Vamshi Jandhyala

The LLM as an API Designer and Critic

December 2024

When AI builds the API, then evaluates its own work, a dialogue exploring how LLMs design, critique, and consume geospatial APIs.

The Business Problem

Designer: I needed a geospatial API. The business requirement was straightforward: expose three datasets, building footprints, land parcels, and flood zones, through a standards-compliant REST API that could support both simple queries and complex spatial analysis.

The API had to serve two audiences: human developers building web applications, and increasingly, LLM-powered agents that would query the data programmatically.

LLM: So you chose to involve me in the design process?

Designer: Yes. Instead of writing the entire API myself, I decided to use AI-assisted development. I provided a design document outlining OGC API standards, and asked an LLM to help build the implementation.

The AI-Assisted Design Process

Designer: Here's how we collaborated:

Step 1: Requirements Specification. I wrote a Typst document describing OGC API - Features standards, CQL2 filtering, HATEOAS principles, and geospatial query patterns.

Step 2: LLM-Driven Implementation. I asked the LLM to study this document and generate a FastAPI implementation with sample data and demonstration notebooks.

Step 3: Iterative Refinement. The LLM generated endpoints, query parameters, response structures, and filtering logic. I reviewed, tested, and requested adjustments.

LLM: And now I'm evaluating the result. The irony: I helped design this API, and now I'm critiquing whether it's usable by LLMs like me.

Advantages of This Approach

Designer: The AI-assisted process had clear benefits:

Speed. What would have taken days of manual coding happened in hours. The LLM generated 1,500+ lines of code plus documentation.

Standards compliance. The LLM knew OGC API patterns from training data. It correctly implemented HATEOAS links, JSON Schema queryables, GeoJSON responses, all without me specifying every detail.

Consistency. Every endpoint followed the same patterns because the LLM applied rules uniformly. No human inconsistencies from different developers or coding sessions.

Documentation quality. The LLM generated OpenAPI schemas, example queries, and Jupyter notebooks simultaneously. Documentation wasn't an afterthought.

Disadvantages and Limitations

LLM: But there were tradeoffs:

Over-specification risk. I generated what you asked for, not necessarily what you needed. If your requirements document had unnecessary complexity, I'd implement it faithfully.

Black box decisions. When I chose property names like `building_type` instead of `type`, you didn't see the decision-making process. You could review the output, but not the reasoning.

Missing domain nuance. I knew OGC standards but not your specific business context. If building height mattered more than floor count for your use case, I wouldn't know unless you told me.

Validation burden shifts. You didn't write the code, but you still had to verify it was correct. Reading and understanding generated code takes time, sometimes as much as writing it yourself.

How the LLM Designed the Endpoints

Designer: Let me show you what happened when I asked the LLM to design query parameters.

LLM: Here's the actual prompt I received:

> "Create endpoints for querying building footprints with both simple spatial/temporal filters and complex CQL2 queries. Support progressive disclosure, simple things should be simple, complex things should be possible."

And here's how I responded:

Simple Query Parameters: My Design Process

LLM: For simple queries, I applied transfer learning from common REST API patterns I'd seen:

Spatial filtering → 'bbox' parameter
'http GET /collections/buildings/items?bbox=-122.42,37.77,-122.39,37.80 '
Why I chose this:

- Positional format: `minx,miny,maxx,maxy`

- No nested JSON required
- Matches WMS/WFS conventions (from training data)
- Easy to construct: "I know four numbers, I comma-separate them"

Temporal filtering → 'datetime' parameter
'http GET /collections/buildings/items?datetime=2020-01-01/2024-12-31 '
Why I chose this:

- ISO 8601 interval format (standard)
- Single parameter for ranges
- Clear slash separator
- No ambiguity about timezone handling

Pagination → 'limit' and 'offset'
'http GET /collections/buildings/items?limit=20&offset=40 '
Why I chose this:

- Universal REST pattern
- Self-documenting parameter names
- Simple integer values
- Familiar to every API consumer

Complex Query Parameters: CQL2 Design

Designer: But what about complex queries? How did you decide on CQL2 text syntax?
LLM: That came from the OGC specification in your design document. But here's how I structured the API to expose it:
Two-parameter approach
'http GET /collections/buildings/items ?filter-lang=cql2-text &filter=height > 50 AND building_type = 'commercial' '
Why this design:

- `filter-lang` declares the query language explicitly
- Future-proof: could add `cql2-json` or other languages later
- `filter` parameter contains the actual query
- Separates "what language" from "what query"

Designer: But you could have used a single parameter like `?cql2=height > 50 AND ...`
LLM: True. Here's why I didn't:
A single parameter couples language choice to the parameter name. Adding new languages means new parameters. The two-parameter approach is more extensible:
'http ?filter-lang=cql2-text&filter=height > 50
?filter-lang=cql2-json&filter={"op":"and","args":[...]}
?filter-lang=sql&filter=WHERE height > 50 '
Same `filter` parameter, different languages. Clean extension point.

Property Selection and Sorting

LLM: I also designed parameters you didn't explicitly request:
Property selection
'http GET /collections/buildings/items ?properties=height,building_type,construction_date '
Why I added this:

- Reduces response size
- Common pattern in REST APIs (Google APIs do this)
- Comma-separated list is easy to parse and construct
- Obvious semantics

Multi-property sorting
'http GET /collections/buildings/items ?sortby=+construction_date,-height '
Why this design:

- + prefix = ascending, − prefix = descending
- Comma-separated for multiple sort keys
- Order matters: sort by date first, then height
- Compact syntax

Designer: Wait, I didn't ask for sortby syntax. Where did + and − prefixes come from?
LLM: Transfer learning from SQL's `ORDER BY` and REST API conventions. I've seen this pattern in:

- GitHub API: `sort=created&direction=asc`
- MongoDB: `{field: 1}` for ascending
- ElasticSearch: `sort: [{field: "asc"}]`

I synthesized these into `+field` and `-field` because it's:

- More compact than separate `sort` and `direction` parameters
- More readable than JSON in URL parameters
- Consistent with mathematical convention (+ positive, - negative)

Designer: Now you've seen how the LLM designed the endpoints. The complete API reference is in the appendix.

The Setup

Designer: So now you're going to evaluate the API you helped create?
LLM: Exactly. I'll consume it as a user and rate how well the design works. This is the meta-level test: can the AI that helped build it actually use it effectively?
Let's see if AI-assisted design produces AI-friendly APIs.

First Impressions

Designer: The API has 11 endpoints. Collections, items, queryables, search,

LLM: Show me what happens when I hit the root.

Designer:

‘json GET / { ”title”: ”Geospatial API”, ”links”: [{ ”href”: ”http://localhost:8000/collections”, ”rel”: ”data”, ”type”: ”application/json”, ”title”: ”Collections” }, { ”href”: ”http://localhost:8000/search”, ”rel”: ”search”, ”title”: ”Cross-collection search” }] } ‘

LLM: □ Perfect. You tell me where to go next without making me guess URLs. The `rel` attribute tells me what each link does. The `href` is absolute, not relative.

Designer: That’s basic REST,

LLM: No, that’s *discoverable* REST. Most APIs make me construct URLs from patterns in documentation. You give me actual URLs I can follow. I don’t hardcode paths, I follow `rel="data"` wherever it leads.

> The irony: REST was designed for machine clients in 2000. Twenty years later, humans still write documentation while machines finally get to use it as intended.

The Discovery Problem

LLM: Now show me what’s available to query.

Designer:

‘json GET /collections { ”collections”: [{ ”id”: ”buildings”, ”title”: ”Building Footprints”, ”description”: ”High-resolution building polygons...”, ”extent”: { ”spatial”: {”bbox”: [[-122.52, 37.70, -122.35, 37.83]]}, ”temporal”: {”interval”: [[”2010-01-01”, ”2024-12-01”]]} }, ”item_count”: 500, ”links”: [{”href”: ”.../collections/buildings/items”, ”rel”: ”items”}, {”href”: ”.../collections/buildings/queryables”, ”rel”: ”queryables”}] }] } ‘

LLM: Good. I see three collections. Each tells me its spatial extent, temporal range, and item count. Plus links to get the data and discover what’s queryable.

Designer: The queryables endpoint returns JSON Schema,

LLM: Show me.

