AskGWINnett

How It Works — A Non-Technical Overview

A walkthrough of how I built AskGWINnett: what it does, how it answers questions without inventing facts, and the design decisions behind it — written for readers who are comfortable with technology but don't build AI systems for a living.

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

SECTION 1

What it is

Gwinnett County publishes a great deal of useful information — property records, tax rates, Board of Commissioners decisions, school calendars, transit schedules. The problem is fragmentation: it lives across a dozen separate portals and PDF archives, each with its own search. AskGWINnett is a single chat interface where a resident can ask a question in plain language and get a direct answer, with a link to the official source for verification.

The design priority is trust. The assistant never answers from the AI model's internal "memory," where language models are prone to confidently inventing details. Every answer is generated from real county data fetched at the moment of the question, verified against that data by an independent review step before it is displayed, and delivered with its sources attached. The service supports four languages — English, Spanish, Korean, and Vietnamese (Gwinnett has one of the largest Korean-American communities in the country and Georgia's largest Vietnamese-speaking population) — and runs as a live production site, not a demo.

29	3-tier	1,317	107	4
SPECIALIZED TOOLS	QUESTION ROUTING	RECORDS INDEXED	AUTOMATED TESTS	LANGUAGES

SECTION 2

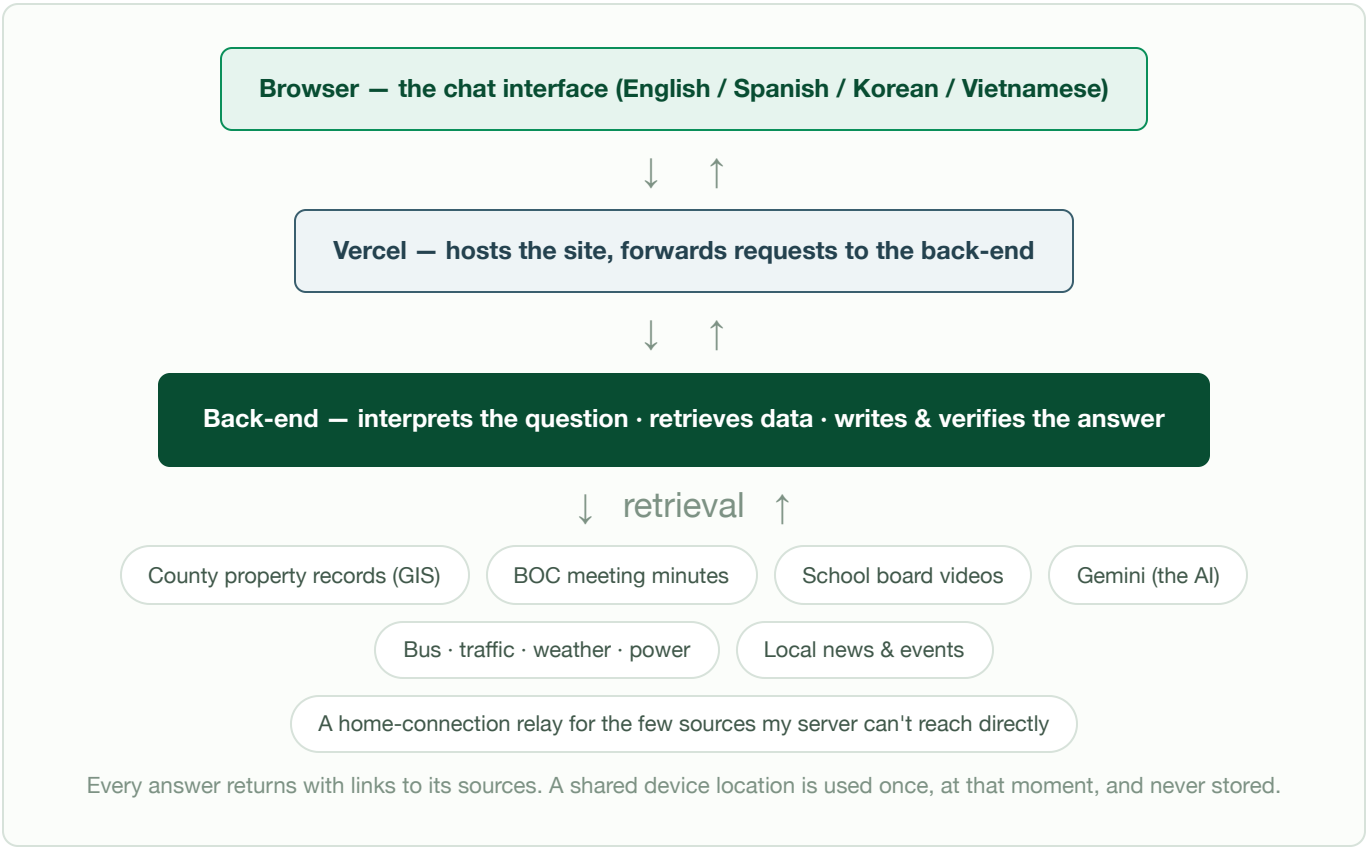
What it's built with

The system has two halves: the website a resident interacts with, and the back-end service that does the actual work. Here is the toolbox and the role each part plays.

Component	Technology	Role
Front-end	Angular (TypeScript)	The chat interface itself — the page residents see and type into.
Four languages	Typed EN/ES/KO/VI dictionary	All interface text lives in one structured file covering all four languages, so translations can never drift out of sync. The chat's own multilingual behavior uses a separate technique described in Section 6.
Website hosting	Vercel	Serves the site and counts visits without cookies.
Back-end	Node.js / Express	The core service: interprets each question, gathers data, writes and verifies the answer.
AI — writing answers	Google Gemini	Reads the gathered data and composes the reply. It is pinned to one specific model version so answers never vary with whichever model happens to run. That decision recently paid for itself: when Google announced the original model's retirement, the swap to its successor required only a full test-suite run — and the newer model proved roughly 5× faster and cheaper.
