intelligrit

CASE STUDY

# 12 Million Words, 2 Days, 1 Engineer

*How one AI-augmented engineer extracted structured data from one of the longest works of fiction ever written — and what it means for anyone working with messy, real-world data at scale.*

Intelligrit LLC | February 2026

intelligrit.com | contact@intelligrit.com | 202-525-6382

## The Short Version

Over a weekend, a single Intelligrit engineer built a system that read 807 chapters of a 12-million-word web serial, extracted every geographical reference using AI, and rendered them as an interactive, spoiler-free map. The entire codebase is 3,200 lines of Go and 1,000 lines of JavaScript. Two days. No infrastructure beyond a laptop.

We didn't build this for a client. We built it because we wanted to see if we could. But the techniques, the speed, and the results speak directly to problems that every organization faces: mountains of unstructured data, manual extraction processes that take months, and timelines that stretch into years — whether you're a federal agency modernizing legacy systems, a startup making sense of customer feedback, or a research team processing literature at scale.

## The Problem We Solved

*The Wandering Inn* is a web serial — one of the longest works of fiction in the English language at over 12 million words across 10 volumes. It builds a detailed fantasy world with hundreds of named locations: continents, nations, cities, dungeons, roads, landmarks. But there's no official map, and fan-created maps are incomplete and full of spoilers.

We wanted to automatically extract every geographical reference from every chapter and build an interactive map that reveals the world progressively — so a reader on Volume 3 only sees what's been mentioned through Volume 3.

This is a toy problem. But the shape of the problem is universal — it shows up in federal modernization, enterprise data migration, research, and product development:

- **Massive volumes of unstructured text** that need structured data extracted
- **No existing database** — the knowledge lives only in documents
- **Progressive disclosure requirements** — different users need different levels of access
- **Quality at scale** — extraction must be accurate across hundreds of documents, not just a handful

## What We Built

A five-stage automated pipeline:

1. **Scrape** — Download 807 chapters from the web (rate-limited, resumable)
2. **Extract** — Send each chapter to Claude AI for structured location extraction
3. **Aggregate** — Deduplicate, normalize, and merge results across all chapters
4. **Coordinate** — Assign map positions using containment hierarchies and seed data
5. **Serve** — Interactive Leaflet.js map with a chapter-progress slider

The entire system compiles to a single binary. No Docker, no database server, no cloud infrastructure. `go build` and run.

## Technology Choices

### Why Go

We chose Go deliberately, and it paid off in several ways:

- **Single binary deployment.** `go build` produces one executable with everything embedded — the web server, static assets, templates. No runtime, no node_modules, no virtualenv. You copy the binary and run it.
- **go:embed for static files.** Go's `embed` directive bundles HTML, CSS, and JavaScript directly into the binary at compile time. The map server is entirely self-contained.
- **Concurrency for free.** The scraper needs rate limiting and the extractor makes hundreds of API calls. Go's goroutines and `golang.org/x/time/rate` made this trivial to implement correctly.
- **Standard library HTTP server.** The `net/http` package is production-grade out of the box. Our map server is 120 lines of code including the API endpoints, static file serving, and graceful handling of the embedded filesystem.
- **Fast iteration with Claude Code.** Go's simplicity — explicit error handling, no inheritance hierarchies, minimal abstraction — makes it an excellent language for AI-assisted development. The code reads linearly, which means the AI can reason about it effectively and produce correct modifications.
- **Compilation catches errors early.** With a pipeline this complex, Go's type system and compiler caught classes of bugs at build time that would have been runtime surprises in Python or JavaScript.

The dependency footprint is minimal: Cobra for CLI structure, goquery for HTML parsing, go-duckdb for storage, and `golang.org/x/time` for rate limiting. Everything else is standard library.

### Why Plain HTML, CSS, and JavaScript

The frontend is 1,000 lines of vanilla JavaScript, a single CSS file, and one HTML file. No React. No Vue. No HTMX. No build step. No bundler. No transpiler. No node_modules.

