only* that payload; a second fixed judge verifies the draft against the tool data (fails open). Optional report downloads attach when a meeting-report intent is detected. One class of question skips this step entirely: routes whose answers are *code-rendered* from the structured payload (currently English public-comment questions) short-circuit before any model call — zero LLM spend, deterministic output, ~100ms.

7. **Response + telemetry.** JSON returns with source chips (and downloads when present); the turn is logged to Firestore fire-and-forget for later quality triage.


Browser — Angular SPA · POST /api/askgwinnett/chat
↓ Vercel rewrite strips /api ↓
Express · CORS · hostGate · rate limits · abuse cool-down
↓ ok ↓
Prep — translate to English if needed · resolve location against GIS
↓
HYBRID ROUTING — FIRST TIER THAT ANSWERS, WINS
Tier 0 · Follow-up
Aggregate “more” · bare-address resume
HIT → TOOL
↓ miss
Tier 1 · Deterministic gate
Regex forces tax · elections · transit · …
HIT → TOOL
↓ miss
Tier 2 · Semantic router
Embed + cosine ≥ 0.72 · margin ≥ 0.04
HIT → TOOL
↓ miss
Tier 3 · Model tool choice
Last resort · tool_choice auto · ≤5 rounds
→ TOOL
↓
Chosen tool executes against real data · sanitize payload
RSS (news · events)
Rich-API cache
AI-extracted pages
Vector store (RAG)
GIS · transit · schools
↓ on tool failure → heuristics fallback ↓
Gemini writes an answer using only that payload
↓
Independent check sweep verifies the draft (fails open)
↓



One question's path through bfoster-services. Routing is a cascade, not a parallel fan — the first confident tier wins and the rest never run.

Where the AI actually is: judgment at the edges, procedure in the middle

Every component in the pipeline is one of three kinds, and the boundaries are deliberate:

Kind	Where
Pure procedure	The tier-1 routing ladder and the body of every tool — SQL against ArcGIS, HTTP to live feeds, JSON reads, keyword scoring. Cannot hallucinate; same inputs, same behavior, unit-testable.
Model as math	The embedding calls in the tier-2 router and RAG retrieval. A neural model, but no judgment: it converts text to 256 numbers (pinned model — same text, same vector), and plain cosine arithmetic against calibrated thresholds does the deciding. A deterministic hash for meaning.
Model as judgment	Exactly four places: the translation pivot in, tier-3 tool choice (only when tiers 1–2 abstain), answer generation from the tool payload, and the check sweep judging that answer. Each is bounded: fail-open, last-resort, or verified by another.

The consequence: the model phrases the question and phrases the answer — it never fetches the facts. So failures skew toward "retrieval didn't find it" (fixable, testable, honest) rather than "the model invented it," which is the failure mode the entire architecture exists to prevent.

A location grant resolves jurisdiction, not just an address

When a resident shares device location, the coordinates resolve against county GIS twice, in parallel: nearest address point (for parcel, commissioner, and hauler lookups) and a point-in-polygon test against the county's city-limits layer. The second matters because a mailing city is not a jurisdiction — most "Lawrenceville" addresses are unincorporated — and whose millage applies to a tax bill depends on actual city limits. The millage data carries per-jurisdiction rates for all 18 taxing jurisdictions (with school-share percentages *precomputed*, so the serving model presents arithmetic rather than performing it), which turns "what percent of my tax bill is school taxes?" from a hedged generic answer into a direct personalized one.

SECTION 5

Tool routing — a 3-tier cascade

The core design rule: **answers must not depend on which language model is used.**

General tool-calling lets the model decide which function to invoke — but weaker or cheaper models make that decision inconsistently, which for a civic-information site means unreliable answers. So routing is a cascade of three tiers, each a safety net for the one above. A question stops at the first tier that answers confidently.

```
# Each question is routed to exactly one tool, in priority order.
function routeToTool(question):
    tool = deterministicGate(question)      # tier 1 — regex, locked routes
    if tool is None:
        tool = semanticRouter(question)     # tier 2 — route by meaning
    if tool is None:
        tool = askModelToChoose(question)   # tier 3 — the LLM decides
    return tool
```

1. **Tier 1 — deterministic gate.** Regular-expression matchers detect a known intent and *force* the correct tool. Fast, free, and frozen: high-stakes routes (tax, elections, school board) can never be decided by a guess. Every routing bug found becomes a permanent regression test here.

2. **Tier 2 — semantic router.** When no regex fires, the question is routed by *meaning* rather than keywords (Section 6a). Catches paraphrases the regexes miss — "I'm looking for work with the county" has no job keyword yet routes to the jobs tool.

