Your AI, Your Context

Raffi Krikorian <raffi@mozilla.org>

January 2026

Your context is scattered across the internet. Emails in Gmail, documents in Drive, purchase history on Amazon, calendar in Outlook, notes in Notion. When you want AI to help with any of it, you connect these services to third-party models directly — on terms you don't set. Which data gets sent, which model processes it, what gets retained. You repeat your preferences to every new AI. You're stuck with whatever model a website chose rather than the one you chose.

We think there's a different model. Your AI, your context, your preferences — these become resources the browser manages on your behalf, the way it manages camera access or location today. Websites don't get access by default. They request it, you grant or deny, and the browser enforces your decision. You bring your own AI to any website. Websites provide domain-specific tools without managing inference. Context flows on your terms.

This document describes the technical foundation: a developer toolkit (APIs, SDKs, standards) that enables a new category of AI-native web experiences. The centerpiece is the **Web Agent API**, a browser-level standard built on the Model Context Protocol (MCP). We've built a sketch of it in **Harbor**, a Firefox extension. But the goal isn't a Firefox feature — it's a standard any browser, OS, or application could implement.

The Problem

For users. AI today requires constant context-shuttling. You manually bridge the gap between your documents, emails, calendars and the AI that could help with them. Switch from one model to another and all accumulated context disappears. There's no secure way to authorize payments or verify identity within AI interactions. You're a renter in someone else's infrastructure — not an owner.

For websites. A news publisher wants AI-native experiences — deep research across decades of archives — but can't absorb inference costs. An e-commerce site wants personalization ("find a hub compatible with my MacBook") but doesn't have your context. A SaaS app wants sophisticated AI features but doesn't want the burden of model deployment. Each builds the same integration from scratch, or doesn't build it at all.

For developers. Building AI into web experiences today requires deep AI expertise and significant API spend. You manage model connections, handle authentication, build tool infrastructure, and pay for inference — all before delivering value. The plumbing should be platform infrastructure, not application code.

Not every site will adopt this. SaaS that sees AI as a moat won't expose tools to third-party models. That's fine — this isn't meant to replace embedded AI. It's for the sites that can't or won't build their own: publishers who can't afford inference, retailers who want personalization without user context, vertical SaaS that wants to focus on product rather than model operations.

Context as a Resource

A decade ago, browsers introduced permission prompts for cameras and microphones — sensitive resources that websites could request but not access without consent. The Web Agent API extends this model to AI and context.

The browser becomes a secure repository for identity, credentials, preferences, and accumulated AI context. When a website needs identity verification, payment authorization, or access to user context, the browser mediates — just as it mediates camera access. The permission model follows familiar patterns: "Allow Expedia to see my Calendar," with grant types ranging from one-time access to persistent grants.

Bring Your Own AI

The current model: websites embed AI, users have no choice. The Web Agent API flips this.

Websites declare their tools via `<link rel="mcp-server">`. Users bring their preferred AI — Claude, ChatGPT, a local Llama, whatever they've configured. The user's AI gains capabilities from the website's tools without the website managing inference. Preferences and history stay with the user. The website pays nothing for AI.

A realistic objection: most users don't have paid AI subscriptions. Local inference is the practical default. This is actually an argument for the browser layer — it can route between local models (simple tasks, privacy-sensitive contexts) and cloud models (when the user has a subscription and the task warrants it). Even basic browser features benefit: smarter autocomplete, URL bar suggestions, routed to small fast local models. As local model quality improves, the threshold shifts.

Why the Browser

This can't live on the model provider side — users switch providers, and context would remain splintered. It could live at the OS level, but users span operating systems, and the companies that make them have incentive to maintain walled gardens. The browser is the natural home: it's already the user agent, it's where knowledge workers spend their time, and it can be independent of both model providers and operating systems.

The browser layer also enables **enterprise policy control**. Organizations can govern which AI models employees use through the same browser policy mechanism used for extensions, security settings, and site permissions — restrict providers to an approved list, require local-only execution for sensitive workloads, or route inference through a corporate endpoint.

The Web Agent API is designed as a portable standard, not a Firefox-specific feature. Firefox is where we build and demonstrate. The standard is what matters.

MCP as Foundation

We build on the Model Context Protocol, an open standard that defines how AI systems connect to tools. MCP provides standardized tool definitions, structured schemas, and discovery mechanisms. Think of it as USB for AI — before USB, every peripheral needed its own connector.

MCP is a starting point, not an endpoint. The architecture separates the browser API surface from the underlying tool protocol. As MCP evolves or complementary standards emerge, implementations adapt without breaking web applications. The bet is on the pattern — browser-mediated AI with user-controlled tool access — not on any single protocol.

The Web Agent API

The Web Agent API is the specification at the heart of this proposal. It defines the surfaces web applications use (`window.ai`, `window.agent`), the permission model that governs access, and the contract between browsers and websites.

Chrome Compatibility

The API is designed for compatibility with Chrome's emerging Prompt API — apps built for Chrome's `window.ai` surface should work here. But the Web Agent API extends beyond basic prompting to include tool discovery, tool execution, autonomous agent loops, and browser context access. These are the capabilities that make Bring Your Own AI possible.