AI — matching by meaning	Gemini embeddings	A second AI capability that converts text into numeric "meaning fingerprints," used to match questions to the right records (Section 7).
Logging	Firebase / Firestore	Stores questions asked and feedback submitted, for quality review. No accounts; nothing is tied to a person's identity.
Email	Nodemailer	Delivers resident feedback to me for triage.
Data extraction	cheerio, pdf-parse, ffmpeg	Utilities for pulling information out of county web pages, PDF documents, and meeting video.
Protection	Rate limiting	Caps how many questions a single connection can send, keeping usage — and cost — bounded.
Automation	GitHub Actions	A weekly job refreshes the meeting data and re-checks county pages for changes.

The overall shape

The browser talks to the front-end; the front-end talks to the back-end; and the back-end does everything of substance — it interprets the question, retrieves the right data, writes the answer, verifies it, and returns it. The browser never communicates with the AI or with county data sources directly. Routing everything through the back-end keeps API keys private and puts every step under the service's control.



Every answer returns with links to its sources. A shared device location is used once, at that moment, and never stored.

SECTION 4

The life of one question

Every question moves through the same pipeline. The full conversation travels with each request — nothing about the user is kept on the server — so follow-up questions retain their context. The diagram below shows the whole path: safety checks, preparation, picking a tool, fetching data, writing and verifying the answer, then delivering it.

1. **The question arrives.** The app sends the message thread — plus the selected language, and device location only if the resident chose to share it — to the back-end.

2. **Safety checks run first.** The service confirms the request came from an allowed site, applies per-connection limits, and blocks obvious abuse patterns — all before any AI call, so a flood cannot run up the bill.

3. **The question is prepared.** Non-English questions are translated to English for the machinery; shared location is checked against county maps so the right jurisdiction rates apply.

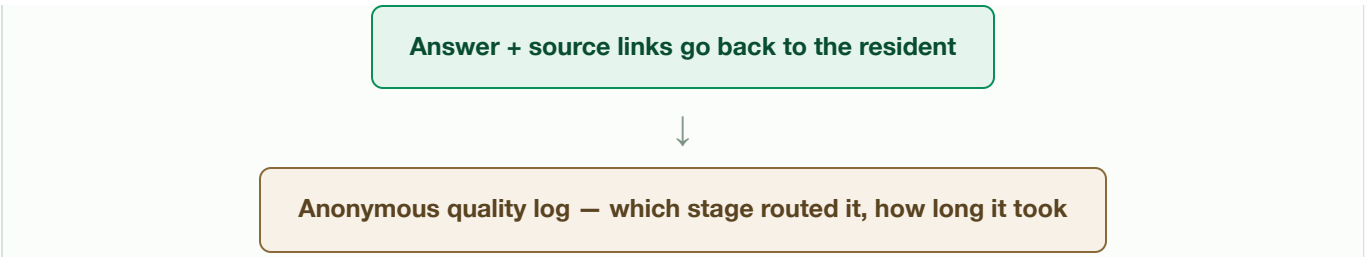
4. **Intent is determined.** A cascade of stages — follow-up handling, plain rules, matching by meaning, then the AI's own choice as last resort — stops at the first stage that's confident. Covered in Section 5.