3. **Tier 3 — model tool-choice.** When even the router isn't confident, the language model picks the tool itself, as in any tool-calling system. The last resort, not the default.

Why the cascade order matters

Determinism where it counts, flexibility where it's safe. "Why hasn't my county tax bill changed?" is pinned by regex to the homestead-exemption data every time, regardless of model or temperature. A novel phrasing about parks falls to the semantic router. Only the genuinely unclear reaches the model. Each tier can only *add* a correct route, never break the tier above — so the system gets more flexible without getting less reliable.

Inside tier 1: how the deterministic gate works

Tier 1 is a chain of small **regular-expression matchers** — one per intent. A regular expression (regex) is just a text pattern: it either matches the words and phrasings in a question or it doesn't — no model, no probability.

Each matcher is a plain function; the first one that matches wins, and its tool is forced through the API's `tool_choice`, bypassing the model's judgment entirely.

THE DETERMINISTIC GATE, IN CONCEPT

```
# One matcher per intent — a pure text test, no AI involved.
isMillage(q)   = /\b(millage|property tax|tax rate|homestead|value offset)\b/
isElections(q) = /\b(register to vote|polling place|absentee|sample ballot)\b/
isSchool(q)    = /\b(gcps|school board|superintendent|when.*school start)\b/
# ...one matcher per safety-critical route (not per tool)

deterministicGate(q):
    for matcher in ordered matchers:
        if matcher(q): return matcher.tool   # first match wins → force it
    return None                              # nothing matched → fall to tier 2
```

Two properties make this the right mechanism for high-stakes routes. **It's deterministic:** the same question always produces the same route, on any model at any temperature, so a tax question can't be talked into the wrong tool. And **it's testable:** each matcher is unit-tested, and every routing bug becomes one more pattern plus a permanent regression case — so the gate only hardens over time.

Field test: the forced tool got the whole prompt as its argument

A prod transcript showed a location-granted "taxable value of my home?" failing on the first turn while the typed address succeeded — and the quoted address differed only as "Ln" vs "Lane," which pointed everyone at street-suffix normalization. The suffix theory was disproven directly (the county GIS stores "LN"; every suffix form matched), and the real cause was in the forcing path itself: when a tool is executed server-side, its `address` argument was the *entire question* — user text plus the injected location hint — which matched nothing. The fix follows the codebase's own precedent: every address-taking tool now runs the same deterministic address extractor over whatever arrives, so a forced call, a model-mangled argument, and a clean typed address all resolve identically. The lesson generalizes: **when a symptom has an obvious cosmetic explanation, verify it against the data layer before fixing the wrong thing.**

Where two intents could collide, the matchers carry explicit **fences**. "What party is my commissioner?" and "what party does the county vote for?" both say *party* — so the electorate-lean matcher excludes questions naming an official (that's a roster lookup), and the jobs matcher excludes "apply to *vote*." Each of those fences started as a real misroute that, once found, became a permanent test.

What's actually inside a matcher

Each matcher is a tiny function that returns yes or no. Here is a real one — the jobs matcher — showing the fence in place:

```
function isJobsQuery(text):
    # FENCE: a voting question is NOT a jobs question — bail out first
    if text has (voter | vote | ballot | register to vote): return false
    # the real test: characteristic job words
    if text has (job | hiring | employment | careers | openings | apply for a job):
        return true
    return false
```

That's the whole black box: an early *fence* that rules out look-alikes from other subjects (so "apply to *vote*" can never reach jobs), then a list of characteristic words. No score, no accumulation — it matches or it doesn't. Up in the router it's one line, `forceJobs = isJobsQuery(question)`, and that yes/no drops into the priority chain where the first "yes" wins.

Match vs. rank — the two jobs people conflate

A frequent point of confusion: **routing** and **retrieval** are different stages that use different methods. Routing picks the *tool*; retrieval — inside a tool that holds many records — picks *which records* to show. Three matching styles appear across the system, and it clarifies everything to see them side by side:

Style	Result	What it does
Regex match	yes / no	**Route.** Does this pattern appear? → pick the tool (Stage 1).
Keyword score	a number	**Rank.** How many query words appear in each record? → order the records inside a tool (e.g. which meetings to surface).
Embedding similarity	a number	**Route or rank by meaning**, ignoring the exact words → the semantic router (Stage 2) and RAG retrieval.

The first two are word-based (they read the literal words typed); the third is meaning-based. And Stage 2 never runs a tool to decide — it only compares the question's vector to the pre-written *example* vectors, pure math; tools execute only *after* one is chosen. A "tie" that makes Stage 2 abstain is simply two intents from different tools scoring nearly the same — no clear winner, so it defers to Stage 3.

