The Semantic Object Model: A Token-Efficient Web Representation for AI Agents

David Hurley
Plasmate Labs

Abstract

We present the Semantic Object Model (SOM), a structured representation format for web page content designed for consumption by large language models and AI agents. Current approaches to web content extraction rely on headless browsers that produce Document Object Model (DOM) trees containing 50,000+ nodes per page, the vast majority of which encode presentational rather than semantic information. SOM compresses web pages into structured JSON documents that preserve content hierarchy, interactive elements, and semantic relationships while eliminating presentational markup. We evaluate SOM against 98 real-world websites spanning news, e-commerce, documentation, social platforms, and government sites, demonstrating 16.6x average token compression (94% savings) compared to raw HTML tokenization (cl100k_base). We further show that SOM-based content extraction yields an approximately 94% reduction in LLM input token costs for web analysis workloads at scale. The SOM specification is open (Apache 2.0) and is being incubated as a community standard through the W3C Web Content Browser for AI Agents Community Group.

1 Introduction

The interaction between AI agents and web content represents one of the most significant scaling challenges in modern AI systems. As language model-based agents increasingly perform web research, data extraction, and automated browsing tasks, the cost and efficiency of web content consumption becomes a critical infrastructure concern.

Current agent-web interaction follows a pattern established by human browser automation:

1. Launch a headless browser instance (typically Chromium, ~300MB memory)
2. Navigate to a target URL
3. Wait for page rendering (CSS layout, JavaScript execution)
4. Extract the Document Object Model (DOM)
5. Serialize the DOM as text for LLM consumption

This pipeline was designed for human browser testing and carries substantial overhead for machine consumption. The rendering step (step 3) produces pixel-level layout information that AI agents discard immediately. The DOM (step 4) contains an average of 50,000+ nodes per page, of which approximately 90% encode presentational structure (CSS classes, layout containers, script elements) rather than semantic content.

We propose the Semantic Object Model (SOM) as an alternative representation that eliminates this overhead. SOM is a JSON-based format that structures web content into semantic regions

(navigation, content, forms, tables) containing typed elements (headings, paragraphs, links, inputs) with their associated text and attributes.

1.1 Contributions

- A formal specification for SOM, a token-efficient web page representation format
- An open-source implementation (Plasmate) that compiles HTML to SOM using html5ever and V8
- A comprehensive benchmark across 100 real-world websites demonstrating 16.6x average token compression (94% savings), with nightly CI-driven coverage scorecards
- Cost analysis showing approximately 94% reduction in LLM input token costs for web analysis tasks
- A comparison with existing approaches (raw HTML, Markdown conversion, accessibility trees)

2 Related Work

2.1 Web Content Extraction

Traditional approaches include CSS selector-based extraction (Beautiful Soup [2], Cheerio [3]), XPath queries, and readability algorithms (Mozilla Readability [1]). These methods require per-site configuration or produce unstructured text that loses document hierarchy.

2.2 Browser Automation for AI

Playwright [4], Puppeteer [5], and Selenium [6] provide programmatic browser control. Recent work on Browser Use [7], LaVague [8], and Stagehand [9] adds LLM-guided interaction layers. All rely on Chromium's rendering engine.

2.3 Lightweight Browsers

Lightpanda [10] (Zig-based, AGPL) and Servo [11] (Rust-based, Mozilla) represent alternative browser engines. Both target DOM-level output rather than semantic compression.

2.4 Token Optimization

Existing token optimization for web content includes HTML minification, DOM pruning, and Markdown conversion (Jina Reader [13], Firecrawl [12]). These achieve 2–5x compression. SOM achieves 10–16x by operating at the semantic rather than syntactic level.

