Corporate White Paper

ZoningLens

Conversational Geospatial Intelligence for Zoning, Parcel and Hazard Analysis

An architecture and engineering overview of the platform for San Diego

14	3	2	5
Live SANDAG map layers	Agent tools (LLM tool-calling)	RAG knowledge corpora	Spatial MongoDB collections

Prepared for
Planning, real-estate and civic-technology stakeholders

Prepared by
SPL Systems

October 2026

Version 1.0 · Public

Executive Summary

Understanding what can be built on a parcel in San Diego currently means consulting several unconnected portals: a GIS viewer for zoning, the county assessor for ownership, separate hazard maps for fire, airport and coastal exposure, and the California legislative codes for the rules that apply. **ZoningLens** replaces that workflow with one browser-based experience. A user types a San Diego address and receives an interactive map of fourteen live planning layers, and can then ask questions in plain English such as *"Is 1234 Main St in a fire hazard zone?"* or *"What does the ADU handbook say about setbacks?"*.

The platform has two cooperating halves. The front end (React, TypeScript, OpenLayers) renders the map and chat interface. The **ZoningLens** back end (FastAPI) performs spatial joins against MongoDB, retrieves passages from two Milvus vector collections, and runs a locally hosted language model that decides which tool to call. All inference is performed on self-hosted infrastructure, so no property query is sent to a third-party AI provider.

1 The Challenge

Property due diligence is fragmented. Public portals such as the SANDAG ArcGIS services expose one layer at a time, provide no synthesis across layers, and offer no natural-language interface. Desktop GIS packages are powerful but heavy and require training. The result is slow, error-prone cross-referencing for the analysts, developers and researchers who need an answer quickly. Three gaps recur:

- **Discovery:** planning, utility and hazard layers live in separate tools.
- **Synthesis:** answering *one* question often requires joining address, parcel and several overlay layers by hand.
- **Interpretation:** zoning codes and handbooks are long documents that are hard to search.

2 Solution Overview

ZoningLens is organised around three capabilities that share a single user interface. **Explore** renders live layers around a searched address. **Analyse** chains address, point, parcel and overlay lookups into one server-side call. **Ask** places a tool-calling language model in front of both, so the user never needs to know which data source answers the question. Figure 1 shows the resulting four-tier architecture: a presentation tier, an API tier, a data and intelligence tier, and the third-party services the browser calls directly.