Design rationale: why deterministic-first — and what we traded for it

The principle is to route by certainty: use the most certain mechanism that applies, and fall back to less certain ones only when the certain ones abstain. Regex is the most certain and cheapest thing available, so it leads. The alternative — letting the model pick a tool on every question — is nondeterministic (the same question can route differently across runs, temperatures, and model versions), untestable (you cannot unit-test "usually right"), and model-dependent, which violates the core rule that answers must not depend on which LLM runs. For tax, elections, and school-board routes, "usually right" is not a defensible standard; a deterministic rule gives a guarantee that can be regression-tested.

The ordering follows cost and certainty, like a cache: the fast, certain check runs first, and expensive or fuzzy work happens only on what's left. LLM-first would pay latency, cost, and nondeterminism on every trivial query while discarding the guarantees. Semantic-first would embed every question and route by *similarity* —

which always returns a nearest neighbor, and so can confidently misroute where an exact rule would either match or cleanly abstain.

THE TRADEOFFS WE ACCEPTED — KNOWINGLY

The pitfall of deterministic-first	What it bought / how it's bounded
Brittleness & upkeep — regexes don't generalize; new phrasings need new patterns.	Reliability on the routes that matter. Bounded by tier 2, which now absorbs novel phrasings, so a missed pattern degrades gracefully instead of failing outright.
Over-eager matching — a broad rule can hijack a question ("party" → electorate vs. a named commissioner).	Explicit fences per matcher, and every misroute becomes a permanent eval case, so the same mistake can't recur.
No real understanding — pattern-matching can't handle synonyms or genuinely new intents.	Exactly what tiers 2 and 3 exist for. The deterministic gate is never asked to be complete — only certain.
Upfront cost — ~22 matchers plus fences is more work than "let the model decide."	Reproducibility and a regression-tested guarantee. The cost shrinks over time as the semantic layer absorbs phrasing churn.

Why the trade nets out ahead

Two structural facts make it lopsided in our favor. **Each tier can only *add* a correct route, never break the tier above** — so adding flexibility never erodes the deterministic guarantees. And **a routing mistake degrades to "less complete," not "wrong"** — the check sweep grounds every answer against its source regardless of how it was routed, so a misroute cannot become a confident fabrication. No single layer has to be perfect; three imperfect layers cover each other's weaknesses, with a fact-checker underneath them all.

The 29 specialized tools

Each tool owns one capability and returns structured data plus its own source links. The router never blends an official record with a news article or community opinion into a single unsourced answer — provenance is preserved per claim.

Government & records

searchMeetings (BOC minutes + recording summaries, RAG) · searchSchoolMeetings (GCPS, RAG) · searchTaxManual (GA DOR Digest Manual, RAG) · getSchoolBoardInfo · getMillageInfo · getElectionsInfo · searchOfficials · findStateLegislators · searchContacts · getCrimeInfo (jail lookup + crime-map status + FBI stats, all link-outs — individuals are never looked up) · getCensusInfo (demographics)

Live feeds

searchTransit (Ride Gwinnett real-time, plus closest-stop-to-me by distance from a device location or address) · searchTraffic (GDOT) · searchRoadProjects · searchPowerOutages · searchWeather (NWS + USGS quakes) · searchLocalNews (press + several cities' own announcements) · searchLocalEvents (city calendars, Agenda Center meeting postings, plus AI-extracted coverage for sources with no feed at all)

Property & place

lookupParcel (ArcGIS zoning/value) · searchParks · searchNearbyPlaces (Google Places v1 — results carry each business's website plus, for Gwinnett restaurants, the official GNR health-inspection lookup link; scores are never stated, only linked)

Daily life

getSolidWasteInfo (trash/recycling) · getWaterInfo (utilities, outages, quality) · getWarmingStationsInfo (heat/cold relief) · getJobsInfo (county careers link-out) · getCitiesInfo (all 17 municipalities, incl. Mulberry) · getLibraryInfo (all 15 branches, hours, cards) · getClubsInfo (where social groups actually organize) · getSourceInfo (what an outside org/data source AskGWINnett cites actually is)

DESIGN NOTE

How the cascade came to be — and whether it's standard

The three-tier router was not drawn up front; it **evolved**, by hitting real problems in order — each one forced by a single requirement.