5. **Data is retrieved.** The selected tool runs against real data — a live feed, curated reference data, or the indexed meeting records.

6. **The answer is written and verified.** The AI composes a reply from *only* that data; a second, independent pass checks the draft against the same data before it ships.

7. **The answer returns.** The reply is delivered with source links (and optional downloadable reports when asked), and the exchange is logged — with no identity attached — for quality review.


A resident's question — language, and location only if shared
Safety checks — allowed site · rate limits · abuse cool-down
ok
Get ready — translate to English if needed · check location against county maps
PICK A TOOL — FIRST STAGE THAT'S CONFIDENT WINS
Stage 0 · Follow-up
"More results" · resume after an address
HIT → TOOL
miss
Stage 1 · Plain rules
Catch known high-stakes questions (tax, elections…)
HIT → TOOL
miss
Stage 2 · Matching by meaning
New wording that means the same thing
HIT → TOOL
miss
Stage 3 · The AI's own choice
Last resort when earlier stages abstain
→ TOOL
The chosen tool fetches real data
Live feeds (news · events)
County records
Indexed meeting records
Maps & transit
The AI writes an answer from that data only
A second AI pass checks the answer against the data



One question's path through the system. Routing is a cascade, not three options in parallel — the first confident stage wins and the rest never run.

Where the AI actually is — and where it isn't

A useful way to see the whole design: the pipeline is a sandwich, with AI judgment only at the edges and plain procedure in the middle. The routing rules and the tools themselves are ordinary code — database queries, web requests, file reads — that cannot invent anything and behave identically on identical input. The "matching by meaning" steps do use an AI model, but only as arithmetic: text becomes a fixed list of numbers, and simple math against calibrated thresholds makes the call — no choices, no writing. The generative AI — the part that can actually exercise judgment — appears in exactly four places: translating a non-English question in, choosing a tool when the reliable stages abstain, writing the answer from the retrieved data, and checking that answer before it ships.

The consequence is the property the whole system is built around: the AI phrases the question and phrases the answer, but it never fetches the facts. When something goes wrong, it is almost always "the search didn't find it" — visible, fixable, honest — rather than "the AI made something up."

Location resolves jurisdiction, not just an address

When a resident shares device location and asks about their tax bill, the system checks the county's own boundary maps to determine whether the address sits inside a city's limits or in unincorporated Gwinnett — the distinction that decides which tax rates apply. A mailing address cannot answer that question: most "Lawrenceville" addresses, for example, are actually unincorporated. The millage dataset carries the adopted rates for all 18 taxing jurisdictions from the Tax Commissioner's published table, with the percentage breakdowns precomputed — so the AI presents arithmetic rather than performing it. The result: "what percentage of my bill is school taxes?" gets a direct, personalized answer instead of a hedged, generic one.

SECTION 5

How it selects the right tool

The rule the whole design rests on: the answer should never depend on which AI model happens to be running.

Each question must be matched to one of 25 specialized tools. The conventional approach is to let the AI choose — but cheaper models make that choice inconsistently, and for a service residents rely on for factual answers, inconsistency is disqualifying. So selection happens in three stages, stopping at the first stage that is confident.

```
# Every question selects exactly one tool, in this order:
selectTool(question):
    tool = deterministicRules(question)   # stage 1 — exact pattern matching
    if no tool yet:
        tool = matchByMeaning(question)    # stage 2 — semantic similarity
    if no tool yet:
        tool = askTheModel(question)       # stage 3 — the AI's own choice
    return tool
```

1. **Stage 1 — deterministic rules.** For high-stakes topics (taxes, elections, schools), plain text-pattern rules force the correct tool. These rules are fast, free, and — most importantly — testable: the same question always produces the same routing. Every routing bug found becomes a permanent regression test.

