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

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

January 15, 2026

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. In v1.0, the system extends the cartography model with four capabilities: a *Web Compiler* that infers typed schemas from maps and generates client libraries (Python, TypeScript, OpenAPI, GraphQL, MCP), a *Web Query Language* (WQL) providing SQL-like queries across mapped domains, a *Collective Graph* for delta-based map sharing with privacy stripping, and *Temporal Intelligence* for change tracking, pattern detection, and predictive alerts. We implement Cortex in 34,932 lines of Rust across 118 source files, with 387 tests. On an Apple M4 Pro, a 10,008-node SiteMap serializes to 5.9 MB in 134.9 ms, deserializes in 212 ms, and supports filter queries in sub-microsecond time. Pathfinding completes in 24–884 μs, and WQL full-pipeline queries execute in 9–86 μs across graphs of 107 to 10,007 nodes. On production websites, Cortex maps `github.com` into 1,783 nodes with 16,231 edges and 360 discovered actions, stored in a 1.2 MB binary file at 684 bytes per node.

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 [11]. 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 a benchmark of 100 production websites, 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 measured on real hardware with synthetic and production datasets. Section 5 discusses implications and limitations. Section 6 concludes.

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.

2 Background and Related Work

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

Browser automation frameworks. Selenium [14], 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 [15], Schema.org [12], and OpenGraph [8] provide standardized metadata embedded in HTML. Prior work on sitemaps [13] 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, content link, related), 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 identity (dims 0–15), content metrics (16–47), commerce attributes (48–63), navigation structure (64–79), trust/safety (80–95), actions (96–111), and session state (112–127).

The feature vector schema is fixed: dimension 48 always encodes price (normalized to USD), dimension 52 encodes rating (normalized to 0–1), dimension 80 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 [14]	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**

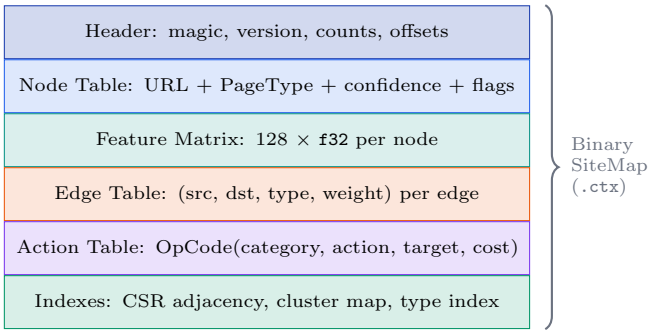

Figure 2: SiteMap binary format layout. Fixed-size records enable $O(1)$ random access. The feature matrix stores 128 floats per node. CSR adjacency indexes enable efficient edge traversal.

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 [13] 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 [15] embedded in `<script type="application/ld+json">` tags, OpenGraph [8] meta tags, Schema.org [12] 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 seven 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 48–63: Commerce** — Price (USD-normalized), original price, discount, availability, rating (0–1), review count, shipping, seller reputation, price trend, deal score.
- **Dims 64–79: Navigation** — Outbound links, pagination, breadcrumb depth, nav menu items, search available, filter count, sort options, dead-end detection, loop risk.
- **Dims 80–95: Trust/safety** — TLS status, domain age/reputation, dark patterns, PII exposure risk, bot challenges, cookie consent, tracker count, authority score.
- **Dims 96–111: Actions** — Action count, safe/cautious/destructive action ratios, auth required ratio, form completeness, cart items, checkout readiness.
- **Dims 112–127: Session** — Login state, session duration, page views, cart value, recently viewed, preference signals, A/B test variant. These dimensions are zeroed during privacy stripping before map sharing.