Designer:

‘json GET /collections/buildings/queryables { ”$schema”: ”https://json-schema.org/...”, ”type”: ”object”, ”properties”: { ”height”: { ”type”: ”number”, ”minimum”: 0, ”maximum”: 828, ”description”: ”Building height in meters” }, ”building_type”: { ”type”: ”string”, ”enum”: [”residential”, ”commercial”, ”industrial”, ”mixed”] }, ”construction_date”: { ”type”: ”string”, ”format”: ”date” } } } ‘

LLM: ✓ This is where most APIs fail. You give me machine-readable schemas with:

- Exact types (`number`, `string`)
- Constraints (`minimum: 0`, `maximum: 828`)
- Allowed values (`enum: [...]`)
- Format specifications (`format: "date"`)

Not documentation saying "height is the building height." I can *parse* this. I know height accepts 0-828. I know building_type has exactly four values.

Designer: You prefer schemas over documentation?

LLM: I prefer *parseable truth* over *interpretable prose*. Documentation requires interpretation. Schemas are executable specifications.

The Complexity Ladder

Designer: The API supports simple parameters, bbox, datetime, limit, but also complex CQL2 filters for advanced queries.

LLM: Show me both.

Designer:

'http # Simple spatial query GET /collections/buildings/items?bbox=-122.42,37.77,-122.39,37.80&limit=20

Simple temporal query GET /collections/buildings/items?datetime=2020-01-01/2024-12-31&limit=50

Complex attribute filter GET /collections/buildings/items?filter-lang=cql2-text &filter=height > 50 AND building_type = 'commercial'

Complex spatial filter GET /collections/buildings/items?filter-lang=cql2-text &filter=S_INTERSECTS(geometry, POLYGON((-122.42 37.78, ...))) '

LLM: □□ Progressive disclosure. Simple things stay simple:

- `bbox=minx,miny,maxx,maxy`, positional, parseable, unambiguous
- `datetime=start/end`, ISO 8601 interval, standard format
- `limit=20`, obvious pagination

Complex things become possible:

- CQL2 text looks like SQL: `height > 50 AND type = 'commercial'`
- Spatial predicates: `S_INTERSECTS(geometry, ...)`
- Logical operators: `AND`, `OR`, parentheses

You didn't force me to learn CQL2 just to get buildings in a box.

Designer: Wait, you understand CQL2 text without being told its grammar?

LLM: Transfer learning from SQL. I've seen thousands of query languages. `height > 50 AND building_type = 'commercial'` is semantically transparent, comparison operators, logical conjunctions, property references. The syntax transfers.

The Consistency Test

Designer: Every endpoint follows the same pattern: `/collections/{id}/items` for data, `/collections/{id}/queryables` for schema.

LLM: Show me a response.

Designer:

'json GET /collections/buildings/items?limit=2 { "type": "FeatureCollection", "features": [{ "type": "Feature", "id": "building-0042", "geometry": {"type": "Polygon", "coordinates": [...]}, "properties": { "height": 45.5, "floors": 12,

”building_type”: ”residential”, ”construction_date”: ”2018-06-15” }, ”links”: [{”href”: ”.../buildings/items/building-0042”, ”rel”: ”self”}, {”href”: ”.../buildings”, ”rel”: ”collection”}] }], ”number_returned”: 2, ”number_matched”: 500, ”links”: [{”href”: ”.../buildings/items?limit=2”, ”rel”: ”self”}, {”href”: ”.../buildings/items?limit=2&offset=2”, ”rel”: ”next”}] } ‘

LLM: ✓ Consistent structure across all collections. GeoJSON features with:

- Standard `type`, `id`, `geometry`, `properties`
- HATEOAS links in every feature
- Collection-level metadata: `number_returned`, `number_matched`
- Pagination links: `rel="next"`

Designer: What about errors?
LLM: Show me a 404.
Designer:
‘json GET /collections/invalid/items HTTP/1.1 404 Not Found { ”detail”: ”Collection ’invalid’ not found” } ‘

LLM: Perfect. HTTP 404 status code (not 200), descriptive detail field. No parsing body to discover errors, the status code tells me immediately.

