Agent Web Protocol: A Purpose-Built Communication Protocol for AI Agent–Web Interaction

David Hurley
Plasmate Labs

March 2026

Abstract

We present the Agent Web Protocol (AWP), a purpose-built communication protocol for bidirectional interaction between AI agents and web browsers. The Chrome DevTools Protocol (CDP), the current de facto standard for programmatic browser control, was designed as a debugging interface for human developers. Its 300+ methods, pixel-oriented output model, and single-user session architecture make it fundamentally mismatched for AI agent workloads that require semantic understanding, token efficiency, and massive concurrency. AWP addresses these deficiencies through seven core design decisions: (1) an intent-based interaction model that replaces coordinate-based clicking with semantic element targeting; (2) integration with the Semantic Object Model (SOM) for token-efficient page observation; (3) a minimal method surface of 7 methods in the initial release, compared to CDP's 300+; (4) first-class session and state management with persistence and concurrent session support; (5) cursor-based incremental observation via JSON Patch mutations; (6) extensibility through WebAssembly skills for domain-specific automation; and (7) a layered security model with fine-grained permission scopes. We describe the protocol architecture, present the complete v0.1 method set with wire-level message examples, and provide a detailed comparative analysis against CDP across twelve dimensions. AWP is specified over WebSocket with MessagePack/JSON encoding and is implemented as part of the Plasmate agentic browser engine. The protocol specification is open (Apache 2.0) and is being incubated through the W3C Web Content Browser for AI Agents Community Group.

1 Introduction

The interaction between AI agents and the World Wide Web is mediated almost exclusively by a single protocol: the Chrome DevTools Protocol (CDP) [1]. Originally designed in 2004 as a debugging interface for the WebKit Web Inspector and later adopted by Chromium, CDP enables programmatic browser control through a rich set of domain-specific commands covering DOM inspection, network interception, JavaScript debugging, CSS manipulation, and rendering control.

CDP was never designed for AI agents. It was designed for human developers who need to inspect and manipulate running web applications through graphical developer tools. This mismatch manifests across several dimensions:

Surface area. CDP exposes over 300 methods organized into 40+ domains [1, 2]. An AI agent performing a simple web search must navigate methods drawn from at minimum the Page, DOM, Runtime, Input, and Network domains. The protocol's surface area imposes a substantial cognitive and token burden on agents that must reason about which methods to call.

Pixel orientation. CDP's primary interaction model is coordinate-based. Clicking an element requires the agent to determine the element's pixel coordinates via `DOM.getBoxModel` or `DOM.getContentQuads`, then dispatch synthetic mouse events via `Input.dispatchMouseEvent` with precise x/y values [2]. This forces agents to reason about visual layout, a task for which they have no native capability.

Output format. CDP returns raw DOM trees containing 50,000+ nodes per page [3]. The vast majority of these nodes encode presentational structure (CSS classes, layout containers, SVG paths, script elements) that is irrelevant to an agent's task. Serializing this output for LLM consumption wastes 90–95% of input tokens on noise [3, 4].

Session model. CDP was designed for single-user debugging sessions. Each WebSocket connection controls a single browser context. Scaling to thousands of concurrent agent sessions requires thousands of separate browser instances, each consuming 300–500MB of memory [5].

Stability guarantees. The tip-of-tree CDP protocol changes frequently and provides no backward compatibility guarantees [1]. The stable 1.3 release captures only a subset of capabilities. This makes CDP a moving target for automation frameworks.

These deficiencies are not incidental; they are structural consequences of CDP's design goals. A protocol designed for human developers inspecting rendered web pages will always be suboptimal for AI agents that need semantic understanding of web content.

This paper presents the Agent Web Protocol (AWP), a purpose-built protocol for AI agent–web interaction. AWP is designed around four premises:

1. Agents need semantics, not pixels. The primary observation output should be a structured, token-efficient representation of page content, not a DOM tree or screenshot.

2. Agents need intent, not coordinates. Interaction should target elements by semantic identity (role, label, stable ID), not by pixel position.

3. Agents need efficiency, not completeness. A small, well-designed method surface serves agent workloads better than an exhaustive debugging API.