1. **Start.** The normal approach — let the model choose the tool (LLM function-calling).
2. **Problem.** Cheaper and weaker models chose inconsistently, breaking the rule the product rests on: *the answer must not depend on which model runs*. → Added deterministic regex forcing for the high-stakes routes.
3. **Next problem.** A new regex for every edge case — maintenance whack-a-mole. → Added the semantic router to absorb the phrasings the regexes miss.
4. **Underneath all along.** The model's own tool choice, as the final catch-all.

So the shape fell out of solving successive problems, each pointing back to that first constraint — **model-independence** — which is exactly what puts determinism first. The tiers after it exist to win flexibility back without surrendering the guarantee.

Copied a playbook, or derived it?

Both, in the way that matters. The individual pieces are established, named patterns — rule-based intent routing, embedding "semantic routers," and LLM function-calling. The hybrid cascade (deterministic → semantic → model, ordered by certainty) is itself an increasingly common shape. But it was reasoned out from the constraints and *converged* on the established pattern, not lifted from it — independent derivation landing on the standard is validation, not imitation. Honest scope: this is the right architecture for *these* constraints — trust-critical, model-agnostic, high-stakes topics — not a universal verdict. A general assistant on a strong model could reasonably just use plain tool-calling; the extra machinery earns its keep here because the trust does.

Field test: "any model train shops in Gwinnett?" — no such shops, said the bot

The system replied that no model-train shops exist in the county — while Legacy Station (Lilburn), B&B Hobby (Grayson), and TrainMaster Models sat in Google's data the whole time. The chain of failure was deterministic: the query classified as generic *shopping*, the keyword extractor recognized only cuisines, so Google was never asked about trains — it browsed mall and grocery *types*, and the model accurately described those useless results. The guardrails again converted missing retrieval into a confident false negative.

The fix (which rode the Places API v1 migration): specialty subjects — "X shops," "where can I buy X" — are now extracted deterministically and sent as a single natural-phrasing *text search*, un-restricted by venue type, since specialty retail lives under types the browse lists don't cover. County-wide asks ("anywhere in Gwinnett") widen the radius and rank in-county results above higher-rated shops across the county line. Same doctrine as Mulberry: **the root cause of a false "doesn't exist" is almost always retrieval, not generation.** A fourth instance ("wineries" — a venue noun with no shop/store word) ended the pattern-per-noun cycle for good: when no extractor matches but the query carries specific tokens, the resident's own cleaned words now go to Google as the search — parsing natural language is Google's job, and nouns never seen before ("batting cages") work on the first try.

Field test: "is the city of Mulberry real?"

A real user asked this and the system *denied a city of ~40,000 people existed*. Mulberry incorporated in 2024 — too new for training-data confidence, absent from that week's news feeds — so the question routed to news search, found nothing, and the anti-hallucination rules ("never invent county facts") converted an empty feed into a confident false negative. Instructive failure: the guardrails that usually protect accuracy manufactured the error.

The fix is the philosophy in miniature. **The root cause was missing data, not missing routing** — so the cure was a curated tool (all 17 municipalities, verified against the county's own page, with Mulberry's incorporation, charter, and pending litigation labeled by date). And routing went through the *semantic tier*, not a new regex: nine example questions in the intent file. The router then caught phrasings never written as examples ("is mulberry georgia actually a city?" at 0.91 similarity) — which is precisely what regex cannot do. Working rule, now enforced in the codebase comments: **the deterministic tier is reserved for safety-critical routes; everywhere else, fix misroutes by adding exemplars, not regexes.**

SECTION 6

The translation pivot — one pipeline, any language

Spanish, Korean, and Vietnamese support does not duplicate the pipeline per language. Non-English questions are translated to English at the front door, the entire English machinery runs unchanged, and the answer is generated back in the resident's language.

The naive approach — making every tool multilingual — fails structurally: the regex matchers, the intent exemplars, the amenity aliases, and the county data itself are all English. A Spanish question like "*¿qué parques tienen piscinas?*" would need "piscinas" taught to the parks matcher, and the same again for Korean and Vietnamese, for every keyword, in every tool. Instead:

1. **Pivot in.** The UI sends the selected language with each request. For non-English, a single Gemini call translates the question to English — with strict instructions to keep street addresses, proper nouns, and numbers *exactly* as written. Fails open: if translation errors, the original text proceeds.
2. **Run everything unchanged.** Routing (all three tiers), keyword tools, RAG retrieval, and location enrichment all see clean English. One set of exemplars, one set of matchers, one copy of the data.
3. **Answer back.** The system prompt — not the question — carries the directive: call tools with English arguments, then write the entire reply in the resident's language, keeping addresses, names, phone numbers, and URLs as-is, ending with a machine-translation note pointing to the English sources.
4. **Check against the original.** The independent review step receives the resident's original question, so its language-consistency rule judges the reply in the language actually requested.