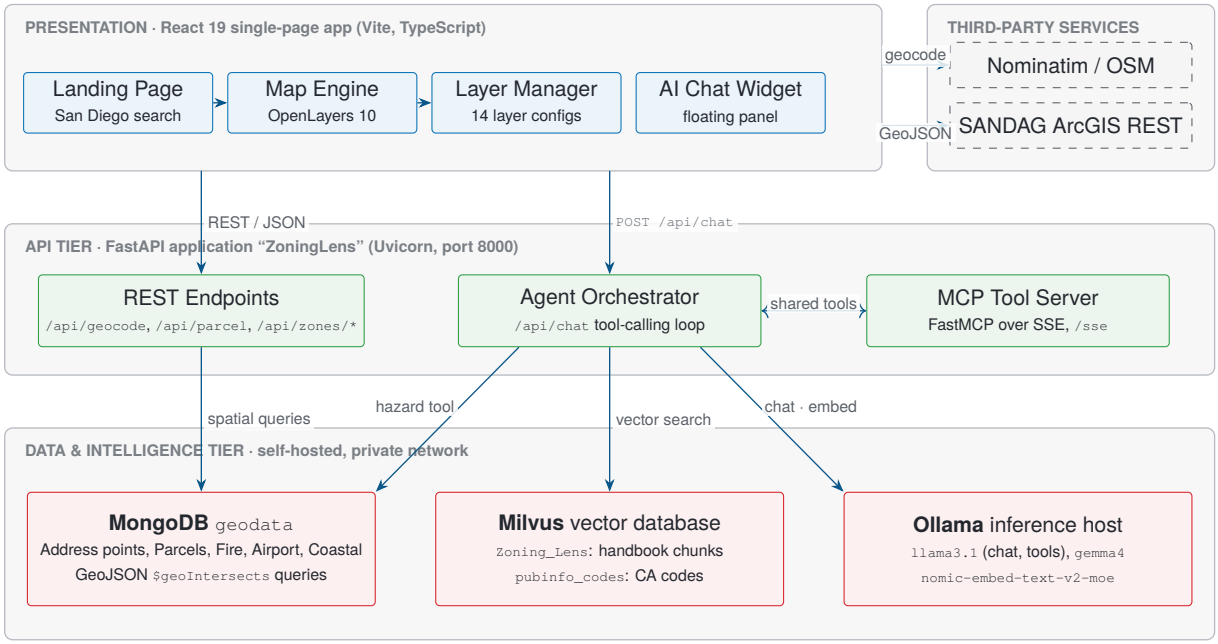


Figure 1: ZoningLens reference architecture. Map layers flow directly from SANDAG to the browser; chat, geocoding and hazard analysis flow through the FastAPI tier.

Key properties. (i) *Self-hosted intelligence*: model inference, embeddings and vector search run on private infrastructure. (ii) *Separation of concerns*: map rendering never depends on the AI stack, so the map stays usable if the language model is offline. (iii) *One tool surface, two protocols*: the same search functions serve the in-app agent and any external client through the Model Context Protocol (MCP).

3 Front-End Experience

The front end is a route-based single-page application built with React 19, Vite 6 and React Router 7. Mapping is handled by OpenLayers 10 with the `ol-ext` extensions; GSAP provides interface animation. The primary routes are the landing page (`/`) and the map workspace (`/open`), with an Esri-basemap variant (`/esri`) and a LegiScan bill-search experiment (`/legi`).

San Diego-locked address search. A global geocoder is the wrong tool for a regional dataset, so every Nominatim request carries a fixed San Diego viewbox with the bounded flag. Because a bounded box can still return edge-of-box results outside the city or county, the client re-validates the returned `city` and `county` fields before showing a suggestion. Requests are debounced to roughly one every 450 ms to respect the Nominatim usage policy. The same check guards both autocomplete and form submission.

Configurable layer system. Each layer is a declarative record (topic, search radius, OpenLayers style) from which a SANDAG FeatureServer query URL is generated. After a search, the layer manager issues the fourteen requests in parallel, each scoped to a radius around the address point, from 1 m for parcels up to 100 km for regional overlays.

Category	Layers	Search radius
Parcel & zoning	Parcels; Base Zoning	1 m; 50 m
Utilities	Sewer Mains; Water Mains	100 m; 50 m
Planning areas	Sustainable Development Areas; Housing Solution Areas; Promise Zones; Multi-Habitat Planning Area (MHPA); Community Plan Implementation Overlay	1 km (first two); 100 km (others)
Hazard & heritage	Fire Hazard Severity Zones; Airport Influence Area; Coastal Zones; Historic Districts; HRB Designated Resources	100 km

Table 1: The fourteen live layers and their query radii (source: front-end layer configuration).

4 Back-End Services: ZoningLens API

The back end is a single FastAPI application using Pydantic v2 models, an asynchronous MongoDB driver (Motor) and the asynchronous Ollama client, so no request blocks the event loop while waiting on the database or the model. Table 2 lists the public surface.

Endpoint	Method	Purpose
`/api/geocode`	POST	Resolve a street address to a point geometry from the address-point collection
`/api/parcel`	POST	Return the parcel (APN and boundary) containing a longitude/latitude
`/api/zones/fire`	POST	Test a polygon for intersection with the fire hazard layer
`/api/zones/airport`	POST	Same test against the airport safety zones
`/api/zones/coastal`	POST	Same test against the coastal zones
`/api/handbook/search`	POST	Top-k semantic search over the zoning/ADU handbook
`/api/codes/search`	POST	Top-k semantic search over California legislative codes
`/api/codes/rag`	POST	Retrieve codes, then generate a grounded answer with source list
`/api/chat`	POST	Tool-calling agent: one prompt in, one grounded answer out
`/sse`	SSE	Model Context Protocol server exposing the search and hazard tools

Table 2: ZoningLens API surface.

