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.

> “
>
> *We didn't build this for a client. We built it because we wanted to see if we could.*

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

Scrape — Download 807 chapters from the web (rate-limited, resumable)

Extract — Send each chapter to Claude AI for structured location extraction

Aggregate — Deduplicate, normalize, and merge results across all chapters

Coordinate — Assign map positions using containment hierarchies and seed data

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.

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.

AI writes it cleanly. AI coding assistants produce excellent vanilla JS because the browser API is abundantly represented in training data.

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.

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).

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.

Containment-based inheritance. The remaining 170+ locations inherit coordinates from their parents in the containment hierarchy. 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, it's considered traceable even without explicit containment.

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.

100%

EXTRACTION RATE

2,485
RELATIONSHIPS

4,200
LINES OF CODE

0
HAND-WRITTEN LINES

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.
- **Visual descriptions**: separate from functional descriptions, these capture terrain, architecture, climate, and atmosphere.

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.

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": "{"}
  ]
}
```

This eliminated 100% of parse failures. The model never once produced invalid output after this change.

Why this matters: Long-document processing is everywhere — regulations, contracts, medical records, legal filings. 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.

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.

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.

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 or real-time updates, we swap the driver and adjust a handful of dialect differences.

Why this matters: The fastest path to demonstrating value is removing infrastructure barriers. A working prototype that runs on a laptop can prove feasibility in days instead of months — 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.

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.

Why this matters: Any system that uses AI to extract structured data faces the same credibility question. An AI assertion without provenance is just a guess. An AI assertion linked to the exact sentence in the source document is evidence. End-to-end provenance should be a baseline requirement for any AI extraction system.

> “
>
> *An AI assertion without provenance is just a guess. An AI assertion linked to the exact sentence in the source document is evidence.*

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**.

What we built:

- **Aggressive label scaling** — Map text labels scale 4x between minimum and maximum zoom
- **Tabbable map markers** — Every marker receives `tabindex="0"` , `role="button"` , and an `aria-label`
- **Full keyboard navigation** — Sidebar supports arrow keys, Enter, and Space
- **ARIA semantics throughout** — Descriptive labels, navigation landmarks, skip links
- **WCAG AA contrast** — All text meets 4.5:1 contrast ratio
- **Screen reader popup announcements** — via `aria-live="assertive"` region
- **Proximity click** — 40-pixel snap radius for motor accessibility

Why this matters: Accessibility isn't just a compliance checkbox — though Section 508 makes it a legal requirement for federal systems. 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.

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), 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** — Three-dropdown navigation (format, section, chapter)
- **Coherent relationship filtering** — Two filters that compose correctly
- **Persistent spatial context** — Continent outlines persist regardless of location visibility
- **Searchable, toggleable sidebar** — Search, bulk controls, per-location toggles

Why this matters: Access control is never just a boolean. Whether it's federal role-based views, SaaS tiered access, or research data with embargo periods — progressive disclosure creates UI complexity that compounds. Building incrementally with automated testing keeps the complexity manageable.

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 controls, toggled filters, and took screenshots to verify behavior — all programmatically.

This turned debugging from a back-and-forth conversation into an autonomous loop: the AI operates the UI, inspects the results, reads console logs, and identifies the root cause without human intervention.

Why this matters: Playwright scripts serve as both the test execution engine and the test documentation — every step is recorded, every assertion is verifiable. 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

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.

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

- **Exactly one visible endpoint** — Lines only draw when one location is visible and the other is hidden
- **Coordinates without markers** — The hidden endpoint's position comes from the coordinate dataset
- **Distinct visual treatment** — Low opacity, thinner dash patterns, faded endpoint markers
- **Discoverable interaction** — 12-pixel-wide invisible hit areas with hover highlights
- **Full popup on click** — Same relationship details as normal lines
- **Off by default** — Separate toggle, persists to localStorage

Why this matters: Any system with layered access controls faces this same composition problem. Filtering data for one purpose can destroy context needed for another. 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 quality-of-life features: proximity click, three-dropdown navigation, ghost provenance lines with invisible hit areas, tabbable map markers with screen reader announcements, WCAG AA contrast on every element. Each is individually straightforward. 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 scans, fixing regressions — the math doesn't work.

AI-augmented development inverts this calculus. The accessibility scan-fix-rescan loop ran three times in a single session until axe-core reported zero violations. Adding tabbable markers to Leaflet was a focused change to the render loop plus a few ARIA attributes.

Why this matters: Every software product accumulates rough edges that persist because fixing them is "too expensive for the value." 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.

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 is an access control mechanism. The same architecture applies to classified systems, subscription tiers, role-based dashboards.
- **Provenance by default.** Every AI-extracted relationship links back to the exact quote from the source text. Trust in AI output starts with traceability.
- **AI-native from day one.** 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 because it compiles to a single binary. Plain HTML/CSS/JS because the problem doesn't need a framework. DuckDB because it requires zero infrastructure. Every choice optimizes for shipping fast without painting into a corner.
- **Open source by default.** Every dependency is permissively licensed. 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 announcements, proximity click — applied to an interactive map, one of the hardest interfaces to make accessible.

Future Work

- **Improved coordinate inference**: using directional relationships and distance mentions to algorithmically position locations rather than hand-seeding.
- **Visual rendering**: the extraction captures visual descriptions that 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.

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.

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.*

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.