4. Agents need concurrency, not isolation. The protocol must support thousands of concurrent sessions on a single connection.

AWP is implemented as part of the Plasmate agentic browser engine [6] and is specified over WebSocket with support for both JSON and MessagePack encoding. The v0.1 release implements 7 core methods sufficient for navigation, observation, interaction, and data extraction. The protocol specification is open (Apache 2.0) and is being developed through the W3C Web Content Browser for AI Agents Community Group [7].

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 states design goals and non-goals. Section 4 describes the protocol architecture. Section 5 provides a detailed treatment of each core method with wire-level examples. Section 6 explains the intent-based interaction model. Section 7 describes SOM integration. Section 8 covers session and state management. Section 9 presents the WebAssembly skill system. Section 10 describes the security model. Section 11 discusses CDP compatibility. Section 12 provides a systematic comparison. Section 13 reports implementation status. Section 14 concludes.

2 Related Work

2.1 Chrome DevTools Protocol (CDP)

CDP [1] is a JSON-RPC-like protocol organized into domains (DOM, Page, Network, Runtime, Input, CSS, Debugger, etc.). Communication occurs over WebSocket, with commands and events serialized as JSON. CDP powers the major browser automation libraries: Puppeteer [8] (Google, JavaScript), Playwright [9] (Microsoft, multi-language), and the ChromeDriver component of Selenium [10].

CDP's strengths are its completeness and maturity. It provides fine-grained control over every aspect of browser behavior, from network request interception to JavaScript heap profiling. However, this completeness comes at a cost: the protocol surface is enormous, undocumented methods outnumber documented ones, and the tip-of-tree version has no stability guarantees [1].

For AI agent use cases, CDP's primary limitation is that it was designed as an observation and debugging protocol, not an interaction protocol. Actions are expressed as low-level browser events (mouse clicks at coordinates, keyboard events with key codes), and observation returns raw DOM structures or pixel screenshots. No semantic abstraction layer exists within the protocol itself.

2.2 WebDriver and WebDriver BiDi

The W3C WebDriver specification [11] defines a REST API for browser automation, originally developed for the Selenium project. WebDriver operates at a higher abstraction level than CDP, with commands like "click element" and "send keys" that accept element references rather than coordinates. However, WebDriver's element references are opaque handles returned by CSS or XPath queries, not semantic identifiers.

WebDriver BiDi [12] is the successor protocol that adds bidirectional communication via WebSocket, enabling event subscription and streaming. While BiDi moves closer to the needs of agent workloads, it retains WebDriver's DOM-centric observation model and does not provide semantic page representations.

2.3 Selenium Wire and Network-Level Approaches

Selenium Wire [13] extends Selenium with network traffic capture, enabling inspection of HTTP requests and responses during automation. This approach provides data that agents can use for understanding page behavior but does not address the fundamental observation problem: agents still receive raw DOM or pixel output.

2.4 Accessibility APIs

Operating system accessibility APIs (MSAA/UIA on Windows, AT-SPI on Linux, NSAccessibility on macOS) provide semantic representations of UI elements including roles, labels, states, and actions [14]. The accessibility tree is conceptually closer to what agents need than the DOM tree. Recent agent frameworks including Browser Use [15] and Stagehand [16] have adopted accessibility tree extraction as part of their observation pipeline.

However, accessibility APIs have significant limitations for agent use: they are operating system specific, require a running renderer to populate, may be incomplete for web content that lacks proper ARIA annotations, and are not designed for remote or headless operation. The Semantic Object Model (SOM) [3] used by AWP draws inspiration from accessibility trees while addressing these limitations through direct HTML analysis and standardized output.

2.5 Agent–Browser Frameworks

A growing ecosystem of frameworks mediates between AI agents and web browsers: Browser Use [15] wraps Playwright with LLM-guided interaction; LaVague [17] generates Selenium code from natural language; Stagehand [16] provides "act" and "observe" abstractions over Playwright; and Crawl4AI [18] focuses on LLM-friendly crawling output.