The conversational agent

The `/api/chat` endpoint gives `llama3.1` three callable tools, and the system prompt instructs the model to base answers entirely on retrieved data:

- `analyze_property_hazards`: fire, airport and coastal exposure for an address;
- `search_pdf_handbook_tool`: rules, definitions and procedures from the ADU/zoning handbook;

- `search_ca_codes_tool`: California legislative code lookup.

Figure 2 shows the control flow. Locally hosted models do not always populate the structured `tool_calls` field; they sometimes emit the call as raw JSON in the message text. A recovery parser detects this case, rebuilds a synthetic tool call and dispatches it, which prevents the user from ever seeing a raw JSON blob in place of an answer.

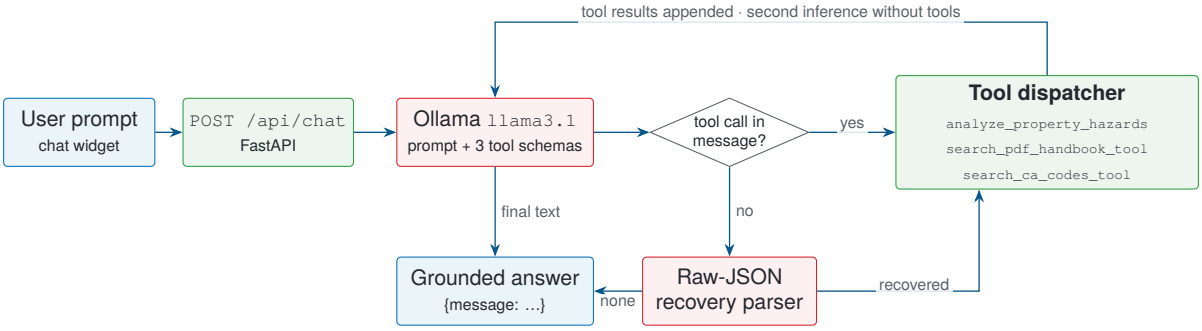


Figure 2: Agent control flow for `/api/chat`. Tool results are appended to the conversation and the model is called again to compose the final answer.

5 Spatial Intelligence

The flagship tool, `analyze_property_hazards`, answers *"what constraints apply to this property?"* in a single call. It tokenises the address into a house number and street name, finds the address point, locates the enclosing parcel with a `$geoIntersects` query, and then tests that parcel polygon against the fire, airport and coastal collections *concurrently* using `asyncio.gather`. Each layer returns an `intersects` flag and, when true, the matching feature's attributes (with geometry projected out to keep payloads small).

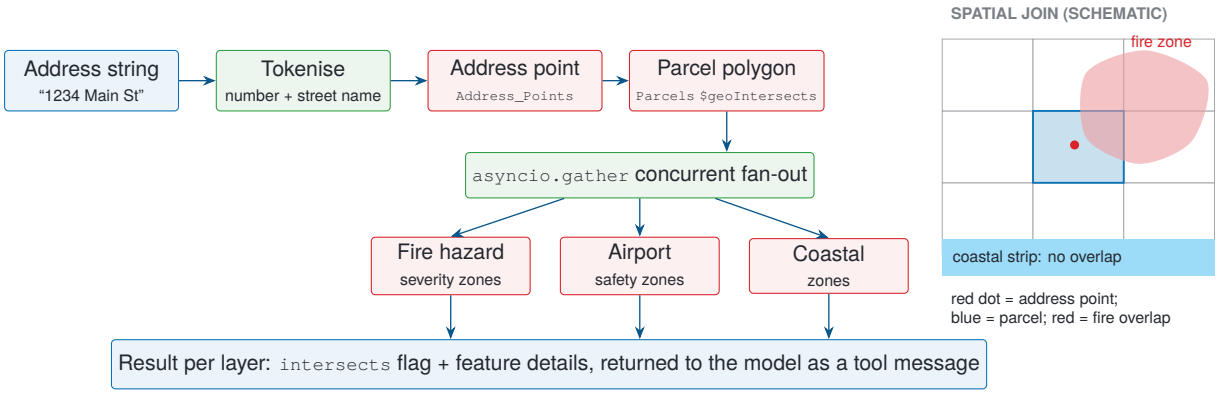


Figure 3: Hazard-analysis pipeline. Address resolution is sequential because each step needs the previous geometry; overlay checks are independent and run concurrently.