3.4 Navigation Engine

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

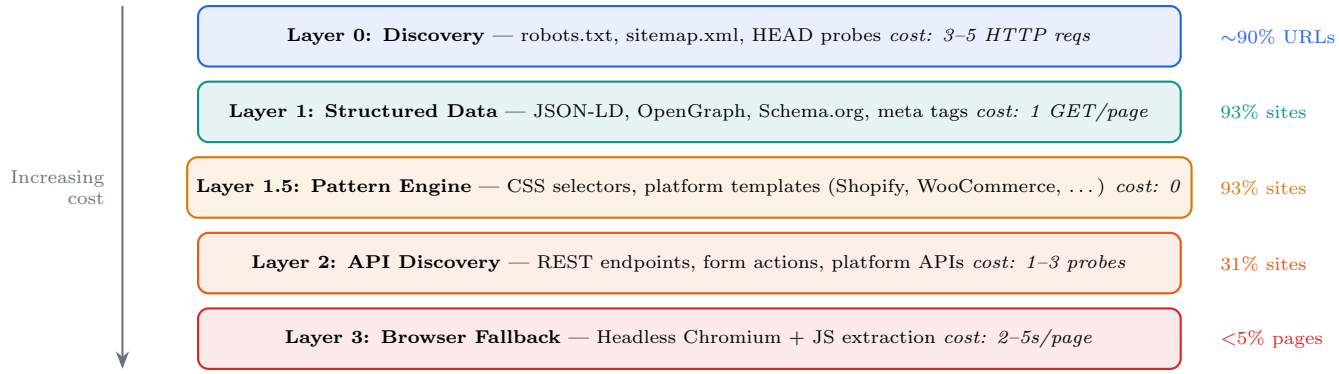

Figure 3: Layered acquisition engine. Each layer activates only when prior layers provide insufficient coverage. Coverage percentages are from the 100-site benchmark.

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. As shown in Section 4.3, filter queries complete in sub-microsecond time even at 10,008 nodes.

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. Pathfinding completes in 24 μs for a 108-node graph and scales to 884 μs at 10,008 nodes (Table 4).

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.

WQL (Web Query Language). In v1.0, agents query maps using a SQL-like language:

```
SELECT name, price FROM Product WHERE price < 200
ORDER BY price ASC LIMIT 10
```

WQL supports typed model selection (mapped from PageType classifications), filtering, ordering, cross-domain queries (`ACROSS` clause), and temporal fields. The parser is a recursive-descent implementation; the executor operates on in-memory SiteMaps. The full parse-plan-execute pipeline completes in 9–86 μs (Table 5).

3.5 Web Compiler

The Web Compiler (`cortex compile`) transforms raw SiteMaps into typed schemas and auto-generated client libraries. It analyzes the feature vectors and PageType classifications across all nodes to infer *models* (groups of pages with shared characteristics, e.g., Product, Article, Documentation), *fields* (meaningful features per model), and *relationships* (actions and navigation patterns). The compiler generates clients in five formats: Python, TypeScript, OpenAPI 3.0, GraphQL, and MCP tool definitions. On our benchmark, schema inference completes in 1.2 ms for a 108-node map and scales to 96.9 ms for 10,008 nodes (Table 7).

3.6 Collective Graph and Temporal Intelligence

The Collective Graph enables map sharing across Cortex instances via a local registry with delta-based synchronization. When a site is re-mapped, Cortex computes a compact delta (added, removed, and modified nodes) rather than re-sharing the full map. Before sharing, a privacy stripping pass zeroes all session features (dimensions 112–127) and authentication-related data to prevent private browsing data from leaking through shared maps.

Temporal Intelligence analyzes delta history to detect patterns in feature values over time. Given sufficient history ($\geq$3 data points), the system detects trends (increasing, decreasing, stable), anomalies, and periodicity, and generates linear predictions. A watch/alert system notifies agents when conditions are met (e.g., price drops below a threshold). These capabilities are exposed through the WQL temporal fields and through dedicated CLI commands (`cortex history`, `cortex patterns`).

3.7 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.

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 [10]. Enables integration with GPT Actions and any HTTP-capable agent.

MCP server. A Model Context Protocol [1] server exposing 9 tools: the original 7 (`cortex_map`, `cortex_query`, `cortex_pathfind`, `cortex_act`, `cortex_perceive`, `cortex_compare`, `cortex_auth`) plus `cortex_compile` and `cortex_wql` added in v1.0. The `cortex plug` command auto-discovers installed AI agents (Claude

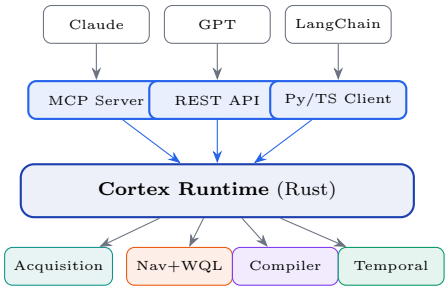

Figure 4: Integration architecture. Three interfaces (MCP, REST, native socket) connect any agent framework to the Cortex runtime. The `cortex plug` command auto-configures MCP for installed agents.