This was a deliberate choice, not a limitation. The application has a single page with a map, a sidebar, and a control bar. It loads data from four API endpoints and renders it with Leaflet.js. The interaction model is straightforward: change a dropdown, fetch data, redraw. A framework would add a build pipeline, a dependency tree, and an abstraction layer — all for a problem that DOM manipulation handles directly.

The result:

- **Zero build step.** Edit a `.js` file, rebuild the Go binary (which embeds the static files), and reload. There is no `npm install`, no webpack config, no babel preset to debug.
- **No framework version churn.** React 18 to 19, Vue 2 to 3, Angular's yearly breaking changes — none of this applies. The browser API is the framework, and it's stable.
- **Readable by anyone.** A developer who knows JavaScript can read the entire frontend in one sitting. There are no component lifecycles, no state management libraries, no custom hooks to trace through.
- **AI writes it cleanly.** AI coding assistants produce excellent vanilla JS because the browser API is abundantly represented in training data. Framework-specific patterns — especially newer versions — are where AI hallucinations cluster. By using the platform directly, we get more reliable AI output with less review overhead.

When the problem fits — and many problems do — plain HTML/CSS/JS is faster to write, easier to maintain, cheaper to deploy, and simpler to audit than any framework-based alternative. The instinct to reach for React on every project is a habit, not a requirement.

### Open Source Licensing

Every dependency in this project uses a permissive open source license. We care about this because licensing is an engineering decision with legal and operational consequences:

| DEPENDENCY | LICENSE | TYPE |
|---|---|---|
| Go standard library | BSD 3-Clause | Permissive |
| Cobra (CLI framework) | Apache 2.0 | Permissive |
| goquery (HTML parsing) | BSD 3-Clause | Permissive |
| DuckDB Go driver | MIT | Permissive |
| golang.org/x/time | BSD 3-Clause | Permissive |
| Leaflet.js (map rendering) | BSD 2-Clause | Permissive |
| TWI Map itself | MIT | Permissive |

No GPL. No AGPL. No SSPL. No license ambiguity. Every component can be used, modified, and distributed without viral licensing concerns. This matters for government procurement (where license review is a gate), for enterprise adoption (where legal review delays deployment), and for open source distribution (where incompatible licenses poison the well).

We audit dependencies deliberately. AI coding assistants will cheerfully import a library without checking its license — the same way they'll use a deprecated package (see Finding 2). Dependency selection is an engineering judgment call: evaluate the license, verify the package is actively maintained, confirm the module path hasn't changed, and check for known vulnerabilities. Automate what you can (`go list -m -u all` catches outdated deps), but the license decision is a human one.

### Coordinate Assignment

Innworld has no canonical coordinate system. We invented one using a hybrid approach:

- **Seed coordinates.** We manually placed ~80 major landmarks in a [-512, 512] coordinate space: continent centers (Izril, Chandrar, Terandria, Baleros, Rhir, Drath), major cities within each continent (Liscor, Pallass, Invrisil on Izril; Reim, Khelt on Chandrar; Ailendamus, Calanfer on Terandria), and key landmarks (The Wandering Inn, Blood Fields, High Passes, Wistram Academy).
- **Containment-based inheritance.** The remaining 170+ locations inherit coordinates from their parents in the containment hierarchy. If we know "Tails and Scales" is in "Liscor" and we have coordinates for Liscor, the tavern gets placed near Liscor with a small random offset. The system walks up to 10 levels of containment to find a positioned ancestor.
- **Keyword-based traceability.** Some locations have containment data that doesn't chain to a seed. If a location's name contains a known geographic keyword (continent name, major city, cultural group), it's considered traceable even without explicit containment. This expanded coverage significantly without sacrificing accuracy.

### Dynamic Landmass Rendering

Continents aren't drawn from a static image. They're generated dynamically from the locations visible at the current chapter position: locations are grouped by continent via containment chains and proximity, a convex-hull-like algorithm with organic noise generates coastlines, and each continent gets a deterministic color derived from its name. Landmasses grow and reshape as more locations are discovered through reading.