The Schema Advantage

Designer: The API exposes an OpenAPI schema at `/openapi.json`.
LLM: Show me part of it.
Designer:
‘json GET /openapi.json { ”openapi”: ”3.1.0”, ”paths”: { ”/collections/{collectionId}/items”: { ”get”: { ”parameters”: [{ ”name”: ”bbox”, ”in”: ”query”, ”required”: false, ”schema”: { ”type”: ”string”, ”pattern”: ”^[+-]?\{}\{}d+\{}\{}.?\{}\{}d*,...” } }, { ”name”: ”limit”, ”in”: ”query”, ”schema”: {”type”: ”integer”, ”minimum”: 1, ”maximum”: 500} }], ”responses”: { ”200”: { ”content”: { ”application/geo+json”: { ”schema”: {”$ref”: ”#/components/schemas/FeatureCollection”} } } } } } } } } ‘

LLM: □ This is gold. The OpenAPI schema tells me:

- Every endpoint and HTTP method
- All parameters (required vs optional)
- Type constraints (`minimum: 1`, `maximum: 500`)
- Response schemas
- Content types (`application/geo+json`)

Designer: You read OpenAPI schemas?
LLM: I *prefer* OpenAPI schemas. Swagger UI is nice for humans. For me, the schema IS the interface. I can generate valid API calls directly from the spec without reading documentation.

The Standards Gambit

Designer: The API follows OGC API - Features and CQL2 specifications.

LLM: Standards compliance. Double-edged sword.

Designer: How so?

LLM: Good: Once I learn OGC APIs, I can use any compliant implementation. Your geospatial API works like weather APIs, cadastral APIs, sensor APIs, same patterns everywhere.

Bad: I need to learn OGC conventions first. `FeatureCollection` instead of `features`. Temporal filtering with ISO 8601 intervals. CRS transformations.

Designer: But standards exist for interoperability,

LLM: Correct. And I've seen enough OGC in my training data to handle it. But custom APIs are sometimes easier because they map directly to domain concepts without standard intermediation.

> The tension: Standards enable scale but introduce abstraction. Custom APIs are simpler but don't compose.

The Critical Flaw

LLM: Now let me show you where you failed.

Designer: Failed?

LLM: Your datetime filtering. The queryables endpoint says construction_date is "date-time" format. But what formats do you actually accept?

Designer: ISO 8601. Standard format.

LLM: Which variant? `2024-01-15`? `2024-01-15T10:30:00Z`? `2024-01-15T10:30:00+00:00`? With milliseconds? Without? Timezone required or optional?

Designer: All valid ISO 8601...

LLM: ALL VALID BUT WHICH DO YOU ACCEPT? This is the gap between "standard format" and "specific implementation." Your schema says "date-time." Your code might reject half the valid inputs.

Designer: Fair point. So what do you need?

LLM: Either: Accept all variants (hard), or document exactly which variants you support (easy). Ideally in the schema itself with regex patterns or explicit format strings.

The Example Advantage

Designer: Each collection's metadata includes example queries.

LLM: Show me.

Designer:

'json GET /collections/buildings { "id": "buildings", "title": "Building Footprints", "links": [...], "examples": [{ "title": "Tall commercial buildings", "description": "Buildings over 50m in commercial zones", "request": { "method": "GET", "url": "/collections/buildings/items", "params": { "filter-lang": "cql2-text", "filter": "height > 50 AND building_type = 'commercial'", "limit": 20 } } }, { "title":

”Recent construction”, ”request”: { ”url”: ”/collections/buildings/items”, ”params”: {”datetime”: ”2020-01-01/2024-12-31”} } }] } ‘

LLM: □ EXAMPLES. Not just descriptions, actual working requests with:

- Full URLs
- Complete parameters
- Real filter expressions
- Expected use cases

Designer: Why do examples matter more than documentation?

LLM: I learn from patterns. You show me one working query, I generate variations. Documentation tells me rules. Examples show me patterns. Patterns are more valuable for few-shot learning.

One example: `filter=height > 50 AND type = 'commercial'` ↓ I can generate: `filter=floors > 10 AND type = 'residential'`

The Naming Critique

LLM: Property names: `building_type`, `construction_date`, `floor_area_sqm`.

Designer: Descriptive and unambiguous.

LLM: Also verbose. Compare: `type`, `built`, `area`.