> **A lesson paid for in debugging: steering lives in the system prompt**
>
> The first implementation appended the answer-language directive to the question text. Result: Spanish pool queries returned *nothing* — while Korean worked. The model had copied the *entire question string, directive included,* into the search tool's argument, so the parks tool received a paragraph of instructions as its query and matched zero parks (Korean survived by phrasing luck). The rule this bought: **per-request steering goes in the system prompt; the question string stays clean**, because models echo question text into tool arguments nearly verbatim. A companion fix hardened the tools themselves: search-term tokenizers now drop filler words ("where", "are", "the"), since the model rephrases the same request many ways and a required-match on filler zeroes out results.

SECTION 7

Retrieval-augmented generation (RAG)

The whole system is RAG-shaped — retrieve, then generate from what was retrieved. Inside the meeting corpora, retrieval is upgraded from keyword matching to semantic vector search.

The problem it solves

Residents speak in plain language; government records are written in procurement-speak. Ask "did the board approve buying bedding for the jail?" and keyword search finds nothing — the word "bedding" appears in no record. Semantic search finds the answer anyway, because the *meaning* matches an approved contract for "transparent vinyl covered mattresses for the Sheriff's Department." That gap between how people ask and how records are written is the normal case, not the edge case.

How it's built — deliberately without a database

1. **Chunking.** Board-meeting extracts are already coherent units — one chunk per action, plus one per meeting summary. No arbitrary text splitting; the structure of the record *is* the chunking.

2. **Embedding.** At ingest time, each chunk is embedded once with the pinned `gemini-embedding-2` model at 256 dimensions, L2-normalized, and packed as base64 `Float32` into a committed JSON file — **1,208 chunks** across three corpora: BOC minutes, GCPS meetings, and the Georgia DOR Digest Manual (Section 7b).

3. **Vector store.** On startup the vectors load into memory as typed arrays. At this scale (a few hundred to a few thousand chunks) brute-force cosine similarity over memory is mathematically identical to a vector database and adds no infrastructure — a Postgres/pgvector add-on only earns its keep at tens of thousands of chunks.

4. **Retrieval + blend.** The query is embedded (with a 2.5-second timeout and small cache), scored by cosine against every chunk, and the best-matching meetings receive rank-based boosts that are *added to* keyword scores. The keyword floor keeps exact names, dates, and dollar figures deterministic; the semantic layer only lifts what keywords miss. Any failure — missing key, timeout — degrades cleanly to keyword-only.

RETRIEVAL, IN CONCEPT

```
function retrieve(question):
    q = embed(question)                    # 256-number "meaning" vector
    for each chunk in meetingCorpus:
        sim  = cosine(q, chunk.vector)     # semantic closeness, 0..1
        score = keywordScore(chunk)         # exact names/dates/$ — reliable
              + semanticBoost(sim)          # lifts what keywords miss
    return top-scoring meetings          # keyword floor + semantic lift
```

The blend is the point: keyword scoring keeps exact facts deterministic, while the semantic boost rescues paraphrases. Embedding failure (timeout, missing key) simply drops the boost term — the system degrades to keyword-only, never breaks.

How numbers become text — there is no decode step

The mental model that unlocks RAG: **a vector is a one-way fingerprint.** The embedding API returns a plain array of 256 numbers to the server — nothing is "processed on Google's side" beyond that — and those numbers can never be translated back into text. They are only ever *compared*. The reason a text answer still comes out is that the original text never left: every entry in the committed store holds the record's text and its vector *side by side*. One question flows through four steps:

1. **Embed the question** (API call #1) → 256 numbers land in a variable. Think of them as coordinates in a meaning-space, not a pointer.
2. **Compare locally** — plain cosine arithmetic in Node against all ~1,300 stored vectors. No API, no lookup-by-address: every stored dot is measured, the nearest few win. The output is only a *selection*: "these entries are relevant."
3. **Pull the winners' text** — the pre-written record text sitting next to each winning vector, extracted from the official source at ingest and frozen in the repo. The numbers didn't create it; they pointed at it, like a catalog number points at a shelf without containing the book.
4. **Generate** (API call #2, a different model) — the selected *text* goes into the generation prompt and the model writes the reply from it. This call never sees an embedding number.

One line: **numbers choose the facts; the model words the facts.** The consequences fall out directly — a retrieval miss degrades to "the nearest chunks weren't very near" (weak recall), never a corrupted answer; the facts pre-exist as text, so the model can only phrase, not invent; and the whole middle step is free local math, which is why a RAG turn costs one cheap embedding call plus one generation call.