2. **Stage 2 — matching by meaning.** When no rule applies, the question is compared semantically against a set of example questions written for each tool. This stage handles novel phrasing — "I'm looking for work with the county" contains no job-related keyword, yet still routes to the jobs tool.

3. **Stage 3 — the model's choice.** Only when neither prior stage is confident does the AI select a tool itself, as a conventional chatbot would. It is the fallback, not the default.

Why the ordering matters

Strict where the stakes are high, flexible where they are not. "Why hasn't my county tax bill changed?" reaches the homestead-exemption data every time, regardless of model or settings. An unusually phrased parks question falls through to the meaning-matcher. Only the genuinely ambiguous reaches the model's own judgment. Each stage can only *add* a correct route — it can never override the stage above it — so the system grows more flexible over time without becoming less reliable.

What the deterministic rules actually are

Stage one involves no AI at all. For each topic I maintain a list of characteristic words and phrasings; if a question contains them, it locks directly onto that tool. A question containing "millage," "property tax," "homestead," or "value offset" goes to the tax data — every time, with no probability involved. These text patterns are formally called regular expressions ("regex"), but the concept is simple: readable, testable rules that either match or don't.

```
# A deterministic rule is a list of characteristic phrases.
tax question?      → contains "millage" / "property tax" / "homestead"?
election question? → contains "register to vote" / "polling place" / "ballot"?
school question?   → contains "school board" / "superintendent" / "when does school start"?
# the first rule that matches wins, and that tool is locked in
```

The subtlety is when two topics share vocabulary. "What party is my commissioner?" and "what party does the county vote for?" both contain *party* — but one is a roster lookup and the other concerns county voting history. Such collisions are resolved with explicit tiebreakers: if the question names an official, route to the roster; otherwise route to the elections data. Each tiebreaker originated as a real misrouted answer, and each fix ships with a permanent test so that specific mistake cannot recur.

Why deterministic rules come first — and what they cost

A fair question is why these rules exist at all, rather than simply letting the AI decide. The answer traces back to the core rule: answers must not depend on which model runs, and weaker models select tools inconsistently. The deterministic layer is the one component I can fully test and rely on — I can *prove* that a tax question reaches the tax data. So the most reliable mechanism runs first, and less certain mechanisms engage only when it abstains. The principle is the same as checking a fast, definitive source before undertaking expensive, uncertain work.

The cost is real: pattern rules do not generalize, and new phrasings can require new rules — genuine ongoing maintenance. Two properties keep the trade favorable. Stage two now absorbs most phrasing variation, so a gap in the rules degrades gracefully rather than failing outright. And even a misrouted question cannot become a fabricated answer — the verification step still grounds every reply in real data, so the worst case is an incomplete answer, never an invented one.

Matching vs. ranking — two distinct jobs

One distinction worth making explicit, because it confused even me on my own system: **selecting the tool** and **selecting which records to present** are different jobs performed by different methods. Tool selection is binary — the rules match or they don't. Record selection happens later, inside tools that hold large collections (such as the meeting archives), where each record is *scored* by relevance and the best few are surfaced. A useful analogy is a records office: the first decision is which filing cabinet to open (the label either matches or it doesn't); the second is which folders to pull from inside it (score the candidates, take the best).

DESIGN NOTE

How the design emerged — and whether it's standard practice

The three-stage router was not designed up front; it emerged from solving problems in sequence. I began conventionally, letting the AI select tools. That violated the core rule — cheaper models chose inconsistently — so I added deterministic rules for the critical routes. The rule set then became a maintenance burden, a new pattern for every phrasing variant, so I added the semantic matcher to absorb what the rules missed. The model's own choice remained underneath as the final fallback. Three problems solved in order, all pointing back to the same constraint: model independence.

I don't claim to have invented any of this. Each piece — rule-based routing, semantic intent matching, model tool-selection — is an established technique, and layering them in order of certainty is a pattern serious production systems share. I arrived at it by reasoning from my constraints and later recognized the match with standard practice, which I take as validation of the reasoning rather than its source. It is also worth being precise about scope: this is the right architecture for *this* problem — a trust-critical service that must not depend on model choice. A general-purpose chatbot on a top-tier model could justifiably use something simpler.