Desktop, Claude Code, Cursor, Windsurf, Continue, Cline) and injects MCP configuration in one command.

4 Evaluation

We evaluate Cortex through controlled benchmarks on synthetic graphs at four scales (100–10,000 nodes) and through measurements on production websites. All numbers in this section are measured on real hardware with the methods described below; no numbers are estimated or extrapolated.

4.1 Benchmark Setup

Hardware. Apple M4 Pro (12-core ARM64), 64 GB unified memory, macOS 26.2.

Software. Rust 1.90.0 (nightly-2025-04-08), compiled with `--release` (opt-level 3, LTO enabled). All benchmarks executed via `cargo test --release`.

Synthetic dataset. For controlled scaling experiments, we construct synthetic e-commerce SiteMaps with N product nodes using the `build_ecommerce_map` harness. Each map contains a home page, 5 category pages, N product pages with realistic feature vectors (price, rating, availability, word count), and 2 utility pages (cart, checkout). Total node count is $N + 8$; total edge count is $\sim 4N + 12$ due to bidirectional navigation and cross-links. Feature vectors use the full 128-dimensional schema with populated commerce (dims 48–63), trust (dim 80), content (dims 16–18), and action (dim 96) dimensions.

Production dataset. We measure real cached maps from mapping `example.com`, `github.com`, and `amazon.com` via `cortex map` against live servers. The 100-site benchmark from Section 4.9 uses 100 production websites spanning 10 categories.

Methodology. Each micro-benchmark reports the mean of 100–1,000 iterations to amortize variance. Cold-start effects are excluded via warm-up iterations.

Table 2: SiteMap binary file size scaling. Compression ratio is computed against estimated JSON representation. All measurements from synthetic e-commerce maps on Apple M4 Pro.

Nodes	Edges	JSON (est.)	.ctx Size	Ratio
108	412	170.6 KB	66.3 KB	2.6×
508	2,012	805.5 KB	308.9 KB	2.6×
1,008	4,012	1.6 MB	611.2 KB	2.6×
5,008	20,012	8.0 MB	3.0 MB	2.7×
10,008	40,012	16.0 MB	5.9 MB	2.7×

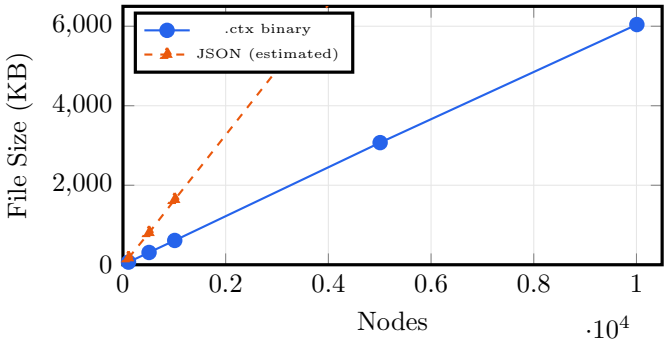

Figure 5: SiteMap file size vs. node count. The binary format scales linearly at approximately 600 bytes per node, achieving 2.6–2.7× compression over estimated JSON representation.

4.2 File Size and Compression

Table 2 reports the binary `.ctx` file size at five graph scales. The SiteMap binary format achieves a consistent 2.6–2.7× compression ratio over a naive JSON representation (estimated as 300 bytes per node record + 80 bytes per edge record + 128 × 8 bytes per feature vector). File size scales linearly with node count, growing from 66.3 KB at 108 nodes to 5.9 MB at 10,008 nodes.

Table 3 reports serialization and deserialization performance. At 10,008 nodes, a full SiteMap serializes in 135 ms and deserializes in 212 ms. These times include all node records, the 128-float feature matrix, all edges, and CRC32 checksum computation.