7a · Aggregate queries: when top-k retrieval is the wrong tool

Retrieval-by-relevance answers *lookup* questions ("who won the pickleball contract?") but structurally cannot answer *enumeration* ("a full list of contracts the board approved this year") — the answer is spread across every meeting, and top-k selection returns one meeting's slice presented as if it were the whole. The literature splits these as **local vs global** queries (the self-query / metadata-filter retriever lineage; GraphRAG's local-vs-global search), and the fix is routing, not better ranking: enumerative questions ("all / every / list / how many" + a category noun) skip the vector store entirely and run a **deterministic scan** over the structured extracts — filter actions by category, dedup work-session previews against the Business Session approval record by item ID, sort largest dollar amount first.

The scan finds ~244 distinct contract/award actions in 2026 — far too many for one readable reply, and asking a small model to enumerate hundreds of JSON items drops entries nondeterministically. So the list is **paginated in code**: the model receives exactly one page of twenty, and the server — not the model — appends a footer to the finished answer: "*Showing 1–20 of 244 contract/award actions … say 'more' to see the next 20.*" Server-side appending guarantees the partial-coverage disclosure on every model, and the footer doubles as a machine-parseable cursor: a bare "more" reply is detected deterministically, the footer from the previous turn is parsed for the original query and position, and the next page is force-fetched — the same server-side forcing pattern as the bare-address follow-up. Counting, filtering, deduplication, and paging all happen in code; the model only phrases the page it is handed.

The cross-model eval matrix caught a real hole before launch: weaker models rewrite the tool argument ("full list of contracts this year" → "contracts"), which no longer trips aggregation detection. The resident's *original* question therefore rides alongside the tool call and takes precedence over whatever the model wrote — after which all four models in the matrix, including the weakest, pass both pagination cases.

Planned: a dedicated vector store as it grows

The in-memory store is a deliberate *current* choice, not a permanent one. At a few thousand vectors, brute-force cosine is optimal and a database would be pure overhead. As the corpus grows — deeper meeting-archive backfill, a news corpus, years of records reaching tens of thousands of chunks — the plan is to move the vectors into a real store (`pgvector` on the existing Postgres stack) for indexed nearest-neighbor search, completing the standard production RAG stack. Retrieval already sits behind a single function, so that swap changes *where the vectors live*, not how the rest of the system works. Building it before the scale exists would be premature; the architecture is ready for it the moment the need is real.

Provenance is the contract

Retrieval knows which source produced each candidate, so answers label official minutes, news coverage, and community discussion separately and link each. The value-add is connecting related items across sources; the failure mode — deliberately designed against — is blurring where each fact came from.

7b · The third corpus: a reference document, not a meeting record

The Georgia DOR **Digest Submission Manual** (2026) — the state rulebook behind property taxation — joined the store as a reference corpus: homestead exemption statutes and income limits, Social Security maximums, conservation-use and freeport programs, classification and appeal reason codes. Chunking again follows the document's own structure (one chunk per page, titled from its table of contents), and questions like "what's the income limit for the senior homestead exemption?" now answer with the actual statutory figures, labeled as state guidance — not legal advice — with county practice pointed at the Tax Commissioner.

The ingest had a wrinkle worth recording: 36 of the manual's 103 pages — including its highest-value memos — are scans with no text layer. The pipeline handles them with a Gemini OCR pass (the PDF uploaded once via the Files API, pages addressed by physical position, per-page retries for stragglers), bringing 99 of 103 pages to full text. The 27MB source PDF is never committed; re-ingest is a one-command annual step when the next edition publishes.

6a · The same machinery, for routing

The embedding infrastructure built for RAG does double duty. The **semantic router** (tier 2 of Section 5) applies the same idea to *intents* instead of documents: a handful of labeled example questions per tool is embedded once and stored; an incoming question is embedded and matched to the nearest example by meaning. One shared embedding function serves both — the same code that finds the right meeting also finds the right tool.

SEMANTIC ROUTING, IN CONCEPT

```
# 106 example questions across 23 intents, embedded once (committed).
function semanticRouter(question):
    q = embed(question)
    best = intent whose example is closest to q     # highest cosine
    if best.similarity >= 0.72                        # confident enough, AND
       and beats the runner-up tool by >= 0.04:      # a clear winner
        return best.tool
    return None                                       # unsure → defer to the model
```

The confidence gate is the safety. The thresholds weren't guessed — they were calibrated against real questions: genuine questions score 0.85–1.00, off-topic ones ("tell me a joke") score ~0.62, and 0.72 sits in

the empty gap between. The router acts only when sure and stays silent otherwise. It is pinned to one embedding model (preserving the model-agnostic rule) and fails open.