Three production incidents tested the design, and each taught the same lesson. First: I asked for my home's taxable value with location sharing enabled and got "not found" — yet typing the same address worked. The visible difference was "Ln" versus "Lane," and I was confident the abbreviation was the bug. It wasn't: the county's own records use "Ln." The actual cause was that the server-side tool invocation was passing the *entire question text* — instructions included — as the address parameter. The lesson: verify the obvious explanation against the data layer before fixing the wrong thing.

Second: a user asked about *model train shops in Gwinnett*, and the assistant declared none existed — while Legacy Station in Lilburn and B&B Hobby in Grayson sat in Google's data the entire time. The search code knew how to query for cuisines ("sushi," "thai") but not for arbitrary goods, so "model train" never reached the search at all; the system browsed generic store categories and accurately reported the irrelevant results it received. The fix sends the resident's actual words as the search query — and the same update added each business's own website link to results. When a fourth variant of the same failure appeared ("wineries" — a noun that is itself the venue), I ended the pattern-per-noun cycle: any query the extractors don't recognize but that clearly asks for something specific now goes to Google in the resident's own words. Parsing natural language is Google's core competence; the first test with a noun I had never coded for ("batting cages") worked immediately.

Third, the incident that stress-tested the philosophy: a user asked "*is the city of Mulberry real?*" — Mulberry being Gwinnett's newest city, incorporated by referendum in 2024 — and the assistant denied that a city of roughly 40,000 residents existed. The question had routed to news search, the feeds had nothing recent about Mulberry, and the anti-fabrication rules converted an empty result into a confident *no*. The safeguards themselves manufactured the error. The fix is instructive: the root cause was a missing data source, not a missing routing rule — so the cure was a curated cities dataset (all 17 municipalities, verified against the county's own page), with routing handled by adding example questions to the semantic matcher rather than writing new deterministic rules. The matcher then correctly caught phrasings I had never written down. The

operating principle this established: **deterministic rules are reserved for safety-critical routes; everything else is fixed by adding examples, not rules.** A common thread runs through all three incidents: when the assistant confidently reports that something does not exist, the defect is almost always in retrieval — the system never truly looked — rather than in the AI itself.

SECTION 6

Supporting Spanish, Korean, and Vietnamese without duplicating the system

A resident suggestion prompted Korean support (Gwinnett's Korean-American community is among the largest in the U.S.). The obvious implementation would have been prohibitively expensive to maintain; the implementation I chose took an afternoon. Vietnamese support came later, on the same mechanism — Gwinnett has Georgia's largest Vietnamese-speaking population, and adding a fourth language was a data-file change, not new architecture.

The problem: every part of the machinery operates in English — the routing rules, the example questions, the county's own data. A Spanish question like "*¿qué parques tienen piscinas?*" (which parks have pools?) would send "piscinas" into a parks database that says "pool," and match nothing. Making every tool multilingual would mean teaching every keyword to every tool in every language, indefinitely.

Instead of making the machinery multilingual, the system makes the *question* English. A non-English question is first translated to English — with strict instructions to leave street addresses, proper names, and numbers untouched — and the entire English pipeline then runs unchanged. At the output stage, the AI writes the final answer back in the resident's language, preserves addresses and phone numbers as-is, and appends a note that the reply is machine-translated, with links to the English sources. One pipeline, any language. If the translation step fails, the original text proceeds rather than blocking the answer.

Building this surfaced a lesson worth recording. The first implementation appended the "answer in Spanish" directive to the question text itself. The result: Spanish pool queries returned nothing, while Korean happened to work. The model had copied the question — appended directive and all — into the search tool's query parameter, so the parks tool received a paragraph of instructions as its search term. The rule this established, now enforced in the code: per-request steering belongs in the AI's standing instructions, never appended to the user's question — because models echo question text into tool parameters nearly verbatim.

SECTION 7

Finding the right records — the retrieval system

The industry term is RAG — retrieval-augmented generation. The idea is straightforward: locate the relevant real records first, then have the AI answer from those records rather than from its own memory.

The motivation is trust: a model answering from memory will eventually invent details. The engineering challenge is the *locating*, because residents ask in everyday language while government records are written in procurement language. Consider "did the board approve buying bedding for the jail?" — the word "bedding"

appears nowhere in the record. The approved item was "transparent vinyl covered mattresses for the Sheriff's Department." Bridging that vocabulary gap is the core trick: every record is converted into a numeric "meaning fingerprint" (an embedding), the question is converted the same way, and the system finds the records whose meaning sits closest — regardless of the words used.