## The Numbers

| METRIC | VALUE |
|---|---|
| Total words processed | ~12,000,000 |
| Chapters analyzed | 807 |
| Extraction success rate | 100% |
| Locations extracted | 600 (after quality filtering) |
| Spatial relationships mapped | 2,485 |
| Containment rules (hierarchical) | 1,555 |
| Lines of code written | ~4,200 |
| External dependencies | 4 |
| Development time | 2 days |
| Team size | 1 engineer |
| Hand-written code | 0 lines — 100% AI-generated |
| Commits to completion | 5 |

## How AI Made This Possible

### The Extraction

Each of the 807 chapters was sent to Claude as a complete document with a structured extraction prompt. The AI returned JSON containing every named location, its type, description, spatial relationships to other locations, and direct quotes from the source text.

Later volumes of the serial have individual chapters exceeding 300,000 characters — roughly the length of a full novel. The AI processed these without issue once we applied a technique called "assistant prefill" that mechanically anchors the output format (more on this below).

The extraction prompt uses a system message defining location types and relationship types with examples drawn from the source material. Key design decisions:

- **No series knowledge**: the prompt instructs the model to extract only from the provided text, preventing hallucinated locations from training data.
- **Quote requirements**: every extracted location and relationship must include a supporting quote from the chapter text, providing end-to-end traceability from map to source material.
- **Visual descriptions**: separate from functional descriptions, these capture terrain, architecture, climate, and atmosphere — data that could drive future visual rendering.

This is the same discipline required for any serious document processing — you need provenance, not guesses.

### The Development

The codebase itself was built interactively with Claude Code — an AI coding assistant. We described what we wanted, reviewed the generated code, and iterated. The AI wrote the scraper, the database layer, the aggregation logic, the coordinate assignment algorithm, and the frontend map. We directed architecture, reviewed outputs, and made judgment calls.

This is how Intelligrit operates: small teams augmented by AI, moving at a pace that traditional staffing models can't match.

## Technical Findings

### Finding 1: Structured Extraction from Long Documents Works — With the Right Technique

On very large documents (200K+ characters), the AI would occasionally ignore extraction instructions and produce a narrative summary instead of structured JSON. This happened on roughly 5% of the longest chapters.

The fix was a single line of code. The Anthropic API supports "assistant prefill" — you can include a partial assistant message that the model must continue from. By prefilling with `{`, we force the model to begin its response as JSON:

```
{
  "messages": [
    {"role": "user", "content": "Extract all geographical data... [chapter text]"},
    {"role": "assistant", "content": "{"}
  ]
}
```

The response comes back without the leading `{`, so we prepend it. This eliminated 100% of parse failures. The model never once produced invalid output after this change. The model's tendency to "forget" instructions scales with input length; anchoring the output format mechanically bypasses that failure mode entirely.

**Why this matters:** Long-document processing is everywhere — regulations, contracts, medical records, legal filings, research papers, customer support archives, compliance documentation. AI can reliably extract structured data from these documents, but the extraction pipeline needs mechanical safeguards, not just prompt engineering. This is an engineering discipline, not a magic trick.

### Finding 2: AI Selects Outdated Dependencies

The AI initially chose a deprecated database driver (`marcboeker/go-duckdb` v1.8.5) instead of the current official driver (`duckdb/duckdb-go` v2.5.5). The old driver had a real bug: large transactions silently failed to persist data. We spent time diagnosing and working around this before identifying the root cause — the fix was simply using the current version.

**Why this matters:** AI coding assistants are powerful but their training data has a cutoff. They will confidently use deprecated libraries, outdated APIs, and superseded security practices. Human oversight on dependency selection, security posture, and architectural decisions remains essential. AI amplifies engineers; it doesn't replace engineering judgment.

### Finding 3: Small Codebases Can Process Massive Datasets

3,200 lines of Go and 1,000 lines of JavaScript process 12 million words, extract structured data via AI, store it in an embedded database, and serve an interactive map with access controls. The compiled binary is a single executable with all assets embedded.

**Why this matters:** Modernization doesn't require million-line codebases or armies of developers. Focused systems built with modern tools and AI assistance can replace sprawling legacy applications. Smaller codebases mean lower maintenance costs, fewer security vulnerabilities, faster security reviews, and easier knowledge transfer — whether you're navigating an Authority to Operate or a startup's first SOC 2 audit.

### Finding 4: Embedded Databases Enable Zero-Infrastructure Prototyping

We used DuckDB — an embedded analytical database that stores everything in a single file. No server to install, no container to manage, no infrastructure team to coordinate with. The entire system runs on a laptop. DuckDB is columnar and optimized for analytical workloads, so aggregation queries across 807 chapters of extraction data run fast.

The deliberate bet is that DuckDB gives maximum velocity now while preserving optionality. The schema is standard SQL, the queries are standard SQL, and Go's `database/sql` interface abstracts the driver. If the project ever needs concurrent users, real-time updates, or multiple data sources, we swap `go-duckdb` for `pgx`, update the connection string, and adjust a handful of dialect differences. The application code doesn't change. Start with the simplest thing that works, but make choices that don't paint you into a corner.

**Why this matters:** The fastest path to demonstrating value is removing infrastructure barriers. A working prototype that runs on a laptop, with no cloud provisioning or infrastructure approval, can prove feasibility in days instead of months. The migration path to production infrastructure is planned from the start but doesn't block the initial demonstration of capability. This is how you win a demo day, close a client, or prove to leadership that a problem is solvable — before anyone writes a procurement document.

### Finding 5: Evidence-Backed AI Output Changes the Trust Equation

Every spatial relationship on the map is clickable. When a reader clicks the dashed line connecting two locations, a popup shows the relationship type, the extracted detail, the **exact quote from the source text** that established the connection, and the chapter where it first appeared.

This solves a fundamental trust problem with AI-extracted data. When an LLM tells you that City A is "two weeks' sail from City B," the natural question is: *says who?* By threading source quotes through the entire pipeline — from extraction prompt, through aggregation, through the database, to the frontend popup — every claim on the map is verifiable against the original text. The user doesn't have to trust the AI. They can check.

The engineering cost was minimal: the quote is just another field in the extraction schema, stored alongside the relationship it supports, and rendered when the user asks for it. We carry provenance for 2,500+ relationships across 807 chapters with no special infrastructure — it's just data flowing through the same pipeline as everything else.

**Why this matters:** Any system that uses AI to extract structured data from documents faces the same credibility question — whether the documents are government contracts, medical records, legal filings, or customer feedback archives. An AI assertion without provenance is just a guess. An AI assertion linked to the exact sentence in the source document is evidence. The difference between "our AI found this" and "our AI found this, and here's the sentence it found it in" is the difference between a tool that analysts distrust and one they rely on. End-to-end provenance should be a baseline requirement for any AI extraction system, and this project demonstrates it can be carried at scale with trivial engineering overhead.

### Finding 6: Accessibility Can't Be an Afterthought — Even on a Map

Interactive maps are inherently visual, which makes them one of the harder interfaces to make accessible. We audited the TWI Map against WCAG guidelines — first manually, then with axe-core automated scanning — and iteratively fixed every issue until the scanner reported **zero violations across 43 rule checks**.

Here’s what we built:

- **Aggressive label scaling.** Map text labels now scale 4x between minimum and maximum zoom, keeping city and town names readable at every zoom level. At high zoom, labels appear for *all* location types — not just major ones — so the map remains informative as you explore regions.
- **Tabbable map markers.** Every location marker on the map — circle markers and icon markers alike — receives `tabindex="0"`, `role="button"`, and an `aria-label` describing the location name and type. Users can Tab through all markers on the map; focusing a marker opens its detail popup, which is announced via the live region (below). This makes the map itself explorable by keyboard, not just the sidebar. The map container uses `role="application"` with `aria-roledescription="interactive map"` to signal that it’s a complex widget with custom keyboard interaction.
- **Full keyboard navigation.** The sidebar location list supports arrow keys, Enter to pan-and-zoom to a location (opening its detail popup), and Space to toggle visibility. Screen reader users navigate the same data as mouse users. Between the tabbable map markers and the sidebar list, every location is reachable without a mouse.
- **ARIA semantics throughout.** The chapter navigation dropdowns have descriptive `aria-label` attributes. The location list uses `role="list"` with `aria-label` on each item describing the location name, type, visibility state, and mention count. Navigation landmarks (`nav`, `main`, `aside`) and skip links let keyboard users jump directly to the map or location list.
- **WCAG AA contrast.** All text colors meet the 4.5:1 contrast ratio against their backgrounds — verified by axe-core automated scanning. This includes popup headings, sidebar headings, metadata text, and even third-party elements like Leaflet’s attribution link, which required a CSS override to bring into compliance.
- **Focus indicators.** Every interactive element — sidebar items, filter chips, checkboxes, map markers — shows a visible `focus-visible` ring for keyboard users.
- **Screen reader popup announcements.** When any popup opens — from clicking a marker, a relationship line, a provenance ghost line, or a proximity click on nearby empty space — the full content is pushed to an `aria-live="assertive"` region. Screen readers announce the location name, type, description, visual description, mention count, and aliases without the user needing to navigate into the popup DOM. On popup close, the live region is cleared so stale content isn’t re-announced. This matters because Leaflet popups are injected dynamically into the DOM; without an explicit live region, screen readers have no way to discover that new content appeared.
- **Proximity click as an accessibility feature.** Clicking anywhere on the map opens the nearest marker or line popup within a 40-pixel radius. This was designed for mouse usability on dense maps, but it doubles as an accessibility feature: users with motor impairments who can’t precisely target a 6-pixel circle marker still get the popup they’re reaching for. The system checks perpendicular distance to line segments — not just endpoints — so relationship lines are discoverable without pixel-perfect aim.

The axe-core scan passes with **zero violations** — on an interactive map with 100+ tabbable markers, dynamic popups, layered relationship lines, and three-dropdown navigation. This isn’t a static page; it’s a complex data visualization with real-time filtering.

**Why this matters:** Accessibility isn’t just a compliance checkbox — though Section 508 makes it a legal requirement for federal systems. Accessible interfaces serve *all* users better: users on low-resolution displays, users with temporary impairments, users in degraded environments, and users who simply prefer keyboard navigation. Building accessibility into a weekend project demonstrates that it doesn’t require a separate “accessibility phase” — it’s a set of engineering decisions applied at the same time as every other feature. The popup announcement pattern is worth highlighting: any web application that injects dynamic content (modals, toasts, live search results) faces the same problem. An `aria-live` region is a single HTML element and a few lines of JavaScript — trivial engineering cost for a dramatic screen reader experience improvement. Running axe-core as part of the development loop — not as a post-delivery audit — means violations are caught and fixed in the same session they're introduced.

### Finding 7: Progressive Disclosure Creates Cascading UI Complexity

The spoiler-free slider seemed simple at first — filter locations by chapter. But real users don't all consume the story the same way. *The Wandering Inn* is available as a web serial (807 chapters across 10 volumes), audiobooks (17 books covering chapters 1–429), and ebooks (17 books, same range). A reader on Audiobook Book 7 needs to see a different slice of the world than a web serial reader on Volume 7.

This created cascading requirements:

- **Multi-format chapter mapping.** The chapter navigation had to dynamically reconfigure based on reading format. Audiobook readers see "Book 7, Chapter 5.09" while web serial readers see "Volume 5, 5.09." What started as a slider with a single jump dropdown evolved into a three-dropdown system — format, section, and chapter — because precision navigation matters when there are 807 chapters. A reader jumping to "Audiobook Book 7, Chapter 5" shouldn't have to drag a slider through hundreds of positions. The section dropdown repopulates when the format changes, the chapter dropdown repopulates when the section changes, and all three stay synchronized as the user navigates.
- **Coherent relationship filtering.** When a user hides locations to focus on two specific places, relationship lines between hidden locations must also disappear. But the chapter filter already gates which relationships exist in the data. These two filters compose: the API returns only relationships that exist at the current chapter, and the frontend draws only those between currently visible locations. Getting both layers correct — and verifying they compose correctly — required Playwright-driven testing across format and visibility combinations. (This filtering later revealed a deeper problem: hiding locations destroyed all spatial context. See Finding 9.)
- **Persistent spatial context.** When users hide most locations to focus on a few, the map becomes a featureless void. Continent outlines need to persist regardless of location visibility so users stay geographically oriented. This required separating the landmass rendering pipeline (which uses all locations with coordinates) from the marker rendering pipeline (which respects hide/type filters).
- **Searchable, toggleable sidebar.** With 250+ locations, users need to find specific places quickly. The sidebar gained search (filtering by name, aliases, and type), bulk hide/show controls, per-location visibility toggles, and a separate "locate" button to pan the map — since clicking a location now toggles its visibility rather than jumping to it.

None of these features were in the original design. Each emerged from watching real interaction patterns and asking "what would a reader actually need?" The total JavaScript grew from 350 to over 1,000 lines, but each addition was a direct response to a usability gap, not speculative complexity.

**Why this matters:** Access control is never just a boolean. Whether it's federal role-based views, SaaS tiered feature access, or research data with embargo periods — progressive disclosure creates UI complexity that compounds. Each filter interacts with every other filter. Building these interactions incrementally, with automated testing at each step, keeps the complexity manageable. Trying to design all the interactions upfront would have been slower and less accurate than the iterative approach.

### Finding 8: Playwright Enables AI-Driven Exploratory Testing

We used Playwright — a browser automation framework — to let the AI assistant directly interact with the running map during development. Rather than describing bugs in words and waiting for manual reproduction, the AI navigated the map, manipulated the chapter slider, toggled filters, searched locations, and took screenshots to verify behavior — all programmatically.

This turned debugging from a back-and-forth conversation (“click hide all, then show these two locations, what do you see?”) into an autonomous loop: the AI operates the UI, inspects the results, reads console logs, and identifies the root cause without human intervention. When we suspected a relationship filtering bug, the AI loaded the map, set the slider to chapter 807, clicked “Hide all,” searched for and un-hid two specific locations, then checked console output to confirm exactly 2 relationships were drawn with 2,483 skipped — proving the filter was working correctly and the issue was a stale browser cache.

**Why this matters:** Rigorous testing documentation and reproducible test procedures aren’t just a government requirement — they’re good engineering. Playwright scripts serve as both the test execution engine and the test documentation — every step is recorded, every assertion is verifiable. Combined with AI-assisted development, this means the AI that wrote the feature can also write and execute the acceptance tests, creating a closed loop from requirement to implementation to verification. For systems requiring IV&V, SOC 2 compliance, or ISO certification, automated browser-level tests provide an auditable trail that manual testing cannot match.

### Finding 9: Hiding Data Destroys Spatial Context — Ghost Lines Restore It

The map draws relationship lines between visible locations — dashed lines showing connections like “three days’ travel” or “visible from the city walls.” When a reader hides most locations to focus on just two or three, every relationship line disappears because both endpoints must be visible. The map becomes a handful of dots floating in empty space with no indication of why they’re positioned where they are.

This is a direct consequence of the progressive disclosure model. The same chapter-filtered relationship data that prevents spoilers also prevents spatial context when the user exercises their hide/show controls. The two features work correctly in isolation but compose poorly: relationships require two visible endpoints, and hiding locations removes endpoints.

We solved this with “provenance lines” — ghost relationship lines that draw from a visible location to the*coordinates* of a hidden related location. Key design decisions:

- **Exactly one visible endpoint.** Lines only draw when one location is visible and the other is hidden. If both are visible, the normal solid relationship line handles it. If both are hidden, neither is relevant to the current view.
- **Coordinates without markers.** The hidden endpoint’s position comes from the coordinate dataset, which contains all 600 locations regardless of visibility filters. The ghost line terminates at the correct map position with a faint dot and a small label naming the hidden location — so the reader sees where the connection points without cluttering the map.
- **Distinct visual treatment.** Ghost lines use low opacity (`#ffffff30`), thinner dash patterns (`2 6` vs `4 4`), and faded endpoint markers. They’re visible when you look for them but don’t compete with normal relationship lines or location markers.
- **Discoverable interaction via hit areas.** Thin lines are nearly impossible to click with a mouse. Each ghost line has an invisible 12-pixel-wide hit area layered on top that captures hover and click events. Hovering brightens the visible line underneath as a visual cue that it’s interactive. This is a standard cartographic technique — the visual line and the interaction target are separate elements with different geometries.

- **Proximity click.** Beyond the hit areas, clicking anywhere on the map selects the nearest marker or line within a 40-pixel radius. This means imprecise clicks — inevitable on a dense map with small targets — still connect users to the data they're reaching for. The system checks both marker positions and perpendicular distance to line segments, so clicking near a relationship line's midpoint works even if the click doesn't land directly on the path.
- **Full popup on click.** Despite the faded visual, clicking a ghost line (or clicking near one) opens the same relationship popup as a normal line — type, detail, source quote, chapter reference. The provenance chain remains intact even for ghost connections.
- **Off by default.** Ghost lines add visual noise for readers who are showing most locations. The feature is a separate toggle ("Show provenance") that persists to localStorage alongside the existing "Show relationships" toggle.

The result: a reader who hides everything except The Wandering Inn and Riverfarm now sees faded lines radiating to Liscor, Invrisil, Celum, and other connected locations. Without showing those locations on the map, the reader still understands the spatial web that positions their selected locations.

**Why this matters:** Any system with layered access controls — classification levels, subscription tiers, role-based views, progressive disclosure — faces this same composition problem. Filtering data for one purpose (preventing spoilers) can destroy context needed for another purpose (understanding spatial relationships). The solution isn't to weaken either filter, but to add a third layer that explicitly bridges the gap. In analytics dashboards, this pattern appears when a filtered view removes the reference points users need to interpret the remaining data. Ghost references — faded, clearly marked as outside the current filter, but positionally accurate — let users maintain orientation without violating the filter's intent.

### Finding 10: AI Labor Makes Polish Economically Viable

The TWI Map has a long tail of small quality-of-life features: proximity click (40-pixel snap to nearest marker), three-dropdown chapter navigation, ghost provenance lines with invisible hit areas and hover highlights, tabbable map markers with screen reader announcements, WCAG AA contrast on every element including third-party attribution links. Each of these is individually straightforward — a few hours of implementation, testing, and iteration. Collectively, they're the difference between a demo and a product.

In traditional development, this kind of polish is the first thing cut. When engineering time costs $150–250/hour and each small feature requires writing code, writing tests, running accessibility scans, fixing regressions, and updating documentation — the math doesn't work. A project manager looks at "make ghost lines hoverable" and estimates 4 hours of developer time for a feature most users won't notice. It gets deprioritized. The product ships with rough edges. Users adapt to the roughness and accept it as normal.

AI-augmented development inverts this calculus. The same polish that would take a team days of careful implementation takes an AI-augmented engineer hours. The provenance-aware filtering rule — "keep low-mention locations if they're relationship endpoints for established locations" — took one conversation to specify and implement. The accessibility scan-fix-rescan loop ran three times in a single session until axe-core reported zero violations. Adding tabbable markers to a Leaflet map (a notoriously inaccessible widget) was a focused change to the render loop plus a few ARIA attributes.

The aggregate effect is that the lower bound of acceptable quality rises. Features that were once "nice to have" become table stakes when the marginal cost of implementing them approaches zero. Accessibility compliance, interaction polish, edge-case handling — these aren't luxuries. They're just work. And when AI makes the work cheap, there's no excuse to skip it.

**Why this matters:** Every software product — government, enterprise, startup — accumulates rough edges that persist because fixing them is "too expensive for the value." Awkward navigation flows, inaccessible interfaces, missing keyboard support, inconsistent filtering behavior — each one individually seems minor, but they compound into systems that frustrate users and fail audits. AI-augmented development doesn't just make big features faster; it makes small fixes and incremental polish economically viable at a scale that traditional staffing models can't match. The question isn't "can we afford to fix this?" — it's "why haven't we fixed this already?"