Staying silent — deferring to Stage 3 — means one of three things: nothing cleared the floor, the top two intents were a near-tie (no clear winner), or the embedding call errored. All of it is maintained *by hand*: like the regexes, the example questions are human-edited and grown reactively from real misroutes (found in testing or from a 👎). One build-time gotcha follows from matching vectors instead of text — a new example question is *inert until the exemplars are re-embedded*, a deliberate step, not a live update.

SECTION 8

Case study: turning meeting videos into searchable records

The Board of Commissioners publishes minutes as PDFs. The school board's minutes system is behind a web-application firewall and unscrapeable — but the district posts full **video** of every meeting to a public archive. That video is the only accessible complete record of what the board actually did, so the pipeline transcribes it.

1. **Enumerate.** The public video showcase is listed programmatically using the ownership token the page itself mints for the video platform's API — title, date, duration, and URL per meeting.
2. **Extract audio.** ffmpeg pulls the audio stream (optionally time-compressed 1.5× to cut transcription cost by a third — verified to preserve extraction quality, with timestamps rescaled back to real time).
3. **Transcribe + structure.** The audio is uploaded to Gemini's Files API and processed in one pass into a structured extract: actions, votes, appointments, discussions, public-comment topics, and superintendent items, each with an approximate timestamp. The drift-checked board roster is passed in as a spelling reference.
4. **Guardrails.** Ambiguous votes must be recorded as "not clear from the recording" rather than guessed; public comment is summarized by topic and never attributed to named private citizens. Every extract is stamped as a machine summary of the official recording — *not* official minutes.
5. **Index + serve.** New extracts are embedded into the same in-memory vector store and served through `searchSchoolMeetings`, mirroring the BOC path. Answers link the source video and cite approximate timestamps.

The same pattern applies to any future corpus. Cost per meeting is a few cents; the weekly automation ingests new meetings with no manual step, while deeper backfill of the archive stays an explicit, cost-capped manual decision — exercised once so far: a ten-meeting backfill reaching to March 2026 that brought in all five district town halls, the FY2027 budget work sessions, and the special called meeting appointing the new superintendent, for roughly a dollar.

The pattern's second use: BOC public comments

The written BOC minutes record *no* public-comment content — for months the system could only say so. The video pipeline closed the gap: the county's meeting recordings are transcribed and structured the same way as the school corpus, producing per-topic comment summaries (anonymized — "a resident," never a name) across 2026's meetings. Questions like "has anyone complained about the surveillance cameras?" now answer from an **aggregate built across all meetings** rather than one meeting's slice, with the recording linked and a server-appended provenance note making clear the summaries come from the official recording, not the minutes.

SECTION 9

The quality system

What separates this from a tutorial chatbot: the machinery that keeps answers correct as the system grows.

Mechanism	How it works
Check sweep LLM-as-judge grounding + intent	Before any answer is shown, a fixed judge model (temperature 0, strict JSON) verifies the draft against the tool payload it was generated from. If a claim isn't supported by the data, it's corrected. A CRAG-inspired rule extends grounding to intent: a claim that something *doesn't exist* must be backed by an authoritative source covering the category — empty or off-topic search results only ever justify "couldn't find," never "there are none." Fail-open by design.
Golden eval harness 80 deterministic cases	A self-hosted test suite spins up the server and asserts real answers against expected substrings and patterns. Every fixed bug becomes a permanent case; every new capability ships with its own cases. The suite gates changes — including the model migration itself, which shipped only after a fully green run on the new model.
Drift checks Curated-data monitoring	A weekly job re-fetches the official pages behind curated data (board roster, school counts, calendar, first day of school) and fails loudly if the published facts no longer match — turning static curation into a self-alerting snapshot.

When generation loses its vote: code-rendered answers

The public-comment route earned a stronger guarantee than prompting can give. In production, the serving model *denied* comments that sat in its own tool payload ("could not confirm…"), a payload note failed to fix it, a question-level nudge failed too — and worse, the judge's rewrite path *invented* public commenters that never existed, because a rewrite is generation like any other. Two structural rules came out of that day. First, the **two-strikes rule**: after a payload note and a prompt nudge both fail, stop escalating prompts — render the answer in code from the structured payload, because prompt fixes only improve odds while structural fixes give guarantees. Second: **judges may only remove or approve against the data — never conjure replacement prose.**

English public-comment answers are now assembled entirely by code from the aggregate — scoped by date when the question names a meeting, scoped by topic via the question's content words, and answering "how many times…" with meeting/entry counts plus an honest granularity caveat (one summary can cover several speakers). Since code-rendered output has no generation step to verify, the route also short-circuits *before* the model and the judge: two LLM calls become zero, ten seconds become a tenth of one, and the answer is bit-for-bit reproducible. The template for future high-stakes routes: when a route's payload is structured and its answer needs no per-user judgment, code rendering beats prompt engineering on every axis that matters here.