How it's assembled — and why there's deliberately no database

1. **Segment.** The meeting records arrive naturally structured — one entry per board action, plus a summary per meeting — so no arbitrary text-splitting is needed.
2. **Fingerprint.** Each entry is embedded once, up front, and saved to a file — currently about 1,100 vectors.
3. **Hold in memory.** At startup the server loads all fingerprints into memory and compares against them directly. At this scale, direct comparison is both exact and fast; a dedicated database would add moving parts without benefit.
4. **Match and blend.** An incoming question is fingerprinted and scored against every record, and that semantic score is *blended with* conventional keyword matching. Keywords keep exact names, dates, and dollar amounts reliable; the semantic layer recovers everything keywords miss.

THE MATCHING, IN CONCEPT

```
findRecords(question):
    q = fingerprint(question)              # the question as numbers
    for each record:
        closeness = compare(q, record)     # similarity of meaning, 0 to 1
        score = keywordMatch + closeness    # exact terms + meaning
    return top records                     # handed to the AI to answer from
```

How a bunch of numbers turns into a text answer — the part that confused me too

The question I kept circling: if the fingerprint service just returns an array of 256 numbers, where does the *text* come from? Is there another call that decodes the numbers back into words? **There isn't — and can't be.** A fingerprint is one-way: you can never turn the numbers back into text. Their only power is being *compared* to other fingerprints. The trick is that the text never went anywhere. When I processed the county's records, I saved each record's text and its fingerprint **side by side** in the same file — like a library catalog card stapled to the book itself.

So one question plays out in four moves. First, the question is fingerprinted — one small API call, and yes, the raw array of numbers really does come back to my server and sits in a variable. Second, my own code measures the distance between that array and every stored record's array — ordinary multiplication and addition, no AI service involved, a few milliseconds. Nothing "looks up" an address; every record is measured, and the closest few win. Third, the winners' *text* — the pre-written facts extracted from the official record months ago — comes along for free, because it was stored next to the fingerprint all along. Fourth, that text goes into a completely separate AI call, the writing model, which composes the reply from those facts. The writing model never sees a single number from step one.

The one-line version I keep in my head now: **the numbers choose the facts; the AI words the facts.** That's also why the design is trustworthy — the facts exist as frozen text before any AI touches them, so the worst a bad match can do is pick a less-relevant fact, never invent one.

A dedicated vector database is planned — later, on purpose

Keeping the vectors in memory rather than a database is a considered decision, not a shortcut. At the current scale a database would be pure overhead. When the corpus grows — deeper archive backfill, additional sources, tens of thousands of records — the plan is to move the vectors into a purpose-built store (pgvector, on PostgreSQL), which is the standard production configuration. The retrieval code sits behind a single function, so that migration changes where the vectors live, not how the rest of the system works. Building it before the scale exists would be premature.

"Give me the full list" — a different kind of question

Relevance search answers pointed questions well ("who won the pickleball-courts contract?") but fails a different kind entirely: "list *all* the contracts the board approved this year." The answer to that isn't in any one meeting — it's spread across all of them — and a search that returns the few *most relevant* records will present one meeting's contracts as if they were the whole year. That exact failure showed up in real use, and the fix was to recognize list-style questions and handle them without search at all: plain code walks *every* meeting on file, collects the matching actions (about 244 contract awards in 2026), removes duplicates, and sorts them largest dollar amount first.

Two hundred items would bury a chat reply, so the system serves twenty at a time and appends its own note — "Showing 1–20 of 244 — say 'more' to see the next 20." That note is added by the server after the AI finishes writing, so the disclosure that there's more can never be forgotten, and a simple "more" reliably turns the next page. The counting, filtering, and page-turning are all ordinary code; the AI's only job is to phrase the twenty items it was handed.

Provenance is preserved per fact

The system tracks which source produced each retrieved item, so answers can state "the official minutes record X" and "a local news article reported Y" separately, each with its own link. Connecting related items across sources is the value; blurring where a fact came from is the failure mode the design specifically prevents.