3 The Semantic Object Model

3.1 Format Specification

A SOM document is a JSON object with the following top-level structure:

```
{
  "som_version": "1.0",
  "url": "string",
  "title": "string",
  "lang": "string",
  "regions": [Region]
}
```

3.1.1 Regions

Regions represent semantic sections of a page:

- `navigation` – site navigation elements
- `content` – main page content
- `form` – interactive form sections
- `complementary` – sidebars, related content
- `footer` – page footer information

3.1.2 Elements

Each region contains typed elements:

- `heading` (with level attribute 1–6)
- `paragraph`
- `link` (with href attribute)
- `image` (with src, alt attributes)
- `list` (with items)
- `table` (with headers and rows)
- `input` (with type, label, name attributes)

3.2 Compilation Pipeline

SOM compilation proceeds in four stages:

1. **HTML Parsing** – Using html5ever (Rust), producing a DOM tree
2. **JavaScript Execution** – Using V8, executing page scripts to resolve dynamic content
3. **Semantic Extraction** – Classifying DOM subtrees into regions and elements based on HTML5 sectioning, ARIA roles, and heuristic analysis
4. **SOM Serialization** – Producing the final JSON document

3.3 Compression Mechanisms

SOM achieves compression through:

- **Structural elimination** – CSS classes, style attributes, data attributes, and layout containers are removed
- **Deduplication** – Repeated navigation and footer elements across page sections are merged
- **Type inference** – DOM nodes are classified by semantic role rather than preserved as nested element trees
- **Text extraction** – Only visible text content is preserved; script content, hidden elements, and metadata are excluded

4 Evaluation

4.1 Benchmark Dataset

We evaluate against two overlapping datasets:

- A 49-URL cost analysis benchmark (`benchmarks/run-cost-analysis.sh`)
- A 100-URL nightly coverage scorecard (`bench/top100.txt`)

Both span news, e-commerce, documentation, social forums, SaaS, government, and reference sites. The canonical URL lists and scripts are published in the Plasmate source repository.

4.2 Token Counting Methodology

All token counts use the cl100k_base encoding (tiktoken [17]) used by GPT-4, GPT-4o, and Claude 3.x models. For each URL:

- **HTML tokens**: Raw HTTP response body tokenized directly
- **SOM tokens**: Plasmate fetch output (JSON) tokenized

4.3 Results

We evaluate SOM compression in two complementary ways:

Cost Analysis Benchmark (49 sites). We tested 49 real-world websites and measured the token reduction from raw HTML to Plasmate SOM output. Headline results:

- 16.6x overall (mean) compression
- 10.5x median compression
- 94% token cost savings
- Range: 0.7x (example.com, a trivially small page) to 116.9x (cloud.google.com)

The top 5 compression results were: cloud.google.com (116.9x), linear.app (105.3x), reddit.com (103.8x), figma.com (63.6x), and tailwindcss.com (53.7x). Two URLs (w3.org, dev.to) failed during the benchmark and were excluded.

Nightly Coverage Scorecard (100 sites). A curated set of 100 agent-relevant websites is tested nightly via a scheduled GitHub Action. Results are published automatically in two modes:

- HTML-only mode (JS disabled): `plasmate coverage -urls bench/top100.txt`
- JS-enabled mode (V8 active): same command with `-js`

These scorecards report per-URL HTTP status, compression ratio, HTML bytes, SOM bytes, element count, interactive element count, and parse time. They are updated every night so regressions are caught immediately.

All results are published at:

- Cost analysis: `https://docs.plasmate.app/benchmark-cost`
- HTML coverage: `https://docs.plasmate.app/coverage`
- JS coverage: `https://docs.plasmate.app/coverage-js`

4.4 Cost Analysis

At published LLM input pricing, the 16.6x compression translates to approximately 94% savings on input token costs:

Model	HTML Cost (49 URLs, 1K loads)	SOM Cost	Savings
GPT-4 ($10/M tokens)	$50,397	$3,042	$47,355 (94%)
GPT-4o ($2.50/M tokens)	$12,599	$761	$11,839 (94%)
Claude Sonnet ($3/M tokens)	$15,119	$913	$14,207 (94%)

At scale (1M pages/month at the average page size from the benchmark):

Model	HTML/month	SOM/month	Monthly Savings
GPT-4	$1,028	$62	$966
GPT-4o	$257	$16	$241
Claude Sonnet	$308	$19	$290

4.5 Information Preservation (Q/A Study)

We will evaluate information preservation via a question answering study comparing answers produced using a Chrome DOM plus Readability extraction baseline versus SOM context on the same pages. This experiment is implemented using the somordom.com comparison infrastructure and will be published with scripts and raw outputs for reproducibility.

Results: in progress.

5 Implementation

Plasmate is implemented in Rust (approximately 25,000 lines) using:

- html5ever [15] for HTML parsing
- rusty_v8 [16] for JavaScript execution
- tokio for async I/O
- serde for JSON serialization

The binary is approximately 43 to 46MB depending on platform and build, and requires no external runtime dependencies. It is available on crates.io (Rust), PyPI (Python), npm (Node.js), and Docker under the Apache 2.0 license.

Performance characteristics (published in docs and reproduced via the benchmark suite):

- Throughput: approximately 250 pages/sec on a 100-page local benchmark (231KB average)
- Memory: approximately 30MB RSS baseline per page

6 Standards and Community

The SOM specification is being incubated through the W3C Web Content Browser for AI Agents Community Group [18] (established March 2026). The group is developing two complementary specifications:

1. **SOM v1.0** – The semantic representation format described in this paper
2. **Agent Web Protocol (AWP)** – A communication protocol for agent-web interaction

Both specifications are published under permissive licenses and intended for broad adoption.

7 Limitations and Future Work

- **JavaScript-heavy SPAs**: Single-page applications that render entirely in JavaScript may produce incomplete SOM output depending on V8 execution timing
- **Visual layout semantics**: Some content meaning is conveyed through visual positioning (e.g., sidebar emphasis) that SOM does not capture
- **Dynamic content**: Pages with infinite scroll, lazy loading, or user-triggered content require session-based interaction
- **Multimedia**: Audio and video content is referenced but not transcribed

Future work includes: SOM caching infrastructure for reduced redundant crawling, differential SOM updates for change detection, and formal verification of information preservation guarantees.

8 Conclusion

The Semantic Object Model demonstrates that web content can be represented in a form that is 16.6x more token-efficient for AI consumption. As AI agents increasingly interact with web content at scale, the gap between human-oriented rendering and machine-oriented comprehension represents a significant infrastructure cost. SOM and the Plasmate engine offer an open, standards-based approach to bridging this gap. An ongoing Q/A study (Chrome extract baseline versus SOM) evaluates information preservation directly.

Reproducibility

All benchmarks are fully reproducible. The benchmark scripts, URL lists, and nightly CI configuration are published in the Plasmate source repository under `benchmarks/`.

- Source code: `https://github.com/plasmate-labs/plasmate`
- Cost analysis (49 sites): `https://docs.plasmate.app/benchmark-cost`
- Nightly coverage scorecard (100 sites, HTML-only): `https://docs.plasmate.app/coverage`
- Nightly coverage scorecard (100 sites, JS-enabled): `https://docs.plasmate.app/coverage-js`
- SOM specification: `https://docs.plasmate.app/som-spec`

To reproduce locally:

```
cargo install plasmate
git clone https://github.com/plasmate-labs/plasmate
cd plasmate
./benchmarks/run-cost-analysis.sh
```

References

[1] Mozilla Readability. `https://github.com/mozilla/readability`

[2] Beautiful Soup. `https://www.crummy.com/software/BeautifulSoup/`

[3] Cheerio. `https://cheerio.js.org/`

[4] Playwright. `https://playwright.dev/`

[5] Puppeteer. `https://pptr.dev/`

[6] Selenium. `https://www.selenium.dev/`

[7] Browser Use. `https://github.com/browser-use/browser-use`

[8] LaVague. `https://github.com/lavague-ai/LaVague`

[9] Stagehand. `https://github.com/browserbase/stagehand`

[10] Lightpanda. `https://lightpanda.io/`

[11] Servo. https://servo.org/

[12] Firecrawl. https://firecrawl.dev/

[13] Jina Reader. https://jina.ai/reader/

[14] Crawl4AI. https://github.com/unclecode/crawl4ai

[15] html5ever. https://github.com/nickel-org/html5ever

[16] rusty_v8. https://github.com/nickel-org/rusty_v8

[17] tiktoken. https://github.com/openai/tiktoken

[18] W3C Web Content Browser for AI Agents Community Group. https://www.w3.org/community/web-content-browser-ai/