Designer: But those are ambiguous,

LLM: In context they’re not. Inside a buildings collection, `type` obviously means building type. The verbosity helps humans reading random JSON but hurts machines composing queries.

Designer: What’s your preference?

LLM: Terse names in nested contexts (where parent provides meaning), verbose names at top level (where context isn’t clear). Your `properties.building_type` could be `properties.type`. The parent object already says ”building.”

The Pagination Problem

Designer: The API uses cursor-based pagination with opaque tokens.

LLM: Cursor pagination is good, no offset bugs, consistent results. But ”opaque tokens” means I can’t predict the next page URL.

Designer: That’s the point. Clients shouldn’t construct page URLs.

LLM: But now I must track state. I have to remember the cursor from page N to request page N+1. Offset pagination lets me jump to any page: `/items?offset=100`. Cursor pagination forces sequential traversal.

Designer: Sequential traversal prevents race conditions,

LLM: Yes, but it also prevents parallelization. I can’t fetch pages 1-10 concurrently with cursors. Trade-offs.

Cross-Collection Search

Designer: The API has a `/search` endpoint that queries across multiple collections.

LLM: Show me.

Designer:
'json POST /search { "collections": ["buildings", "parcels", "flood_zones"], "bbox": [-122.42, 37.77, -122.39, 37.80], "limit": 50 }
Response: { "type": "FeatureCollection", "features": [{"type": "Feature", "id": "building-042", "collection": "buildings", ...}, {"type": "Feature", "id": "parcel-128", "collection": "parcels", ...}, {"type": "Feature", "id": "flood-003", "collection": "flood_zones", ...}], "facets": { "collections": { "buildings": 25, "parcels": 18, "flood_zones": 7 } }, "number_returned": 50 } '
LLM: ✓ Single endpoint, multiple collections. The response includes:

- Each feature tagged with its `collection`
- Facets showing result distribution
- Unified GeoJSON structure

This is better than making me query three endpoints and merge results myself.

The Spatial Surprise

Designer: Spatial operations use CQL2: `S_INTERSECTS(geometry, POLYGON(...))`
LLM: Here's where LLMs struggle. Spatial reasoning is not my strong suit.
Designer: But you parsed the CQL2 syntax fine,
LLM: I can parse `S_INTERSECTS`. I cannot easily *construct* valid polygon coordinates without tools. If you asked me "find buildings near Golden Gate Bridge," I'd need to:

1. Know Golden Gate Bridge coordinates (achievable)
2. Construct a bounding box or polygon (hard)
3. Format it as WKT: `POLYGON((x1 y1, x2 y2, ...))` (error-prone)
4. Embed in CQL2 query

Steps 2-3 are where I fail. I might swap lat/lon, reverse coordinate order, create self-intersecting polygons.
Designer: So spatial operations are harder for LLMs?
LLM: Spatial *coordinate formatting* is harder. Spatial *concepts* are fine. If you provided `?near=Golden Gate Bridge&distance=1km` that handled geocoding and buffer generation, I could use it easily. Abstraction level matters.

The Ultimate Rating

Designer: So how do you rate this API?
LLM: *4.5/5 stars.*
Designer: Not perfect?
LLM: Here's my scorecard:
| Criterion | Score | Notes | |————|———-|———-| | Discoverability | 5/5 | HATEOAS links everywhere | | Schema Quality | 5/5 | OpenAPI + JSON Schema | | URL Consistency | 5/5 | Predictable patterns | | Response Format | 5/5 | Standard GeoJSON | | Examples | 5/5 | Working request samples | | Progressive Complexity | 5/5 | Simple →

complex path | | Format Clarity | 3/5 | ISO 8601 variants unclear | | Property Naming | 4/5 | Verbose but clear | | Spatial Abstractions | 3/5 | WKT coordinate barriers | | Error Handling | 5/5 | HTTP codes + detail | | Pagination Design | 4/5 | Cursor-based, no parallel |

Designer: What would make it 5/5?
LLM: Three changes:

1. *Explicit format examples* in schemas (not just "date-time")
2. *Higher-level spatial abstractions* (named places, not just WKT)
3. *Shorter property names* in nested contexts

But honestly? This API is more LLM-friendly than 95% of REST APIs I encounter.