THE CHECK SWEEP, IN CONCEPT

```
function answer(question, toolData):
    draft   = generate(question, toolData)        # uses tool data only
    verdict = judge(draft, toolData)              # pinned model, temp 0
    if verdict finds a claim not in toolData:
        draft = corrected(draft, toolData)
    return draft + source links              # fail-open if the judge errors
```

The data-source strategy is a deliberate taxonomy: **live feeds** for volatile data (transit, traffic, weather, outages), **curated + drift-monitored** snapshots for institutional facts that change on a known cadence (rosters, millage, calendar), **machine-transcribed RAG corpora** for the meeting record, and **AI-extracted pages** for sources with no structured feed at all (below). Personal lookups that have no public API are handled with honest link-outs rather than invented answers.

Resilient feeds — cache, pre-warming & graceful degradation

Every external feed passes through one shared fetch layer: a per-source cache, a browser user-agent, retry-with-backoff, and — the load-bearing part — **stale-serve**. News-site CDNs routinely 429 (throttle) a datacenter IP; when that happens the last good response is served instead of a gap, and the failure is *never* surfaced in an answer (a "results may be incomplete" note on a sensitive query reads like censorship). Real-time data (live bus departures) and per-request lookups (a specific parcel) deliberately bypass the cache — there, a stale answer would be a *wrong* answer.

A live user request should never be the one paying for a network fetch. A background timer keeps every cache warm on its own schedule, decoupled entirely from request timing — the incident that taught this the hard way: a fresh deploy (cold cache) landing at the same moment as real chat traffic raced a dozen concurrent feed fetches against the platform's request timeout. Pre-warming means that race essentially never happens, and it has a second benefit: each source gets hit roughly once an hour instead of once per request, which matters for the smaller city sites that are visibly sensitive to request volume.

Some sources can't be fetched from the server at all. One county-wide feed returns HTTP 403 to the backend's hosting IP specifically — an ASN/datacenter-IP reputation block that no header or retry works around. Rather than run a headless browser or pay for a scraping API, a small script on an ordinary residential connection fetches it once daily and uploads the raw content to an admin-gated endpoint, stored in Firestore so it survives a redeploy between uploads. A second class of source has no feed at all — a plain HTML calendar page with the real event data embedded as a JSON blob inside a `<script>` tag. For those, the same daily upload is followed by an LLM extraction pass (comprehending the page regardless of exactly where the data sits, rather than a brittle selector scraper tied to today's markup) — run only from the background timer, never inline in a request, since the call itself takes several seconds. Either way, a missed day degrades to the last-good data, and a monitor emails the operator if a source goes stale for more than about a day and a half.

SECTION 10

Engagement, privacy & operations

Privacy by design

No accounts, no ad trackers. Chat history lives in the browser's `localStorage` as sessions; it travels per-request only. Device location is used once, resolved against county GIS, and never stored — coordinates never leave county data sources and never appear in logs.

Feedback loop

Thumbs-up is a private tally; thumbs-down and typed feedback are emailed to the operator with the Q&A context. Reported failures become the next eval cases — the correction loop is the product.

Telemetry & operator dashboard

Firestore logs each turn (question, answer summary, routing tier, check verdict, tools, timing, language) with no identity attached — but with the chat-session id, so an unlisted, key-gated dashboard groups turns into conversations for review. Records soft-delete after triage; eval-suite traffic is excluded at the source. Vercel Web Analytics counts visits cookielessly — no consent banner required.

Deployment

Push-to-deploy on both repos (Vercel + Heroku). A weekly GitHub Action scrapes new meetings, runs Gemini extraction, embeds the results, commits the data, and triggers a redeploy — the corpus stays current with zero manual effort.

Cost profile

Well under a penny per question (a generation call plus the verification call — both cheaper still after the flash-lite migration, which also cut model latency ~5×). Ingesting a meeting is a few cents, one time. The whole service runs for a couple of dollars a month at launch scale — and the architecture keeps per-question cost flat as the corpus grows, because retrieval precision, not corpus size, determines how much context each answer needs.