All of these frameworks operate as layers above CDP/Playwright/Selenium, inheriting the underlying protocol's limitations. They add value through LLM integration and higher-level APIs but cannot overcome the fundamental mismatch between CDP's design and agent requirements. AWP proposes to address this mismatch at the protocol level.

2.6 Alternative Browser Engines

Lightpanda [19] (Zig-based, AGPL) and Servo [20] (Rust-based, Mozilla) represent alternative browser engines that could serve as AWP backends. Both target DOM-level output rather than semantic compression, but their lightweight architectures align with the resource efficiency goals of agent workloads.

3 Design Goals and Non-Goals

AWP's design goals and non-goals are drawn directly from the protocol specification [21] and reflect deliberate choices about what the protocol should and should not attempt.

3.1 Goals

Goal 1: Intent-first interaction. Agents issue semantic actions, not pixel coordinates. The protocol's action model accepts element references, semantic queries, and fallback chains rather than (x, y) positions.

Goal 2: Token efficiency. The primary observation output is the Semantic Object Model (SOM) [3], achieving 16.6× average token compression compared to raw HTML. AWP is designed so that agents can operate effectively without ever requesting a screenshot.

Goal 3: Deterministic references. Elements are addressed by stable IDs derived from semantic identity (origin, role, accessible name, DOM path). These IDs survive page refreshes and minor layout changes.

Goal 4: Massive concurrency. The protocol supports multiplexing thousands of sessions over a single WebSocket connection, with configurable per-session resource limits.

Goal 5: Robustness. Sessions survive reconnects, and operations are idempotent when practical. The protocol supports idempotency keys and trace IDs for reliable distributed operation.

Goal 6: Extensibility. Site-specific logic is implemented as WebAssembly skills in a sandboxed environment, not baked into the engine.

Goal 7: Observability. First-class telemetry for latency, errors, retries, costs, and provenance is built into the protocol through trace and span identifiers.

Goal 8: Security. Clear trust boundaries and a permission model govern high-risk operations.

3.2 Non-Goals

AWP does not define a UI, rendering pipeline, or pixel output requirements. AWP does not define scraping legality or usage policy. AWP does not mandate a specific SOM schema beyond normative minimums. AWP does not prescribe a particular LLM, agent framework, or planning strategy.

4 Protocol Architecture

4.1 Transport

AWP is specified primarily over WebSocket [22], chosen for its bidirectional communication, widespread support, and compatibility with existing browser infrastructure. The default endpoint is `ws://{host}:{port}/` with `127.0.0.1:9222` as the default binding.

In v0.1, a single WebSocket connection carries a single session. The full specification supports multiplexing multiple sessions over one connection, with session isolation enforced by the server. Message ordering follows two rules: (1) requests and responses are matched by a client-generated id field, and (2) the server preserves per-page action ordering unless the client opts into concurrent execution. Events are out-of-band and do not carry an id field.

4.2 Encoding

AWP supports two encoding formats:

- **JSON** (default in v0.1): UTF-8 text frames with lower_snake_case keys.
- **MessagePack** [28] (planned for v0.2): Binary serialization yielding 30–50% smaller messages.

Optional per-message compression using zstd [29] is specified but deferred to v0.2.

4.3 Message Model

Every AWP message conforms to a common envelope with three types. A *request* (client to server) carries id, method, and params:

```
{
  "id": "msg-1",
  "type": "request",
  "method": "page.navigate",
  "params": {
    "session_id": "s_a1b2c3d4",
    "url": "https://example.com"
  }
}
```

A *response* (server to client) carries id and exactly one of result or error:

```
{
  "id": "msg-1",
  "type": "response",
  "result": {
    "url": "https://example.com",
    "status": 200,
    "som_ready": true
  }
}
```

An *event* (server to client, unsolicited) carries method and params but no id:

```
{
  "type": "event",
  "method": "page.som_mutation",
  "params": {
    "session_id": "s_a1b2c3d4",
    "cursor": "c:12346",
    "patch": [
      {"op": "replace", "path": "/regions/0/items/2/badge",
       "value": "4"}
    ]
  }
}
```

Unknown fields MUST be ignored for forward compatibility. Requests may include a meta object with idempotency_key and trace_id for distributed tracing.

4.4 Error Model

AWP defines a fixed set of 11 error codes (Table 1).

Table 1: AWP Error Codes

Code	Meaning
`INVALID_REQUEST`	Malformed message, missing fields, unknown method
`UNSUPPORTED`	Feature or method not supported
`NOT_FOUND`	Session, page, or element not found
`TIMEOUT`	Operation exceeded timeout
`CONFLICT`	Concurrent state conflict
`RATE_LIMITED`	Server rate limit exceeded
`PERMISSION_DENIED`	Action not permitted
`NAVIGATION_FAILED`	HTTP, DNS, or TLS error
`SCRIPT_ERROR`	JavaScript evaluation error
`SKILL_ERROR`	Wasm skill execution failed
`INTERNAL`	Unexpected server error

Error responses include structured details:

```
{
  "id": "msg-5",
  "type": "response",
  "error": {
    "code": "NOT_FOUND",
    "message": "Element reference not found in current SOM",
    "details": {
      "ref": "e_8f3a1b",
      "strategy": "direct_lookup"
    }
  }
}
```

4.5 Capability Negotiation

Upon WebSocket connection, the client MUST send `awp.hello` as its first message:

```
{
  "id": "1",
  "type": "request",
  "method": "awp.hello",
  "params": {
    "client_name": "my-agent",
    "client_version": "0.1.0",
    "awp_version": "0.1"
  }
}
```

The server responds with its version and feature list:

```
{
  "id": "1",
  "type": "response",
  "result": {
    "awp_version": "0.1",
    "server_name": "plasmate",
    "server_version": "0.1.0",
    "features": ["som.snapshot", "act.primitive", "extract"]
  }
}
```

The `features` array enables clients to degrade gracefully. Protocol versioning uses `MAJOR.MINOR` semantics with capability negotiation.

5 Core Methods

AWP v0.1 implements seven methods covering the complete agent interaction loop. This section describes each with complete wire-level examples drawn from the AWP MVP specification [23].

5.1 `awp.hello` – Handshake

The mandatory first message establishing protocol version and available features. The server MUST NOT process any other method before a successful handshake.

5.2 `session.create` – Create Session

Creates an in-memory browsing session.

```
{
  "id": "2",
  "type": "request",
  "method": "session.create",
  "params": {
    "user_agent": "Mozilla/5.0 ...",
    "locale": "en-US",
    "timeout_ms": 30000
  }
}
```

Response:

```
{
  "id": "2",
  "type": "response",
  "result": {
    "session_id": "s_a1b2c3d4"
  }
}
```

All parameters are optional with sensible defaults. In v0.1, one session is active per connection. The full specification supports multiple concurrent sessions with configurable limits (default 500 sessions, 16 pages per session).

5.3 `session.close` – Destroy Session

Destroys a session and frees all associated resources.

5.4 `page.navigate` – Fetch and Parse

Fetches a URL, parses HTML with html5ever [24], and compiles a SOM snapshot:

```
{
  "id": "10",
  "type": "request",
  "method": "page.navigate",
  "params": {
    "session_id": "s_a1b2c3d4",
    "url": "https://news.ycombinator.com",
    "timeout_ms": 15000
  }
}
```

Response:

```
{
  "id": "10",
  "type": "response",
  "result": {
    "url": "https://news.ycombinator.com",
    "status": 200,
    "content_type": "text/html; charset=utf-8",
    "html_bytes": 42891,
    "som_ready": true,
    "load_ms": 287
  }
}
```

This single method replaces the multi-step CDP sequence of `Page.navigate`, waiting for `Page.loadEventFired`, `DOM.getDocument`, and `DOM.describeNode` calls. The full specification defines four wait strategies: `dom_content_loaded`, `load`, `network_idle`, and `som_stable`.

5.5 `page.observe` – Retrieve SOM Snapshot

Returns the current page's Semantic Object Model:

```
{
  "id": "20",
  "type": "request",
  "method": "page.observe",
  "params": {
    "session_id": "s_a1b2c3d4"
  }
}
```

Response (abbreviated):

```
{
  "id": "20",
  "type": "response",
  "result": {
    "som": {
      "som_version": "0.1",
      "url": "https://news.ycombinator.com",
      "title": "Hacker News",
      "lang": "en",
      "regions": [
        {
          "id": "r_nav",
          "role": "navigation",
          "elements": [
            {"id": "e_a1b2c3", "role": "link",
             "text": "Hacker News",
             "attrs": {"href": "https://news.ycombinator.com"},
             "actions": ["click"]}
          ]
        },
        {
          "id": "r_main",
          "role": "main",
          "elements": [
            {"id": "e_story1", "role": "link",
             "text": "Show HN: Plasmate",
             "attrs": {"href": "https://plasmate.app"},
             "actions": ["click"]}
          ]
        }
      ],
      "meta": {
        "html_bytes": 42891,
        "som_bytes": 1847,
        "element_count": 94,
        "interactive_count": 38
      }
    }
  }
}
```

The SOM response includes compression metadata: 42,891 bytes of HTML compressed to 1,847 bytes, a 23.2× reduction.

5.6 page.act – Execute Action

The central interaction method:

```
{
  "id": "30",
  "type": "request",
  "method": "page.act",
  "params": {
    "session_id": "s_a1b2c3d4",
    "intent": {
      "action": "click",
      "target": {"ref": "e_story1"}
    }
  }
}
```

Response:

```
{
  "id": "30",
  "type": "response",
  "result": {
    "status": "ok",
    "resolved": {
      "element_id": "e_story1",
      "role": "link",
      "text": "Show HN: Plasmate..."
    },
    "effects": {
      "navigated": true,
      "som_changed": true
    }
  }
}
```

The response confirms which element was acted upon, what effects occurred, and whether the SOM changed. v0.1 supports four action types: `click`, `type`, `select`, and `scroll`.

5.7 `page.extract` – Structured Data Extraction

Extracts structured data from the SOM using declarative field queries:

```
{
  "id": "40",
  "type": "request",
  "method": "page.extract",
  "params": {
    "session_id": "s_a1b2c3d4",
    "fields": {
      "title": {"role": "heading", "level": 1},
      "links": {"role": "link", "all": true,
                "props": ["text", "href"]},
      "price": {"text_match": "\\$\\d+\\.\\d{2}"}
    }
  }
}
```

Response:

```
{
  "id": "40",
  "type": "response",
  "result": {
    "data": {
      "title": "Widget Pro",
      "links": [
        {"text": "Home", "href": "/"},
        {"text": "Products", "href": "/products"}
      ],
      "price": "$49.99"
    },
    "provenance": {
      "title": "e_1a2b3c",
      "price": "e_7d8e9f"
    }
  }
}
```

The `provenance` field maps each extracted value to the SOM element ID from which it was derived, enabling verification and re-targeting.

6 Intent-Based Interaction

6.1 The Problem with Coordinate-Based Interaction

CDP's interaction model requires agents to perform a multi-step process for every click: (1) query the DOM to find the target element; (2) retrieve the element's bounding box via `DOM.getBoxModel`; (3) calculate the center coordinates; (4) dispatch `Input.dispatchMouseEvent` at those coordinates. This process is fragile (coordinates change with viewport size, zoom, and dynamic content), expensive (multiple round-trips), and semantically meaningless.

6.2 AWP's Intent Model

AWP replaces coordinate-based interaction with an intent model. An intent consists of an *action* (click, type, select), a *target* (element reference, semantic query, or fallback chain), an optional *value*, and optional *options*.

6.3 Target Resolution

Targets are resolved through a priority chain:

1. **Direct reference** (`ref`): Look up by stable SOM ID. Fastest and most deterministic.
2. **Semantic query** (`text` + `role`): Find element matching role and visible text. Resilient to DOM changes.
3. **CSS selector** (`css`): Traditional CSS selector fallback.
4. **Fallback chain**: Alternative targets tried in sequence.

Example with full fallback chain:

```
{
  "target": {
    "ref": "e_9f2c",
    "fallback": [
      {"text": "Add to Cart", "role": "button"},
      {"css": "button[data-testid='add-to-cart']"}
    ]
  }
}
```

6.4 Intent Levels

The full specification defines a namespace hierarchy:

- `primitive.*`: Low-level browser events (click, type, select). v0.1 only.
- `semantic.*`: SOM-aware actions (e.g., `semantic.fill_form`).
- `intent.*`: Goal-oriented actions (e.g., `intent.checkout`).
- `skill.*`: WebAssembly skill actions (e.g., `skill.stripe_pay`).

Each level builds on the one below. A `semantic.fill_form` decomposes into multiple `primitive.type` and `primitive.select` actions. An `intent.checkout` might invoke a skill or fall back to semantic actions.

6.5 Action Effects

Every `page.act` response includes an `effects` object. In the full specification, this includes a `confidence` field (0.0 to 1.0) indicating target resolution certainty, and a `som_cursor` for efficient incremental observation.

7 SOM Integration

7.1 The Semantic Object Model

AWP's observation model is built on the Semantic Object Model (SOM) [3], a token-efficient web page representation. A SOM document organizes page content into typed regions (navigation, main, form, complementary, footer) containing typed elements with stable IDs, semantic roles, visible text, available actions, and role-specific attributes.

7.2 Element Types and Affordances

SOM elements carry an `actions` array declaring available affordances (Table 2).

Table 2: SOM Element Types and Affordances

Role	**Interactive**	**Actions**	**HTML Sources**
link	Yes	[click]	`<a href>`
button	Yes	[click]	`<button>`, `input[type=submit]`
text_input	Yes	[type, clear]	`<input type="text|email|...">`
textarea	Yes	[type, clear]	`<textarea>`
select	Yes	[select]	`<select>`
checkbox	Yes	[toggle]	`<input type="checkbox">`
radio	Yes	[select]	`<input type="radio">`
heading	No	–	`<h1>` through `<h6>`
paragraph	No	–	`<p>`
image	No	–	`<img>`, `<picture>`
table	No	–	`<table>`
list	No	–	`<ul>`, `<ol>`

Interactive elements are never dropped by SOM budget limits, ensuring agents always have a complete view of available actions.

7.3 Stable Element IDs

SOM element IDs are deterministic, computed as:

```
element_id = "e_" + hex(sha256(
    origin + "|" + role + "|" + name + "|" + dom_path
))[0..12]
```

This ensures the same element on the same page always produces the same ID across fetches, enabling persistent references.

7.4 Incremental Observation via Mutations

For dynamic pages, the full specification supports cursor-based incremental observation using JSON Patch [25] semantics:

```
{
  "type": "event",
  "method": "page.som_mutation",
  "params": {
    "session_id": "s:1",
    "cursor": "c:12346",
    "patch": [
      {"op": "replace",
       "path": "/regions/0/items/2/badge", "value": "4"},
      {"op": "add",
       "path": "/regions/1/elements/-",
       "value": {"id": "e_new1", "role": "paragraph",
                 "text": "New comment added"}}
    ]
  }
}
```

Agents store the latest cursor and request only changes since that point, analogous to event sourcing.

8 Session and State Management

An AWP session encapsulates browsing state: cookies, storage, proxy configuration, and loaded pages. The full specification supports session persistence (surviving server restarts), TTL-based expiration, and portable session export/import.

The server advertises concurrent session limits during handshake:

```
{
  "limits": {
    "max_sessions": 500,
    "max_pages_per_session": 16
  }
}
```

Sessions maintain an internal cookie jar populated automatically from HTTP responses. The full specification exposes explicit cookie management via `session.cookies.get/set/clear`. Multi-page support (multiple tabs per session) is specified via `page.create/close`, with v0.1 using one implicit page per session.

9 Extensibility via WebAssembly Skills

Web automation often requires domain-specific logic (Stripe checkouts, OAuth flows, CAPTCHA handling). AWP addresses this through WebAssembly skills: Wasm modules that declare actions, receive SOM and arguments as input, and emit sequences of primitive actions.

Agents discover skills via `skills.list` and invoke them via `skills.invoke`:

```
{
  "id": "70",
  "type": "request",
  "method": "skills.invoke",
  "params": {
    "session_id": "s:1",
    "page_id": "p:1",
    "skill": "stripe-checkout",
    "action": "stripe_pay",
    "args": {
      "card_number": "4242424242424242",
      "expiry": "12/30",
      "cvc": "123"
    }
  }
}
```

Skills run in a sandboxed Wasm environment with no direct DOM, filesystem, or network access [30]. Skill invocation requires the `skills:invoke` permission scope.

AWP encourages a "skill with fallback" pattern: agents check for a relevant skill first, then fall back to primitive actions if unavailable.

10 Security Model

10.1 Deployment Modes and Authentication

AWP supports three deployment modes: local (no auth), LAN (shared secret), and cloud (API key with per-tenant permissions). Authentication methods include `auth.none`, `auth.bearer`, and `auth.hmac`.

10.2 Permission Scopes

Fine-grained scopes control access to protocol features (Table 3).

Table 3: AWP Permission Scopes

Scope	Governs
`som:read`	Reading SOM snapshots and mutations
`page:act`	Executing primitive and semantic actions
`page:evaluate`	Running arbitrary JavaScript (high risk)
`session:write`	Creating, closing, modifying sessions
`network:configure`	Proxy and impersonation changes
`skills:invoke`	Executing WebAssembly skills

The `page:evaluate` scope is separated from `page:act` because JavaScript evaluation enables arbitrary code execution. The server MAY enforce per-domain allowlists.

10.3 Trust Boundaries

Three trust boundaries: (1) agent to engine (WebSocket, TLS), (2) engine to web (HTTP with request policies), and (3) engine to skills (Wasm sandbox).

11 CDP Compatibility and Migration

AWP proposes a three-phase migration: (1) a shim layer translating CDP calls into AWP primitives for existing Playwright/Puppeteer scripts; (2) native AWP adoption by agent frameworks;

(3) AWP-native operation with an optional CDP shim.

The shim maps `DOM.querySelector` to SOM queries, `Input.dispatchMouseEvent` to `page.act` with click intent, and `Page.navigate` to `page.navigate`. CDP methods depending on Chromium internals (debugger, CSS profiling, GPU compositing) cannot be shimmed.

12 Comparison with CDP

Table 4 presents a systematic comparison across fifteen dimensions.

Table 4: AWP vs CDP Comparison

Dimension	**CDP**	**AWP**
Primary consumer	Human developer	AI agent
Method count	300+ (40+ domains)	7 (v0.1), ∼25 (full)
Output format	Raw DOM (50K+ nodes)	SOM (16.6× compression)
Interaction model	Pixel coordinates	Semantic targeting
Session scope	Tab in single browser	Durable session with state
State management	Implicit browser state	Explicit create/close/export
Token efficiency	∼50K tokens/page	∼3K tokens/page
Concurrency	1 connection per context	Multiplexed sessions
Element addressing	Opaque node IDs, CSS/XPath	Stable semantic IDs
Extensibility	Chrome extensions	Wasm skills (sandboxed)
Error model	Domain-specific	11 fixed codes
Stability	Tip-of-tree: none	Semantic versioning
Observation	Full DOM or nothing	Snapshot or mutations
Wait strategies	`loadEventFired`, polling	4 built-in strategies
Authentication	None	Bearer, HMAC, scoped

12.1 Quantitative Comparison

Surface area. CDP defines 300+ methods; AWP v0.1 implements 7. Even the full AWP specification targets approximately 25, a 12× reduction.

Token cost per observation. SOM achieves 16.6× mean compression (94% savings) across a 49-site benchmark [3]. At GPT-4 pricing ($10/M input tokens), observation cost drops from $0.50 to $0.03 per page.

Round-trips per interaction. A CDP click requires minimum 3 round-trips (query, coordinates, dispatch). An AWP click requires 1 (`page.act` with target reference).

Memory per session. CDP requires a full Chromium context (300–500MB). Plasmate requires approximately 30MB [6].

13 Implementation Status

13.1 Plasmate v0.1

AWP v0.1 is implemented in the Plasmate engine [6]: approximately 25,000 lines of Rust using html5ever [24], rusty_v8 [26], tokio, and serde. All 7 MVP methods are implemented with JSON/WebSocket transport. A Python client SDK is included.

13.2 Benchmark Results

Published results from the 49-site benchmark [3]: 16.6× mean compression, 10.5× median, 94% token cost savings. Range: 0.7× (example.com) to 116.9× (cloud.google.com). A nightly CI scorecard tests 100 websites.

13.3 Deferred Features

MessagePack encoding, zstd compression, connection multiplexing, SOM mutation mode, session persistence, cookie API, WebAssembly skills, authentication, proxy/stealth, screenshots, and JavaScript evaluation are specified but deferred to v0.2.

14 Conclusion

The Chrome DevTools Protocol has served as the backbone of web automation for over a decade. It was designed for human developers inspecting rendered web pages. AI agents are a fundamentally different consumer: they need semantic understanding, not pixel rendering; token-efficient observations, not 50,000-node DOM trees; intent-based interaction, not coordinate dispatching; massive concurrency, not single-user sessions.

The Agent Web Protocol addresses these needs through a minimal, purpose-built design. Seven methods replace hundreds. Semantic element targeting replaces coordinate clicking. The SOM replaces the DOM as the primary observation format, achieving 16.6× token compression. Cursor-based mutations replace full-page re-observation. WebAssembly skills replace monolithic automation scripts.

AWP does not seek to replace CDP for its intended use case. Browser debugging and inspection remain valuable. AWP's claim is narrower: for AI agent workloads, a purpose-built protocol is categorically more efficient than repurposing a debugging interface. The v0.1 implementation demonstrates this with a working protocol, a reference engine, and reproducible benchmarks.

The protocol specification is open (Apache 2.0) and is developed through the W3C Web Content Browser for AI Agents Community Group [7]. We invite the research community, agent framework developers, browser engine implementors, and web infrastructure providers to participate in its evolution.

References

[1] Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/.

[2] Chrome DevTools Protocol – Input Domain. https://chromedevtools.github.io/devtools-protocol/tot/Input/.

[3] D. Hurley. The Semantic Object Model: A Token-Efficient Web Representation for AI Agents. Plasmate Labs, 2026.

[4] D. Hurley. The Agentic Web: Rethinking Web Infrastructure for Machine Consumption. Plasmate Labs, 2026.

[5] Playwright System Requirements. https://playwright.dev/docs/library.

[6] Plasmate: An Agent-Native Headless Browser Engine. https://plasmate.app.

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

[8] Puppeteer. https://pptr.dev/.

[9] Playwright. https://playwright.dev/.

[10] Selenium. https://www.selenium.dev/.

[11] W3C WebDriver Specification. https://www.w3.org/TR/webdriver2/.

[12] W3C WebDriver BiDi Specification. https://w3c.github.io/webdriver-bidi/.

[13] Selenium Wire. https://github.com/wkeeling/selenium-wire.

[14] W3C WAI-ARIA Specification. https://www.w3.org/TR/wai-aria-1.2/.

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

[16] Stagehand. https://github.com/browserbase/stagehand.

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

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

[19] Lightpanda. https://lightpanda.io/.

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

[21] AWP Draft Specification v0.1. Plasmate Labs, 2026.

[22] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, 2011.

[23] AWP v0.1 MVP – Implementable Specification. https://docs.plasmate.app/awp-mvp.

[24] html5ever. https://github.com/servo/html5ever.

[25] P. Bryan and M. Nottingham. JavaScript Object Notation (JSON) Patch. RFC 6902, 2013.

[26] rusty_v8: Rust bindings for V8. https://github.com/nickel-org/rusty_v8.

[27] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, 1996.

[28] MessagePack Specification. https://msgpack.org/.

[29] Zstandard Compression Algorithm. https://facebook.github.io/zstd/.

[30] WebAssembly Specification. https://webassembly.github.io/spec/core/.

[31] tiktoken: BPE tokeniser for OpenAI models. https://github.com/openai/tiktoken.

[32] LangChain. https://github.com/langchain-ai/langchain.

[33] M. Koster et al. Robots Exclusion Protocol. RFC 9309, 2022.