Design decision: a local spatial store. Map rendering reads live from SANDAG, but agent queries use a MongoDB copy of the shapefiles converted to GeoJSON. This lets the back end chain several spatial joins in one process without depending on ArcGIS REST semantics or rate limits on every chat turn. The trade-off is that the snapshot must be refreshed to stay current, and overlapping layers have two sources of truth. Unifying them is a roadmap item (Section 9).

6 Retrieval-Augmented Knowledge

Rules and definitions are not geometry, so they are served by retrieval-augmented generation (RAG) over two corpora in Milvus: the zoning and ADU handbook (`Zoning_Lens`, with text, page and source metadata) and a table of California legislative codes (`pubinfo_codes`, code abbreviation and title). Queries are embedded with `nomic-embed-text-v2-moe` and matched by vector similarity. Corpus ingestion runs in an external pipeline outside this repository; the service itself is read-only against both collections.

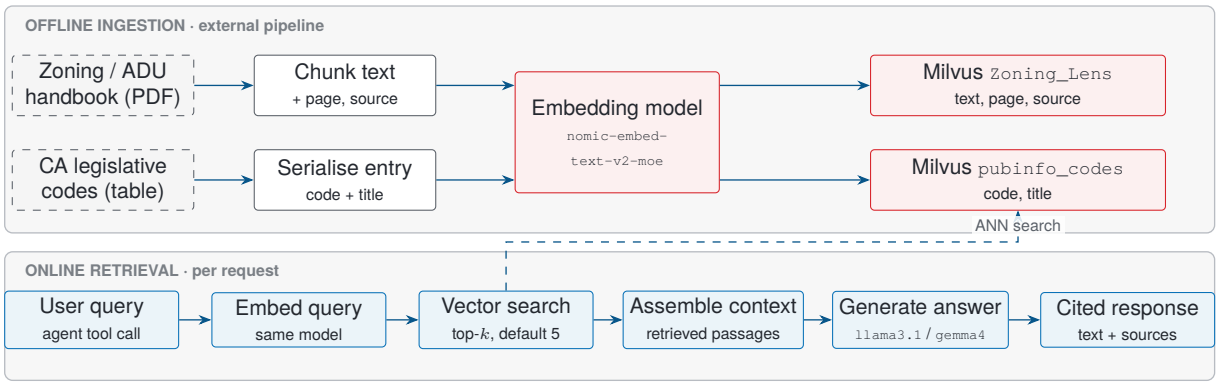


Figure 4: RAG data flow. Ingestion and retrieval share one embedding model so query and document vectors live in the same space.

The dedicated `/api/codes/rag` endpoint pins the generator to a low temperature (0.1), instructs it to admit when the retrieved context does not contain the answer, and returns the ranked sources alongside the text so a reader can audit every claim.

7 Deployment and Operations

Both tiers are containerised. The back end image is based on `python:3.12-slim` and serves Uvicorn on port 8000; the front end image runs the Vite server on port 5173. MongoDB, Milvus and Ollama run as separate self-hosted services on a private network, and the browser reaches only the public back-end endpoint and the two third-party map and geocoding services.