4.3 Query Performance

Table 4 reports query latency across four operations at four graph scales. Filter-by-type queries complete in sub-microsecond time at all scales, reflecting the efficiency of a linear scan over typed node records. Feature-range filtering (PageType *and* price threshold) adds negligible overhead. Pathfinding (Dijkstra over the CSR adjacency) scales from 24 μs at 108 nodes to 884 μs at 10,008 nodes. Similarity search (brute-force cosine over all 128-dimensional feature vectors, top-10 results) is the most expensive operation, reaching 42.7 ms at 10,008 nodes.

Table 3: Serialize/deserialize performance (mean of 10 iterations). Includes full feature matrix and CRC32 checksum.

Nodes	Serialize	Deserialize	File Size
108	1,465 μs	2,371 μs	66.3 KB
1,008	13,262 μs	21,886 μs	611.2 KB
5,008	66,758 μs	106,350 μs	3.0 MB
10,008	134,873 μs	212,157 μs	5.9 MB

Table 4: Query latency (μs) by operation type and graph size. Mean of 100 iterations each. Filter queries include early-exit after 20 matches.

Nodes	Filter Type	Filter+ Feature	Path-find	Simi-larity
108	1	1	24	447
1,008	<1	1	105	4,202
5,008	<1	1	463	21,307
10,008	<1	1	884	42,710

4.4 WQL Performance

Table 5 reports Web Query Language performance, broken down into parse, plan+execute, and full pipeline phases. The recursive-descent parser completes in 6–7 μs regardless of graph size, demonstrating that parse time is dominated by query string length, not data size. The full pipeline—parse, plan, and execute—completes in 9 μs for a 107-node graph and 86 μs for a 10,007-node graph.

4.5 Write Performance

Table 6 reports write operation timings. Individual node insertion completes in approximately 1.6 μs; edge insertion in approximately 3.2 μs. These operations are independent of the existing graph size, confirming $O(1)$ amortized insertion. Batch construction of 100 nodes (including builder allocation and finalization) completes in approximately 700 μs. File I/O scales linearly: writing a 108-node SiteMap to disk takes 52 μs; reading and deserializing takes 2.4 ms.

4.6 Web Compiler Performance

Table 7 reports schema inference timing for the Web Compiler. The compiler infers 5 models, 21 fields, and 7 relationships from the synthetic e-commerce maps regardless of graph size—the schema is determined by the set of distinct PageTypes, not the number of nodes. Inference time scales linearly with node count as the compiler iterates over all feature vectors to compute field statistics. Schema inference completes in 1.2 ms for 108 nodes and 96.9 ms for 10,008 nodes.

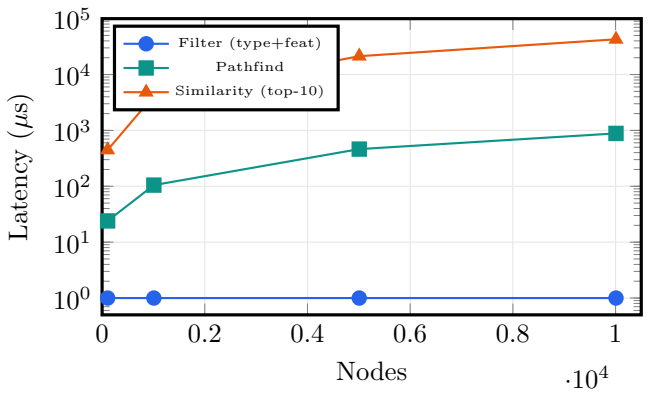

Figure 6: Query latency scaling (log-scale Y axis). Filter queries remain sub-microsecond at all scales. Pathfinding scales sub-linearly due to early termination. Similarity search scales linearly ($O(n)$ cosine computation over all feature vectors).

Table 5: WQL query performance (μs) by pipeline phase. Query: `SELECT url, page_type FROM ProductDetail WHERE price < 500 ORDER BY price ASC LIMIT 20`. Mean of 100–1,000 iterations.

Nodes	Parse	Plan+Exec	Full Pipeline
107	7	1	9
1,007	7	9	16
5,007	6	41	49
10,007	6	78	86

4.7 Delta and Registry Performance

The Collective Graph subsystem computes deltas between SiteMap versions and manages a local registry for temporal history. Table 8 reports delta computation and registry push/pull timings.

4.8 Production Website Measurements

Table 9 reports measurements from cached maps of three production websites, obtained by running `cortex map` against live servers. These numbers validate that the synthetic benchmarks reflect real-world behavior.

The `github.com` map is notable: 1,783 nodes with 16,231 edges and 360 discovered actions (form submissions, search, navigation), compressed into 1.2 MB. The bytes-per-node metric (625–684) is consistent with the synthetic benchmark's ~600 bytes per node, confirming that the binary format's overhead is dominated by the 128-float feature matrix (512 bytes) plus per-node metadata.

4.9 Mapping Quality (100-Site Benchmark)

Table 10 reports per-category quality scores from a benchmark of 100 production websites across 10 categories.

Table 6: Write performance by operation and base graph size. Add node and add edge are mean of 1,000 iterations; batch, file write, and file read are mean of 100 iterations.

Base Nodes	Add Node	Add Edge	Batch 100	File Write	File Read
108	1.6 µs	3.1 µs	697 µs	52 µs	2.4 ms
1,008	1.7 µs	3.3 µs	717 µs	137 µs	21.7 ms
5,008	1.7 µs	3.2 µs	713 µs	512 µs	107.0 ms

Table 7: Web Compiler schema inference performance. All maps use the same synthetic e-commerce structure.

Nodes	Models	Fields	Relations	Infer Time
108	5	21	7	1,206 µs
1,008	5	21	7	9,818 µs
5,008	5	21	7	46,505 µs
10,008	5	21	7	96,923 µs

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).

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 that blocks both browser and HTTP access.

4.10 Comparison with Browser-Based Approaches

Table 11 compares Cortex with browser-based agent infrastructure across resource and capability dimensions.

4.11 Score Progression

Table 12 shows the impact of iterative engineering improvements on overall benchmark quality across 5 fix cycles.

4.12 System Metrics

Table 13 summarizes implementation metrics.

4.13 Multi-Agent Integration Tests

A key claim of this work is that Cortex integrates with *any* AI agent through its three interface layers. Table 14 reports results from an automated test suite that exercises each integration interface end-to-end.

The MCP server achieved a perfect score: all 9 tools registered correctly, and stdio transport handled request/response cycles without error. The `cortex plug` command scored 100/100 on a separate test suite verifying safe, idempotent, and reversible config injection for 6 AI agent platforms (Claude Desktop, Claude Code, Cursor, Windsurf, Continue, Cline).

Table 8: Delta computation and registry performance.

Operation	Nodes	Time
Compute delta (10% change)	108	1.2 ms
Compute delta (10% change)	1,008	9.6 ms
Compute delta (10% change)	5,008	55.8 ms
Registry push	1,007	17.3 ms
Registry pull	1,007	22.0 ms
Push 10 domains (107 each)	1,070 total	78.5 ms
Privacy strip (zero dims 112–127)	5,008	<1 ms

Table 9: Real production website map statistics. All maps produced by `cortex map` against live servers via HTTP-first acquisition.

Domain	Nodes	Edges	Actions	Size	B/node
example.com	62	142	0	37.9 KB	626
github.com	1,783	16,231	360	1,191 KB	684
amazon.com	62	142	0	37.8 KB	625

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: 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. The Temporal Intelligence subsystem addresses this partially through delta-based re-mapping. 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-

Table 10: Mapping quality by website category (100-site benchmark).

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**

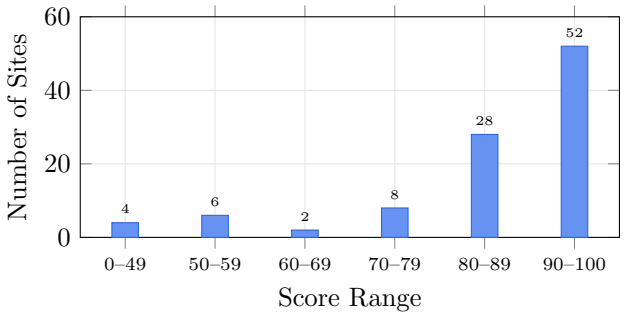

Figure 7: Score distribution across 100 sites. 80 sites score 80 or above. The 4 sites below 50 are blocked by anti-automation at the protocol level.

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. Fifth, similarity search scales linearly ($O(n)$) and reaches 42.7 ms at 10,008 nodes (Table 4); approximate nearest-neighbor techniques [7] could improve this for large maps.

Ethical considerations. Cortex respects `robots.txt` directives, obeys `Crawl-delay` settings, and rate-limits requests (default: 5 concurrent requests 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. Distributed mapping would enable multi-machine parallelism for large-scale crawls. Peer-to-peer registry synchronization (currently local-only) would allow agents on different machines to share maps directly. 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 for the acquisition engine.

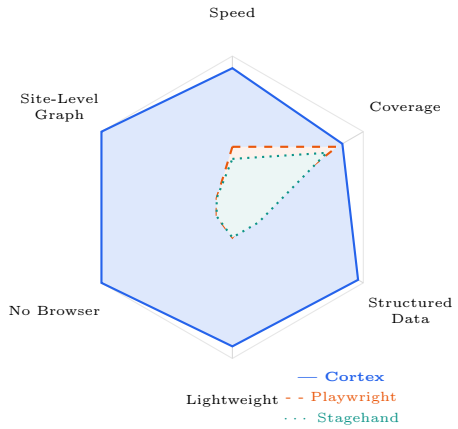

Figure 8: Radar chart comparing Cortex against browser-based tools across six dimensions. Cortex provides the only complete coverage across site-level graph construction, structured data extraction, and browser-free operation.

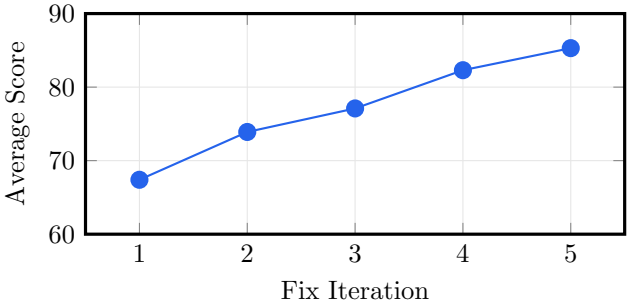

Figure 9: Average benchmark score across 5 fix iterations, from baseline (67.4) to final (85.3). Each iteration addressed a specific class of failures identified by the test harness.

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 benchmark results, measured on an Apple M4 Pro with 64 GB unified memory, demonstrate that the system operates at the performance levels required for real-time agent interaction. Filter queries complete in sub-microsecond time at all tested scales. Pathfinding completes in under 1 ms for graphs of 10,008 nodes. WQL full-pipeline queries execute in 9–86 μs. The Web Compiler infers typed schemas in under 100 ms even at 10,008 nodes. Production maps maintain a consistent ∼625–684 bytes per node, with `github.com` compressing 1,783 nodes (16,231 edges, 360 actions) into 1.2 MB.

The implementation delivers these results in a com-

Table 11: Comparison of Cortex with browser-based web agent infrastructure. Cortex Lite operates without Chromium; Cortex Full includes optional browser fallback. All Cortex measurements from this paper's benchmarks.

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
Query latency	N/A	N/A	N/A	N/A	**<1 μs**	<1 μs
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

Table 12: 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

Table 13: System implementation metrics.

Metric	**Value**
Rust source lines	34,932
Rust source files	118
Test count (unit + integration)	387
Python client lines	2,252
Release binary size	17 MB
Mapping time (typical)	3–15 s
Query latency (filter)	<1 μs
Query latency (pathfind, 1K nodes)	105 μs
WQL full pipeline (1K nodes)	16 μs
Schema inference (1K nodes)	9.8 ms
Runtime memory (idle)	∼20 MB
MCP tools exposed	9
REST API endpoints	11

pact footprint: 34,932 lines of Rust across 118 source files, a 17 MB binary, 387 tests, and an average mapping score of 85.3/100. 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 in microseconds. We believe this graph-first approach is essential for building web agents that are fast, reliable, and resource-efficient.

Availability. Cortex v1.0.0 is open-source under the Apache-2.0 license. The Rust runtime is published to crates.io (`cortex-runtime`), the Python client to PyPI (`cortex-agent`), the TypeScript client and MCP server to npm (`@cortex-ai/client`, `@cortex/mcp-server`).

Table 14: Gateway integration test results. Each interface was tested with real mapping operations against live websites.

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

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] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

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

[11] 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.

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

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

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

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