## What This Demonstrates About Intelligrit

This project wasn't built for a client. It was a weekend experiment — the kind of thing our team does because we're curious and the tools make it possible. But it demonstrates exactly how we work on every engagement:

- **Small teams, big output.** One engineer in two days. Not because we cut corners, but because AI handles the volume work while the human handles the judgment calls.
- **Ship working software, not slide decks.** Five commits and a working interactive map. Every Intelligrit engagement includes working demos, not just status reports.
- **Build for the real constraints.** The spoiler-free slider isn't just a feature — it's an access control mechanism. Different users see different data based on their authorization level. The same architecture applies to classified systems, subscription tiers, role-based dashboards, and progressive disclosure of sensitive information.
- **Provenance by default.** Every AI-extracted relationship on the map links back to the exact quote from the source text. Users click a connection and see the evidence. This isn't a separate audit layer — it's a field in the extraction schema that flows through the entire pipeline. Trust in AI output starts with traceability.
- **AI-native from day one.** We don't bolt AI onto existing processes. AI is in the extraction pipeline, in the development workflow, in the dependency management. It's how we achieve 20x efficiency without 20x headcount.
- **Pragmatic technology choices.** Go for the backend because it compiles to a single binary. Plain HTML/CSS/JS for the frontend because the problem doesn't need a framework. DuckDB for storage because it requires zero infrastructure — but the data layer is standard SQL behind Go's `database/sql` interface, so dropping in PostgreSQL for production is a connection string change, not a rewrite. Leaflet.js for the map because it's lightweight, proven, and BSD-licensed. Every choice optimizes for shipping fast without painting into a corner.
- **Open source by default.** Every dependency is permissively licensed — MIT, BSD, or Apache 2.0. No viral licenses, no license ambiguity, no surprises during procurement or legal review. We audit dependencies deliberately because AI assistants don't check licenses.
- **Accessibility as engineering, not afterthought.** WCAG AA contrast, keyboard navigation, ARIA semantics, skip links, screen reader popup announcements, and proximity click for motor accessibility — applied to an interactive map, one of the hardest interfaces to make accessible. Accessibility built in from the start, not bolted on after delivery.

## Future Work

- **Improved coordinate inference**: using directional relationships ("north of Liscor") and distance mentions ("three days ride") to algorithmically position locations rather than hand-seeding.

- **Visual rendering**: the extraction captures visual descriptions (terrain, architecture, climate). These could drive AI image generation for location portraits or stylized map tiles.
- **Incremental updates**: as new chapters publish, run only the new chapter through extraction and re-aggregate.
- **Deep search**: full-text search across location descriptions and relationship quotes, beyond the current name/type sidebar search.

## The Bigger Picture

Every organization sits on enormous volumes of unstructured data — regulations, correspondence, case files, research papers, customer feedback, support tickets, procurement documents. Extracting structured, actionable information from these documents has traditionally required large teams doing manual review over months or years.

The techniques demonstrated in this project — AI-powered extraction with mechanical output guarantees, automated quality filtering, progressive access controls, zero-infrastructure prototyping, permissive open source licensing, plain-HTML frontends that ship without a build pipeline — apply to any domain where messy real-world data needs to become usable structured information. Federal agencies, research institutions, startups, enterprises — the shape of the problem is the same.

The difference isn't theoretical. One engineer processed 12 million words in two days.

The question for leaders isn't whether AI can do this work. It's how fast they want to start.

---

**Intelligrit LLC** — Intelligence, Integrity, Results

Ship Faster. Modernize Smarter. Deliver Better.

contact@intelligrit.com | intelligrit.com

## Acknowledgments

Built with Claude (Anthropic) for both the extraction pipeline and development assistance via Claude Code. Map rendering powered by Leaflet.js. Storage by DuckDB. *The Wandering Inn* is written by pirateaba and published at wanderinginn.com.