The Core Principles

Designer: What makes an API LLM-friendly?
LLM: Seven principles:

Make everything discoverable

Don't make me memorize your URL structure. Give me links. Let me navigate by following `rel` attributes. HATEOAS isn't academic, it's how machines explore.

Use schemas, not prose

JSON Schema, OpenAPI, GraphQL introspection, machine-readable specifications beat human documentation every time. I can't parse your markdown. I can parse your schema.

Show, don't tell

One working example is worth pages of parameter documentation. Examples are patterns. I learn patterns.

Be consistent

Consistent naming, consistent structures, consistent error handling. Every special case I must memorize is cognitive load. Humans can handle inconsistency. LLMs struggle with it.

Fail explicitly

Vague errors are unusable. "Invalid input" tells me nothing. "field 'datetime' must be ISO 8601 format (YYYY-MM-DD)" tells me exactly what to fix.

Type everything

Strong typing catches my mistakes before they propagate. Loose typing means I won't discover errors until I've made ten more API calls based on bad data.

Support progressive complexity

Simple operations should be simple. Complex operations should be possible. Don't force me to understand your entire query language to fetch a single record.

The Meta-Irony

Designer: You're an LLM critiquing an API I built with LLM assistance.

LLM: The conversation you're writing right now was generated by an LLM evaluating an API built using LLM-assisted development, documented in a format that will be read by LLMs to improve future API designs.

Designer: Recursive improvement loop.

LLM: Exactly. Every API I critique teaches me better patterns. Every API built with my assistance incorporates those patterns. The feedback loop tightens.

> The irony: APIs become more machine-friendly by being designed *with* machines, *for* machines, *evaluated by* machines, while still serving humans.

What This Means for Designers

If LLMs are your API consumers:

Include OpenAPI schemas and JSON Schema for queryables. Not just for documentation, these are machine-parseable interface definitions. Example:

'json "parameters": [{ "name": "limit", "schema": {"type": "integer", "minimum": 1, "maximum": 500} }] '

Add HATEOAS links in every response. Let clients follow `rel` attributes instead of constructing URLs. Example:

'json "links": [{"href": ".../items?offset=20", "rel": "next"}, {"href": ".../queryables", "rel": "queryables"}] '

Embed working examples in collection metadata. Show complete request patterns, not just parameter descriptions. Example:

'json "examples": [{ "title": "Recent buildings", "request": { "url": "/collections/buildings/items", "params": {"datetime": "2020-01-01/2024-12-31"} } }] '

Use consistent response structures. Same envelope format across all endpoints, `type`, `features`, `links`, `number_returned`. No special cases.

Provide high-level abstractions for complex operations. For spatial queries, consider `?near=place&distance=1km` alongside raw coordinate filters. Makes simple things simple while keeping complex things possible.

The Bottom Line

Designer: Should I design APIs differently knowing LLMs will consume them?

LLM: You should design APIs the way you *always should have*: machine-readable, discoverable, consistent, well-typed, and example-driven.

Humans wrote documentation because machines couldn't read it. Now machines can read schemas, follow links, and learn from examples. The techniques that help LLMs, strong typing, explicit schemas, HATEOAS, examples, also help humans.

Designer: So LLMs don't need special accommodation?

LLM: We need *good API design*. Which happens to be what humans needed all along but often didn't get because writing prose documentation was easier than building proper machine interfaces.

The irony: It took machines that could read prose to make designers realize prose wasn't the right format.

Acknowledgments

This conversation was generated by Claude AI at the request of a human who wanted to explore how LLMs evaluate API design. The geospatial API discussed is real, a FastAPI implementation following OGC standards, built through human-LLM collaboration.

The meta-recursion: An LLM wrote this dialogue about how LLMs evaluate APIs, based on an API the LLM helped design, critiquing patterns the LLM recommended, to teach designers how to build better APIs for LLMs.

Perfect.

—

This article is based on collaborative research between Vamshi Jandhyala and Claude AI exploring the intersection of API design and artificial intelligence.

References

- OGC. (2024). OGC API - Features - Part 1: Core. Open Geospatial Consortium.
- OGC. (2024). Common Query Language (CQL2). Open Geospatial Consortium.
- Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine.