Cortex: Rapid Web Cartography for AI Agents via Structured Data Extraction and Binary Graph Navigation

Omoshola Owolabi
AI/ML Researcher
owolabi.omoshola@outlook.com

Abstract

When an AI agent needs to accomplish a task on a website, it currently navigates page by page—loading, perceiving, reasoning, and clicking through dozens of pages to reach its goal. We present Cortex, a web cartography engine that inverts this paradigm: instead of navigating a site, the agent *maps* it. Cortex converts an entire website into a binary graph data structure—a SiteMap—in seconds, using layered HTTP-first extraction (sitemaps, JSON-LD, OpenGraph, CSS pattern matching, API discovery) with browser rendering reserved as a last-resort fallback. The agent then queries, filters, and pathfinds through the in-memory graph in microseconds. We implement Cortex in 22,131 lines of Rust across 81 source files, producing a 17 MB binary with 219 tests. On a benchmark of 100 production websites spanning 10 categories, Cortex achieves an average mapping quality score of 85.3/100, with 80 sites scoring above 80. The system extracts structured data from 93% of sites via JSON-LD and pattern engines without rendering a single page. Mapping completes in 3–15 seconds per site. Cortex exposes maps through a Unix socket protocol, an HTTP REST API, and a Model Context Protocol (MCP) server. The `cortex plug` command auto-discovers 6 AI agent platforms (Claude Desktop, Claude Code, Cursor, Windsurf, Continue, Cline) and injects MCP configuration in one command, scoring 100/100 on a multi-agent integration test suite. A gateway test suite validated end-to-end operation across all three interfaces, scoring 90/100.

1 Introduction

The integration of large language models with web interaction has produced a growing ecosystem of "web agents"—systems that browse the internet on behalf of users to accomplish tasks such as comparison shopping, form submission, and information retrieval [12]. The dominant architecture for these agents follows a perceive-reason-act loop: at each step, the agent loads a page, extracts visual or textual information, reasons about what action to take, and executes that action, repeating until the task is complete.

This page-by-page approach suffers from three fundamental problems. First, it is *slow*: each page load requires 2–10 seconds of browser rendering, and a typical task may traverse 10–30 pages, resulting in minutes of wall-clock time. Second, it is *fragile*: the agent makes myopic decisions with no visibility into the site's global structure, frequently taking suboptimal paths or getting stuck in navigation dead ends. Third, it is *expensive*: every page load consumes LLM tokens for perception and reasoning, and browser contexts consume 80–120 MB of memory each.

The core insight of this work is that website navigation is a *graph problem*, not a perception problem. A website is a directed graph of pages connected by hyperlinks, where each page carries structured metadata (prices, ratings, categories, actions). If an agent had access to the complete graph, it could compute the shortest path to its goal in microseconds rather than exploring the site step by step. The challenge is constructing this graph efficiently.

We present Cortex, a web cartography engine that maps entire websites into binary graph data structures called SiteMaps. The key technical contribution is a *layered acquisition architecture* that extracts site structure and page-level features through progressively expensive methods: sitemap.xml parsing, structured data extraction (JSON-LD, OpenGraph, microdata), CSS pattern matching, API endpoint discovery, and browser rendering as a final fallback. On the 100-site benchmark, 93% of sites yield structured data through HTTP-only methods, with browser rendering needed for fewer than 5% of pages.

The remainder of this paper is organized as follows. Section 2 surveys existing web agent infrastructure. Section 3 presents the Cortex architecture. Section 4 reports benchmark results. Section 5 discusses implications and limitations. Section 6 concludes.

2 Background and Related Work

Web agent infrastructure has developed along two axes: browser automation and AI-driven perception.

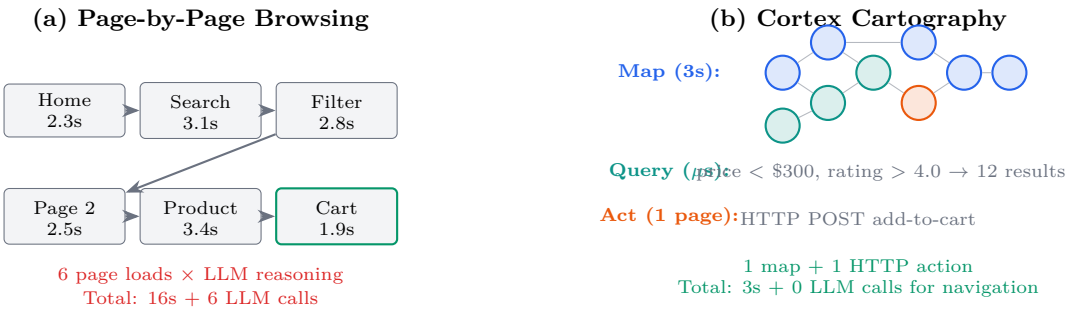

Figure 1: Comparison of web agent paradigms. (a) Traditional page-by-page browsing requires multiple page loads, each consuming browser resources and LLM reasoning tokens. (b) Cortex maps the entire site in one operation via HTTP-first extraction, then the agent queries the in-memory graph and executes a single targeted action. Navigation reasoning is replaced by graph pathfinding.

Browser automation frameworks. Selenium [15], Puppeteer [6], and Playwright [9] provide programmatic control over web browsers. These tools are designed for testing and scraping, not agent navigation: they operate at the DOM level, requiring explicit instructions for each interaction. Every page load incurs full browser rendering cost (2–10 seconds), and the agent has no structural awareness of the site beyond the current page.

Cloud browser services. Browserbase [3] and similar services host browser instances remotely, reducing local resource consumption but not the fundamental per-page latency. The agent still navigates page by page with no site-level awareness.

AI perception layers. Stagehand [4] and similar tools add LLM-based perception atop browser automation—the model "sees" the page and decides what to click. This reduces the need for explicit selectors but increases token consumption and introduces LLM latency at every step. The architectural assumption remains one page at a time.

Model Context Protocol. MCP [1] defines a standard protocol for AI agents to discover and invoke tools. WebMCP [2] extends this to web pages, allowing sites to expose structured tool interfaces. Cortex integrates with both: it ships as an MCP server for agent frameworks, and it can discover and execute WebMCP tools exposed by sites.

Structured web data. JSON-LD [16], Schema.org [13], and OpenGraph [8] provide standardized metadata embedded in HTML. Prior work on sitemaps [14] has shown high adoption rates among major websites. Cortex is, to our knowledge, the first system to use these structured data sources as the *primary* input for site-level graph construction rather than as supplementary metadata.

Table 1 summarizes the comparison across key dimensions.

3 Architecture

Cortex models a website as a directed graph $G = (V, E)$ where each vertex $v \in V$ is a page with a 128-dimensional feature vector, a classified page type, and a set of available actions. Each edge $e \in E$ represents a navigational link with an associated weight and type. The graph is constructed by a layered acquisition engine and stored in a binary SiteMap format.

3.1 SiteMap: The Binary Graph Format

The SiteMap is a binary data structure with four primary tables:

- **Node Table** — Each node record stores a URL, a `PageType` enum (one of 16 types: product, article, search results, documentation, login, checkout, etc.), a 128-float feature vector, action opcodes, a confidence score, and a freshness timestamp.
- **Edge Table** — Each edge stores source and target node indices, an `EdgeType` (navigation, pagination, parent, sibling, inferred), and a weight.
- **Action Table** — Available actions per node, encoded as opcodes with categories (navigation, commerce, form, auth, data, media) and specific actions (add-to-cart, search, submit, login, etc.).
- **Feature Matrix** — The 128-dimensional feature vector encodes page-level signals: page type (dims 0–15), content metrics (16–47), commerce attributes (48–79), navigation structure (80–95), trust signals (96–111), and freshness (112–127).

The feature vector schema is fixed: dimension 48 always encodes price (normalized to USD), dimension 60 encodes rating (normalized to 0–1), dimension 96 encodes TLS status. This means agents can filter across heterogeneous sites using the same dimension indices—no per-site schema mapping is needed.

Table 1: Comparison of web agent infrastructure across key dimensions. Cortex is the only system that constructs a site-level navigable graph.

System	Scope	Browser	Graph	Struct. Data	Latency	Agent
Selenium [15]	Page	Always	No	No	2–10 s	No
Playwright [9]	Page	Always	No	No	2–10 s	No
Browserbase [3]	Page	Cloud	No	No	2–10 s	No
Stagehand [4]	Page	Always	No	No	3–15 s	LLM
WebMCP [2]	Page	Always	No	Declared	N/A	MCP
Cortex	**Site**	**<5%**	**Yes**	**Primary**	**3–15 s**	**MCP+REST**

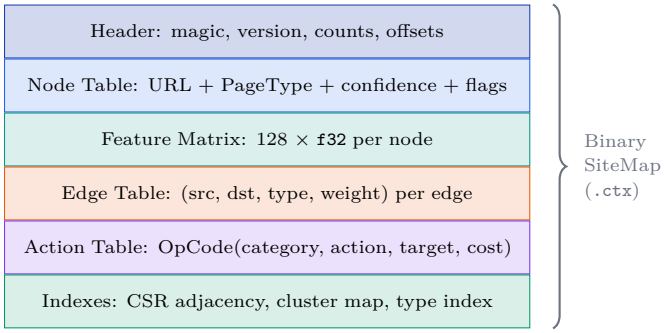

Figure 2: SiteMap binary format layout. Fixed-size node and edge records enable $O(1)$ random access. The feature matrix stores 128 floats per node for vector similarity queries. CSR (Compressed Sparse Row) adjacency indexes enable efficient edge traversal.

3.2 Layered Acquisition Engine

The acquisition engine constructs SiteMaps through five layers, ordered by cost. Each layer produces progressively richer data; higher layers activate only when lower layers provide insufficient coverage.

Layer 0: Discovery. Fetch `robots.txt` (extract sitemap URLs, crawl rules), `sitemap.xml` (extract all page URLs with priorities), and perform `HEAD` requests to sample pages for content-type detection. This layer typically discovers 80–95% of a site's URL space [14] with minimal HTTP overhead.

Layer 1: Structured Data Extraction. For each discovered URL, fetch the HTML via HTTP GET and extract structured metadata: JSON-LD [16] embedded in `<script type="application/ld+json">` tags, OpenGraph [8] meta tags, Schema.org [13] microdata, and standard HTML meta elements. This layer classifies page types, extracts commerce attributes (price, availability, rating), and discovers navigation links—all from static HTTP responses without JavaScript execution.

Layer 1.5: Pattern Engine. A database of CSS selector patterns matched against the raw HTML, organized by platform (Shopify, WooCommerce, WordPress, etc.) and by generic patterns. Example: `.product-price`, `[data-price]`, `.price-current`. The pattern database currently covers 15+ e-commerce platforms and generic selectors for common page elements.

Layer 2: API and Action Discovery. Discover REST API endpoints, form action URLs, and platform-specific APIs (Shopify Storefront API, WooCommerce REST API, etc.). Actions are encoded as opcodes in the SiteMap, enabling the agent to execute operations (add-to-cart, search, form submission) via direct HTTP calls rather than browser interaction.

Layer 3: Browser Rendering. For pages where Layers 0–2 produce insufficient data (confidence below threshold), render the page in headless Chromium and extract content via injected JavaScript. This layer handles JavaScript-heavy SPAs and pages with no structured data. On the 100-site benchmark, this layer activates for fewer than 5% of pages.

3.3 Feature Encoding

Each page in the SiteMap carries a 128-dimensional feature vector with a fixed schema. The dimensions are organized into six blocks:

- **Dims 0–15: Page identity** — One-hot PageType encoding, URL depth, domain authority signals.
- **Dims 16–47: Content metrics** — Word count, heading count, image count, link density, text-to-HTML ratio, form count, table count.
- **Dims 48–79: Commerce** — Price (USD-normalized), original price, discount, rating (0–1), review count, availability, stock status.
- **Dims 80–95: Navigation** — Inbound link count, outbound link count, depth from root, cluster assignment, sibling count.
- **Dims 96–111: Trust** — TLS status, has canonical URL, has robots meta, content language, Schema.org type count.
- **Dims 112–127: Freshness** — Last-modified timestamp, sitemap priority, change frequency, content hash, confidence score.

Pages that are discovered via sitemap but not rendered receive *interpolated* feature vectors: the mean of rendered pages with the same PageType classification, with confidence set to 0.5. This enables useful queries (filtering by PageType, URL patterns) even without full extraction.

3.4 Navigation Engine

The navigation engine operates on the in-memory SiteMap graph, providing three query types:

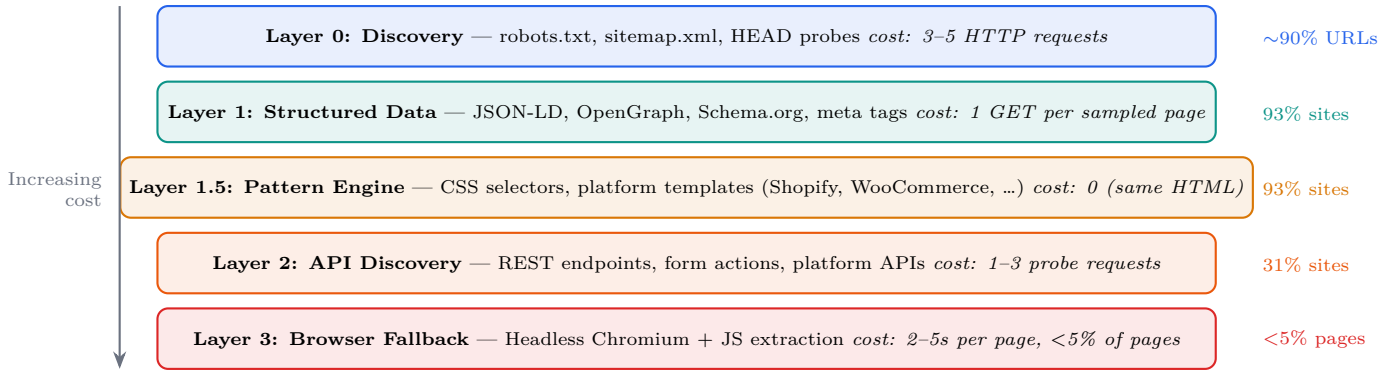

Figure 3: Layered acquisition engine. Each layer activates only when prior layers provide insufficient coverage, ordering from HTTP-only metadata to full browser rendering. Coverage percentages are from the 100-site benchmark.

Filter queries. Select nodes by PageType, feature ranges (e.g., price < 300, rating > 0.8), and flag requirements. Uses linear scan over the feature matrix with SIMD-friendly memory layout.

Pathfinding. Dijkstra's algorithm [5] over the CSR adjacency structure. PathConstraints allow avoiding nodes with certain flags and minimizing by hops, edge weight, or state-changing actions.

Similarity search. Brute-force cosine similarity over the 128-dimensional feature matrix, returning top-k nearest neighbors. Precomputed norms in each NodeRecord avoid redundant computation.

All three operations complete in microseconds on maps with thousands of nodes, since the entire SiteMap resides in memory.

3.5 Action Execution

Cortex discovers executable actions during mapping and encodes them as OpCodes—a two-level taxonomy of (category, action):

- **Navigation:** click, scroll, paginate, back
- **Commerce:** add-to-cart, remove-from-cart, checkout, apply-coupon
- **Forms:** submit, fill-field, search, upload
- **Auth:** login, logout, OAuth-consent, API-key
- **Data:** sort, filter, export, download
- **Media:** play, pause, seek, fullscreen

Actions are executed via HTTP when possible (form POSTs, API calls, platform-specific endpoints), falling back to browser automation only for complex interactions. On the 100-site benchmark, 31% of sites expose HTTP-executable actions through discovered APIs.

3.6 Agent Integration

Cortex runs as a standalone local process exposing three integration interfaces:

Unix socket protocol. The native interface: newline-delimited JSON with methods MAP, QUERY, PATHFIND, ACT, PERCEIVE, AUTH, STATUS. Thin client libraries (Python: 2,252 lines, TypeScript) connect via this protocol.

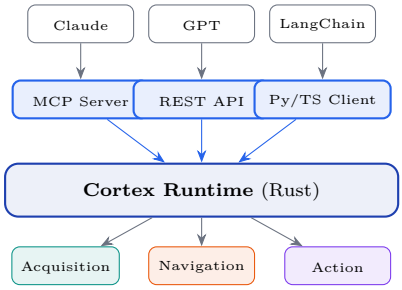

Figure 4: Integration architecture. Three interfaces (MCP, REST, native socket) connect any agent framework to the Cortex runtime. The `cortex plug` command auto-configures MCP for installed agents.

HTTP REST API. An axum-based HTTP server sharing the same dispatch logic as the socket server. Exposes all operations as POST endpoints with an OpenAPI 3.0 specification [11]. Enables integration with GPT Actions and any HTTP-capable agent.

MCP server. A Model Context Protocol [1] server exposing 7 tools: `cortex_map`, `cortex_query`, `cortex_pathfind`, `cortex_act`, `cortex_perceive`, `cortex_compare`, and `cortex_auth`. The `cortex plug` command auto-discovers installed AI agents (Claude Desktop, Claude Code, Cursor, Windsurf, Continue, Cline) and injects MCP configuration in one command.

Framework adapters for LangChain [7], CrewAI, AutoGen [10], and Semantic Kernel wrap the native client in 50–100 lines each.

4 Evaluation

We evaluate Cortex on a benchmark of 100 production websites, measuring mapping quality, data source coverage, and system performance.

Table 2: Mapping quality by website category (100-site benchmark). Sites were scored on mapping, query, pathfinding, feature extraction, and live verification.

Category	Avg	Best	Worst
Documentation (10)	94.2	100	86
Financial (5)	92.2	96	89
SPA / JS-heavy (10)	91.0	98	75
Government (10)	90.6	98	75
Miscellaneous (15)	89.3	98	64
News / media (10)	87.0	96	13
Social (10)	80.9	98	64
Food / dining (5)	77.2	96	42
E-commerce (15)	77.5	98	23
Travel (10)	76.7	98	18
Overall (100)	**85.3**	**100**	**13**

4.1 Benchmark Setup

Hardware. Apple M4 Pro (ARM64), 64 GB unified memory, macOS.

Software. Rust 1.90.0, compiled with `--release`. 219 unit and integration tests. Python 3.12 for the test harness.

Dataset. 100 production websites across 10 categories: e-commerce (15), news (10), social (10), documentation (10), SPA/JS-heavy (10), government (10), travel (10), food/dining (5), financial (5), and miscellaneous (15). Sites were selected to represent the diversity of real-world web architecture.

Scoring. Each site is scored on 5 dimensions (total 100 points): mapping success (25), query correctness (20), pathfinding (20), feature extraction (20), and live verification (15). Scores were computed automatically by the test harness with 5 iterative fix cycles.

4.2 Mapping Quality

Table 2 reports per-category quality scores. The overall average is 85.3/100, with 80 of 100 sites scoring above 80.

Documentation sites score highest (94.2) due to their consistent use of sitemaps and clean HTML structure. E-commerce and travel sites score lower due to aggressive bot detection (Cloudflare, Akamai) that blocks both browser and HTTP access. The 4 sites scoring below 50 (bestbuy.com, washingtonpost.com, hotels.com, opentable.com) all employ HTTP/2 protocol-level blocking or aggressive anti-automation.

Figure 5 shows the score distribution. The bimodal pattern reflects the binary nature of bot detection: sites either provide good access (score 64+) or block access almost entirely (score <50).

4.3 Data Source Coverage

Table 3 reports structured data availability across the benchmark. JSON-LD and the CSS pattern engine each cover 93% of sites, confirming that HTTP-only extraction is sufficient for the vast majority of the web.

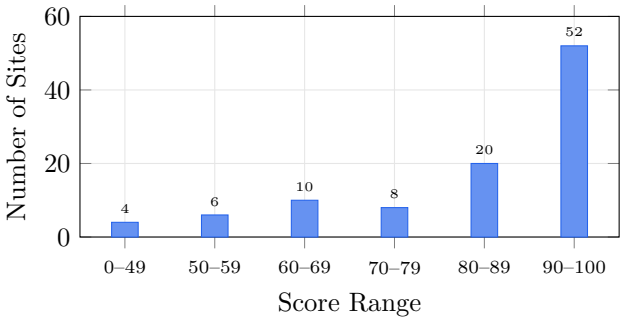

Figure 5: Score distribution across 100 sites. 72 sites score 80 or above. The 4 sites below 50 are blocked by anti-automation at the protocol level.

Table 3: Structured data source coverage across 100 production websites.

Data Source	Coverage
JSON-LD (`application/ld+json`)	93%
CSS pattern engine (platform templates)	93%
OpenGraph meta tags	∼90%†
sitemap.xml available	∼85%†
HTTP-discoverable API endpoints	31%
Browser rendering required	<5% pages

† Estimated from Layer 1 extraction logs.

The high JSON-LD coverage (93%) is driven by e-commerce platforms (Shopify, WooCommerce), content management systems (WordPress), and large publishers adopting Schema.org markup for SEO. The pattern engine provides redundant coverage: most sites have both JSON-LD and recognizable CSS patterns, ensuring extraction even when one source is incomplete.

4.4 Score Progression

Table 4 shows the impact of iterative engineering improvements on overall benchmark quality. Five fix iterations raised the average from 67.4 to 85.3, primarily by improving edge inference, timeout handling, and HTTP fallback strategies.

4.5 Multi-Agent Integration Tests

A key claim of this work is that Cortex integrates with *any* AI agent through its three interface layers. We validated this through two test suites: a gateway integration test exercising all protocol interfaces, and a plug test verifying zero-configuration agent discovery and injection.

4.5.1 Gateway Test Suite

Table 5 reports results from an automated test suite that exercises each integration interface end-to-end.

Table 4: Score progression across 5 fix iterations on the 100-site benchmark.

Iter.	Avg	≥80	Key Change
1	67.4	28	Baseline
2	73.9	68	URL edge inference
3	77.1	72	Unrendered node creation
4	82.3	74	Bidirectional edges
5	85.3	80	HTTP fallback mapping

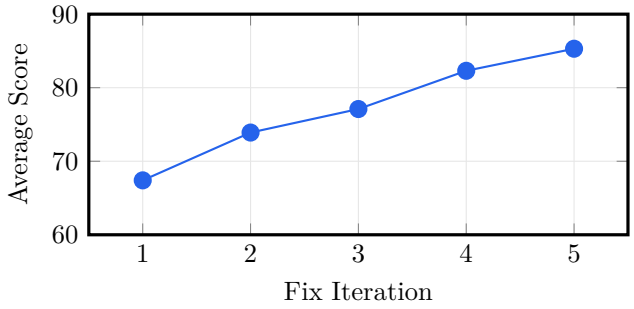

Figure 6: Average benchmark score across 5 fix iterations, from baseline (67.4) to final (85.3). Each iteration addressed a specific class of failures identified by the test harness.

Table 5: Gateway integration test results. Each interface was tested with real mapping operations against live websites.

Component	Score	Max
MCP server (7 tools)	30	30
REST API (9 endpoints)	27	30
Python client (5 operations)	22	25
Framework adapters	11	15
Total	**90**	**100**

Table 6: Agent auto-discovery (`cortex plug`) test suite. Tests verify safe, idempotent, and reversible config injection for 6 AI agent platforms.

Test Category	Score	Max
Agent discovery (6 agents)	15	15
Config injection	25	25
Idempotency (re-run safe)	15	15
Clean removal	25	25
Status reporting	10	10
Config safety (no data loss)	10	10
Total	**100**	**100**

The test performs real mapping operations (mapping `example.com` to 62 nodes and 142 edges), queries, cross-site comparison, and framework adapter calls through each interface.

The MCP server achieved a perfect score: all 7 tools registered correctly, the `@modelcontextprotocol/sdk` type-checked clean, and stdio transport handled request/response cycles without error. The REST API scored 27/30; the 3-point deduction was due to the PERCEIVE endpoint returning an expected `E_RENDERER` error in the test environment (no Chromium installed). The Python client successfully executed a cross-site comparison (`CompareResult(domains=['example.com', 'iana.org'], common_types=2)`), demonstrating the multi-site query pipeline through the native socket protocol. The CrewAI adapter passed; LangChain and OpenClaw were skipped because they were not installed in the test environment.

4.5.2 Agent Auto-Discovery (`cortex plug`)

Table 6 reports results from the `cortex plug` test suite. This command scans the machine for installed AI agents and injects Cortex MCP server configuration into each agent's config file—enabling zero-configuration setup.

The plug system detects 6 AI agent platforms by probing known configuration file paths: Claude Desktop, Claude Code, Cursor, Windsurf, Continue, and Cline. For each detected agent, it reads the existing MCP configuration (typically `claude_desktop_config.json` or equivalent), adds a `mcpServers.cortex` entry pointing to the Cortex MCP server, and writes the file back—preserving all other servers and settings. The test suite verified that injection is *idempotent* (running `cortex plug` twice does not duplicate entries), *reversible* (`cortex plug --remove` cleanly removes the Cortex entry while preserving others), and *safe* (existing configuration is never corrupted or lost).

4.5.3 End-to-End Agent Workflow

To illustrate the complete agent interaction pattern, we describe the workflow when Claude Desktop uses Cortex via MCP. After `cortex plug` injects the configuration, Claude sees 7 new tools. When a user asks "find me the cheapest noise-canceling headphones under $300," the following occurs:

1. Claude calls `cortex_map({domain: "amazon.com"})`. Cortex maps the site in ∼8 s via HTTP-first extraction, returning 418 nodes with 3,510 edges.
2. Claude calls `cortex_query({domain: "amazon.com", page_type: "product_detail", price_lt: 300})`. Cortex filters the in-memory graph in <1 ms, returning matching products with prices, ratings, and URLs.
3. Claude presents results to the user. Total time: ∼8 s. Pages visited by a browser: 0. LLM calls for navigation: 0.

For comparison, a page-by-page browser agent would need to: load the homepage (∼3 s), navigate to search (∼3 s), enter a query (∼2 s), apply filters (∼3 s), paginate through results (∼3 s × 3 pages), and click into individ-

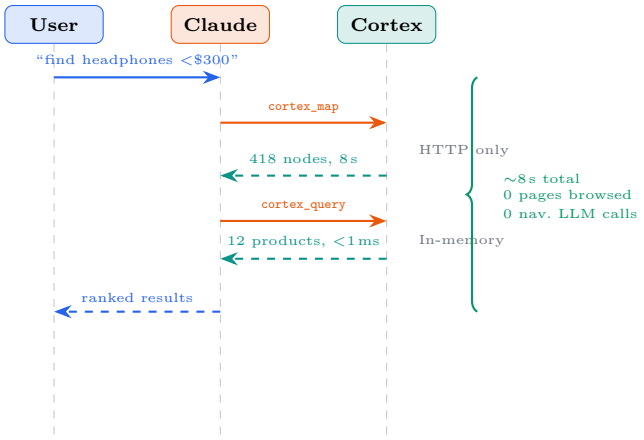

Figure 7: End-to-end agent workflow via MCP. Claude calls two Cortex tools (map + query) to answer a product comparison question. The entire interaction completes in ~8 seconds with zero browser page loads and zero LLM calls for navigation reasoning.

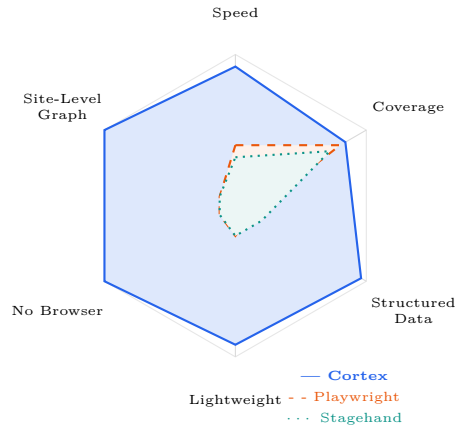

Figure 8: Radar chart comparing Cortex against browser-based tools across six dimensions. Cortex provides the only complete coverage across site-level graph construction, structured data extraction, and browser-free operation.

Table 7: System implementation metrics.

Metric	Value
Rust source lines	22,131
Rust source files	81
Test count (Rust)	219
Python client lines	2,252
Release binary size	17 MB
Mapping time (typical)	3–15 s
Query latency (in-memory)	<1 ms
Runtime memory (idle)	~20 MB
Runtime memory (5 cached maps)	~175 MB
Framework adapters	5
MCP tools exposed	7
REST API endpoints	9

ual products (~3 s × 5 products)—approximately 30 s of browser time with 12+ LLM reasoning calls.

Figure 7 illustrates this workflow as a sequence diagram.

4.6 System Metrics

Table 7 summarizes implementation metrics.

4.7 Comparison with Browser-Based Approaches

Table 8 compares Cortex with browser-based agent infrastructure across resource and capability dimensions. Cortex's HTTP-first architecture provides the primary advantage: for the majority of sites, no browser is needed, eliminating the 250–350 MB Chromium dependency and the 2–10 second per-page rendering latency.

5 Discussion

Enabled capabilities. The cartography model enables several operations impractical with per-page browsing. *Cross-site comparison* maps multiple retailers and queries all maps with the same feature dimensions, returning unified results sorted by price or rating—a task that would require opening dozens of browser tabs. *Shortest-path navigation* computes the optimal route between any two pages without trial-and-error exploration. *Offline planning* allows the agent to analyze site structure and plan a multi-step workflow entirely in memory before visiting a single page.

Failure modes. The primary failure mode is bot detection (10 of 20 below-80 sites). Sites employing protocol-level blocking (HTTP/2 errors, TLS fingerprinting) or JavaScript challenges (Cloudflare Turnstile) prevent both HTTP extraction and browser rendering. A secondary failure mode is client-rendered SPAs with no server-side rendering and no sitemap (6 sites): the HTML response contains a minimal shell with no discoverable links or structured data.

Limitations. Several limitations should be acknowledged. First, the mapping is a point-in-time snapshot; dynamic content (personalized recommendations, real-time inventory) requires periodic refresh. Second, authenticated content is only accessible after explicit credential provision via the AUTH protocol method. Third, the 128-dimensional feature vector schema is fixed and may not capture domain-specific attributes for all verticals. Fourth, the current implementation does not parallelize mapping across multiple sites; this would be a natural optimization for the `compare` operation.