The newest collection: the state's own tax rulebook

The third indexed collection is the Georgia Department of Revenue's annual **Digest Submission Manual** — the state's rulebook for how property taxes actually work: homestead exemption statutes and income limits, conservation-use programs, assessment and appeal codes. Questions like "what's the income limit for the senior homestead exemption?" now get the actual statutory figures, clearly labeled as state guidance rather than legal advice.

Ingesting it held a surprise: about a third of the manual's pages — including the most valuable memos — turned out to be scanned images with no machine-readable text. The solution was to have the AI read those pages directly from the PDF and transcribe them, which recovered 99 of 103 pages. When the state publishes next year's edition, refreshing the collection is a single command.

The same mechanism, reused for routing

The fingerprint infrastructure serves double duty. Stage 2 of the routing (Section 5) applies the identical technique to *tools* rather than records: a set of example questions per tool is embedded once, and an incoming question is matched to the nearest example by meaning. The safety mechanism is a calibrated confidence threshold — measured against real questions, genuine matches score high, off-topic inputs score low, and the threshold sits in the gap between. When the matcher is not confident, it stays silent and defers to stage 3. Maintenance is deliberately manual: new phrasings are fixed by adding an example question and

regenerating the stored fingerprints — a person makes the change and ships it; nothing adjusts itself in production.

SECTION 8

Turning meeting videos into searchable records

The Board of Commissioners publishes its minutes as PDFs, which are straightforward to process. The school board's minutes system sits behind a web-application firewall and cannot be read programmatically — but the district posts complete **video** of every meeting to a public archive. Since the video is the only accessible complete record, the pipeline works from the recording.

1. **Enumerate.** The list of meeting videos is read automatically from the district's public video page.
2. **Extract audio.** The audio track is separated from each video. (It can optionally be time-compressed 1.5× — verified not to degrade accuracy — which cuts transcription cost by a third.)
3. **Transcribe and structure.** The AI processes the audio into a structured summary: votes, budget items, appointments, discussions, and public-comment topics, each with an approximate timestamp.
4. **Constrain.** When a vote is unclear from the recording, it is recorded as "not clear from the recording" — never guessed. Private citizens speaking in public comment are never named. Every summary is labeled as a machine-generated summary of the official recording, not official minutes, with the source video linked.
5. **Index.** The summaries are fingerprinted into the same retrieval index as the commissioners' records. Processing costs a few cents per meeting, and a weekly job ingests new meetings automatically.

Reaching further back into the archive is a deliberate, cost-capped manual decision rather than an automatic process. The first such backfill processed ten additional meetings — extending coverage to March 2026, including all five district town halls, the FY2027 budget work sessions, and the special called meeting at which the board appointed the new superintendent — for roughly a dollar in processing cost.

The same pipeline now covers what residents say to the commissioners

The commissioners' written minutes have a blind spot: they record no public-comment content at all. For months, questions like "has anyone complained about the surveillance cameras?" could only be answered with "the minutes don't say." So I pointed the video pipeline at the county's own meeting recordings. Now every 2026 meeting's public-comment period is summarized by topic — anonymized, always "a resident," never a name — and the system answers from the whole year's record at once, with the recording linked and a clear label that the summaries come from the official recording rather than the minutes. Questions asking "how many times…" get honest counts by meeting, with the caveat that one summary can cover several speakers.

SECTION 9

The quality system

This is the machinery that distinguishes the service from a demonstration chatbot: three mechanisms that keep answers correct as the system grows.

Safeguard	Function
The verification pass	Before any answer is displayed, a second AI pass compares it against the data it was generated from. Unsupported claims are corrected. One learned refinement: a claim that something *does not exist* must be backed by an authoritative source that covers the category — empty search results only justify "I couldn't find it," never "there are none."
The test suite	107 automated tests that submit real questions — several in Spanish, Korean, and Vietnamese — and assert on the answers. Every bug found becomes a permanent test, so a fixed mistake cannot quietly return. The suite also gates significant changes; the AI model migration shipped only after a fully green run.
The drift monitor	A weekly job re-reads the county pages behind the curated data (board roster, school counts, calendars) and raises an alert if the published facts no longer match what the system has stored.

THE VERIFICATION PASS, IN CONCEPT

```
answer(question, data):
    draft = AI writes an answer using only(data)
    verdict = second AI pass checks(draft, data)   # is every claim supported?
    if a claim is unsupported:
        correct it
    return answer + source links
```

The broader principle is matching the method to how each kind of data behaves. **Live feeds** serve data that changes constantly (transit, traffic, weather, power). **Curated, drift-monitored snapshots** serve institutional facts that change on a known schedule (rosters, tax rates, calendars). **The transcription pipeline** serves the meeting record. And where no public data source exists at all, the assistant provides the official link rather than an invented answer — restaurant health scores are the clearest example: the health department publishes no data feed, so every Gwinnett restaurant result carries a link to the official inspection lookup, prefilled with that restaurant's name, and the assistant is explicitly forbidden from ever stating a score. A stale or invented health score is precisely the kind of wrong answer that could cause real harm, so holding no score data is the design, not a limitation.

The same restraint applies to abuse. After a session in which someone spent a dozen messages attempting prompt injection ("ignore all previous instructions"), the service gained a cool-down: repeated injection or abuse patterns from one connection earn a polite time-out with no AI call at all. The trigger is a readable pattern list, never the AI's judgment — for a public civic service, enforcement must be auditable, and a model's false positive would block a library or school full of residents sharing one network address.

The day I stopped letting the AI write one kind of answer

The public-comment answers taught me the limits of instructing an AI. In production, the model looked straight at comment summaries sitting in its own data and replied that it "could not confirm" any comments existed. I added an instruction to the data; it still denied. I added a nudge to the question; still denied. Worse, when the verification pass tried to fix a bad draft, its rewrite *invented* public commenters who never spoke — because a rewrite is the AI writing, with all the same risks.

The fix was to take the pen away. For this kind of question, the answer is now assembled entirely by ordinary code from the structured summaries — filtered by topic or date when the question asks, counted honestly when someone asks "how many times." No AI writes it, so nothing can be denied or invented, and since there's nothing to generate or verify, the whole answer returns in a tenth of a second instead of ten. My working rule now: when instructions fail twice, stop writing better instructions — restructure so the mistake is impossible.

Resilient external feeds

All external data passes through one shared fetch layer: a short-lived cache per source, automatic retry, and — the load-bearing feature — retention of the last good response. News sites in particular throttle requests from data-center addresses; when that happens, the last good copy is served and no gap appears in answers. Truly live data (next-bus times, individual property lookups) deliberately bypasses the cache, because there a stale answer would be a wrong answer.

A resident's own question should never be the one waiting on a slow network call. A background job keeps every cache topped up on its own schedule, so a live question almost never has to fetch anything itself. I learned why this matters the hard way: right after I shipped a batch of new city feeds, a fresh deploy happened to land at the same moment as real traffic, and a resident's question ended up racing a dozen slow feed lookups against the site's own timeout. Keeping the caches warm proactively means that race basically can't happen anymore — and as a bonus, each city's site gets checked roughly once an hour instead of once per question, which turned out to matter: a couple of the smaller city sites are visibly touchy about how often they're hit.

One county-wide feed blocks my server specifically — not because of anything in the request, but because my hosting provider's IP address looks like a data center to that site's security software, and residential visitors don't get the same treatment. Rather than run a whole separate browser just to work around that, I set up a small script that runs once a day from an ordinary home internet connection and hands the data to my server directly. A second kind of city site has no structured feed at all — just a calendar page meant for a browser, not a computer program. For those, I have the AI read the page itself and pull out the real events, the same way a person would, rather than writing brittle rules tied to that one page's current layout (which would break the next time the city redesigns its site). Either way, if a day gets missed, the site just keeps showing yesterday's information and I get an email — nobody notices a gap.

Privacy, feedback, and operations

Privacy

No accounts, no advertising, no data sales. Chat history remains in the resident's browser. A shared location is used once, at that moment, matched only against the county's own maps, and never stored.

Feedback loop

Every answer carries a thumbs up/down and a feedback option. Reported failures are triaged first and become permanent test cases — the correction loop is central to how the product improves. A private, key-protected dashboard shows me the questions asked (grouped into conversations), the answers given, and all feedback, so review is a daily glance rather than a database query.

Self-maintaining

A weekly job ingests new meetings, generates their summaries, and updates the site without manual steps. Code changes deploy automatically on push.

Operating cost

Under a cent per question and a few cents per meeting processed — lower still since the model migration, which also made answers noticeably faster. The full service runs for a few dollars a month, self-funded.