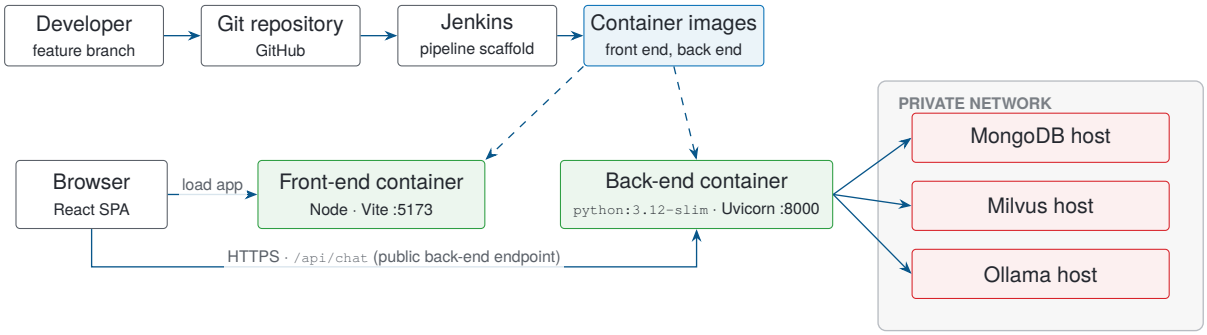


Figure 5: Delivery pipeline and runtime topology.

8 Security and Operational Readiness

ZoningLens is in active development and has not yet been hardened for open public release. We state the current posture plainly so that adopters can plan around it.

Area	Current state	Planned remediation
Authentication	None on API routes; CORS permits all origins	Token-based authentication; restrict origins to the front-end domain
Secrets	Database credentials and one third-party API key are held in source	Rotate, then move to environment variables or a secrets manager
Configuration	Hosts and model names are hard-coded constants	Externalise to environment configuration per stage
Input handling	Pydantic validates request shapes; address parsing is manual string splitting	Add a validated address grammar and request rate limiting
CI/CD	Jenkins pipeline with placeholder build and test stages	Real lint, type-check, test and image-build stages
Front-end image	Development server in the container	Static production build behind Nginx

Table 3: Security posture and remediation plan.

9 Roadmap

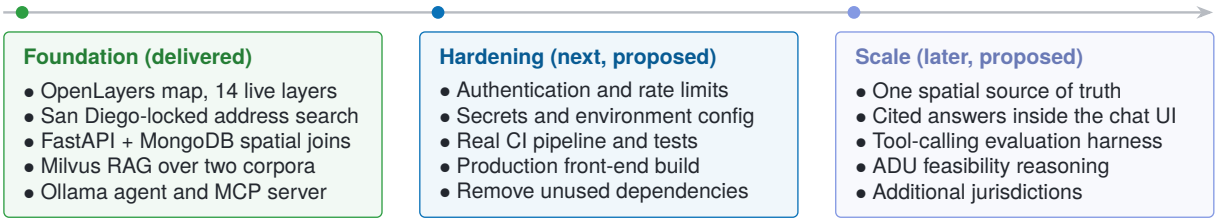


Figure 6: Delivery roadmap. Items after the foundation phase are proposals, not commitments.

10 Conclusion

ZoningLens shows that a small, self-hosted stack can turn scattered GIS layers and planning documents into one conversational workflow. Its central design choice is to let a language model *route* questions but never *answer from memory*: geometry questions go to MongoDB, rules questions go to Milvus, and the model only composes the retrieved facts. The hardening items above are the path from a capable prototype to a service that planners and developers can rely on every day.

Appendix: Technology Stack

Layer	Technology
Front end	React 19, TypeScript, Vite 6, React Router 7, OpenLayers 10 with `ol-ext`, GSAP, lucide-react
Back end	Python 3.12, FastAPI, Pydantic v2, Uvicorn, Motor (async MongoDB), PyMilvus, Ollama client, FastMCP
Spatial data	MongoDB database `geodata`: address points, parcels, fire hazard, airport safety and coastal zone collections
Vector search	Milvus collections `Zoning_Lens` and `pubinfo_codes`
Models (via Ollama)	`llama3.1` (chat and tool-calling), `gemma4` (code-RAG answers), `nomic-embed-text-v2-moe` (embeddings)
External services	Nominatim / OpenStreetMap (geocoding), SANDAG ArcGIS FeatureServer (map layers)
Delivery	Docker (one image per tier), Jenkins pipeline scaffold, ESLint and TypeScript checks

Table 4: Technology stack summary.