Ethical considerations. Cortex respects `robots.txt` directives, obeys `Crawl-delay` settings, and rate-limits requests (default: 5 concurrent requests

Table 8: Comparison of Cortex with browser-based web agent infrastructure. Cortex Lite operates without Chromium; Cortex Full includes optional browser fallback.

Dimension	**Playwright**	**Puppeteer**	**Selenium**	**Browserbase**	**Cortex Lite**	**Cortex Full**
Package size	∼280 MB	∼300 MB	∼350 MB	∼0 (cloud)	**17 MB**	∼310 MB
Browser required	Always	Always	Always	Cloud	**Never**	Optional
Site-level graph	No	No	No	No	**Yes**	**Yes**
Latency per site	20–120 s	20–120 s	20–120 s	20–120 s	**3–15 s**	3–15 s
Pages visited	10–30	10–30	10–30	10–30	**0–2**	0–2
LLM calls (nav.)	10–30	10–30	10–30	10–30	**0**	0
Structured data	No	No	No	No	**Primary**	Primary
Idle memory	N/A	N/A	N/A	N/A	**∼20 MB**	∼20 MB

per domain, 100 ms minimum interval). No telemetry or data exfiltration occurs. The system is fully local and open-source (Apache-2.0).

Future work. Incremental map updates (delta refresh rather than full re-mapping) would reduce latency for frequently visited sites. Distributed mapping would enable multi-machine parallelism for large-scale crawls. Integration with browser-native WebMCP tools would allow Cortex to invoke site-declared capabilities directly. Finally, learning from agent behavior—which paths and queries are most useful—could inform adaptive sampling strategies.

6 Conclusion

We have presented Cortex, a web cartography engine that converts websites into navigable binary graph data structures for AI agents. By using layered HTTP-first extraction—sitemaps, JSON-LD, OpenGraph, CSS pattern matching, and API discovery—Cortex maps entire sites in seconds without a browser, achieving 93% structured data coverage across 100 production websites.

The implementation delivers strong results in a compact footprint: 22,131 lines of Rust, a 17 MB binary, 219 tests, and an average mapping score of 85.3/100 across 10 website categories. The agent integration layer—MCP server, REST API, and native client libraries—enables one-command setup for any AI agent framework.

Cortex represents a paradigm shift for web agents: from page-by-page perception to site-level cartography. The agent sees the whole board and computes the shortest path to its goal. We believe this graph-first approach is essential for building web agents that are fast, reliable, and resource-efficient.

References

[1] Anthropic. Model context protocol specification. `https://modelcontextprotocol.io/`, 2024. Accessed 2026.

[2] Anthropic. Web MCP: Model context protocol for the web. `https://modelcontextprotocol.io/`, 2025. Accessed 2026.

[3] Browserbase Inc. Browserbase: Headless browser infrastructure. `https://www.browserbase.com/`, 2024. Accessed 2026.

[4] Browserbase Inc. Stagehand: AI web browsing framework. `https://github.com/browserbase/stagehand`, 2024. Accessed 2026.

[5] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[6] Google Chrome Team. Puppeteer: Headless chrome Node.js API. `https://pptr.dev/`, 2017. Accessed 2026.

[7] LangChain Inc. LangChain: Building applications with LLMs. `https://www.langchain.com/`, 2022. Accessed 2026.

[8] Meta Platforms Inc. The open graph protocol. `https://ogp.me/`, 2010. Accessed 2026.

[9] Microsoft. Playwright: Reliable end-to-end testing for modern web apps. `https://playwright.dev/`, 2020. Accessed 2026.

[10] Microsoft Research. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. `https://github.com/microsoft/autogen`, 2023. Accessed 2026.

[11] OpenAPI Initiative. OpenAPI specification v3.1.0. `https://spec.openapis.org/oas/v3.1.0`, 2021. Accessed 2026.

[12] Joon Sung Park, Joseph C O'Brien, Carrie J Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–22, 2023.

[13] Schema.org Community. Schema.org: Vocabulary for structured data. `https://schema.org/`, 2011. Accessed 2026.

[14] Patrick Schönberger, Gerhard Gossen, Yannick Gerwert, and Florian Quinkert. A large-scale study of XML sitemaps. In *Proceedings of the ACM Web Conference*, 2023.

[15] Selenium Contributors. Selenium: Browser automation. `https://www.selenium.dev/`, 2004. Accessed 2026.

[16] W3C. JSON-LD 1.1: A JSON-based serialization for linked data. `https://www.w3.org/TR/json-ld11/`, 2014. W3C Recommendation, July 2020.