`window.ai` — Model Access

Compatible with Chrome's Prompt API, plus extensions for provider choice:

JavaScript
```javascript
// Basic prompting (Chrome-compatible)
```

```
const session = await ai.languageModel.create({
  systemPrompt: "You are a helpful assistant"
});
const response = await session.prompt("Summarize this page");

// Extended: choose your provider and model
const providers = await ai.providers.list();
const session = await ai.languageModel.create({
  provider: "anthropic",
  model: "claude-sonnet",
  allowTools: true
});
```

`window.agent` — Tools and Autonomous Execution

The `agent` namespace is where the Web Agent API goes beyond prompting:

JavaScript
```javascript
// Discover tools the website offers

const tools = await agent.tools.list();

// Call a tool directly
const result = await agent.tools.call(
  "newsarchive/search",
  { query: "climate policy", dateRange: "2020-2024" }
);

// Run an autonomous agent loop
for await (const event of agent.run({
  task: "find accessories compatible with my laptop",
  maxToolCalls: 10
})) {
  if (event.type === "token") process.stdout.write(event.text);
  if (event.type === "tool_call") console.log(`using:
${event.tool}`);
}
```

MCP Server Discovery

Websites declare MCP servers with a link tag:

```html
<link rel="mcp-server"
      href="https://example.com/mcp"
      title="Example Tools">
```

The browser discovers these automatically. Servers can also be registered at runtime via `agent.mcp.register()`.

API Summary

Area	Key Methods
Model access	`ai.languageModel.create()`, `session.prompt()`, `session.promptStreaming()`
Provider choice	`ai.providers.list()`, `ai.runtime.getBest()`
Tool discovery	`agent.tools.list()`, `agent.mcp.discover()`
Tool execution	`agent.tools.call()`
Autonomous agents	`agent.run()` — yields events: `status`, `tool_call`, `tool_result`, `token`, `final`, `error`
Page context	`agent.browser.activeTab.readability()`
Server registration	`agent.mcp.register()`

The full API reference is in the Web Agent API Specification.

Permission Model

All operations require explicit user consent. Permissions are granted per-origin with complete isolation.

Scopes

Scope	Risk	What It Enables
`model:prompt`	Low	Basic text generation
`model:tools`	Medium	AI with tool calling
`mcp:tools.list`	Low	Listing available tools
`mcp:tools.call`	High	Executing tools
`mcp:servers.register`	Medium	Registering MCP servers at runtime
`browser:activeTab.read`	Medium–High	Reading current page content
`context:identity`	High	Accessing user identity
`context:payment`	High	Accessing payment methods

Grants

Grant	Duration	Behavior
`ALLOW_ONCE`	10 min or tab close	Temporary; re-prompts after
`ALLOW_ALWAYS`	Persistent	No future prompting for this scope
`DENY`	Persistent	Blocks re-prompting from this origin

Tool Allowlisting

The `mcp:tools.call` permission is necessary but not sufficient. Even with it granted, an origin has access to **no tools** by default. Users explicitly allowlist specific tools per origin. This prevents a blanket "allow tools" grant from exposing everything.

Origin Isolation

Each origin operates in complete isolation — separate permissions, separate tool allowlists, separate session state, separate rate limits, separate MCP registrations. Origin is verified by the browser, not by page-provided values.

Errors are structured (`ERR_PERMISSION_DENIED`, `ERR_TOOL_NOT_ALLOWED`, `ERR_TOOL_NOT_FOUND`, etc.). See the full spec for the complete error reference.

Implementation Flexibility

The Web Agent API is a standard. Implementations can vary — what matters is that they expose the same surfaces to web applications and speak MCP to tools.

Possible approaches:

- **Browser extension + external bridge** — An extension bridges to a process that hosts MCP servers and manages LLM connections. Harbor takes this approach. Works across browsers without engine changes.
- **Browser extension + WASM** — Everything in-browser, no external process.

- **Native browser integration** — MCP hosting in the engine, like Chrome's Prompt API with Gemini Nano.
- **OS-level service** — Shared across browsers and applications via IPC.
- **Cloud proxy** — For mobile or resource-constrained environments.

What You Could Build

- **A news site** exposes an MCP server to its 20-year archive. Readers run deep research with their own AI. The publisher pays nothing for inference.
- **An e-commerce site** provides product search tools. Your AI brings the context ("I own a MacBook Pro M3, prefer brand-name electronics") and surfaces compatible accessories without you re-explaining.
- **A SaaS app** offers document analysis, intelligent search, workflow automation — by exposing domain tools, while focusing on product rather than model APIs.
- **Privacy-preserving personalized ads** where context never leaves the device.
- **Agent-to-agent coordination** for group planning across users.
- **Portable AI identity** that persists across model switches.

Get Involved

This is a proposal, not a finished standard. We're figuring this out and we want thought partners.

Harbor is the sketch — a working Firefox extension that proves the architecture. Start there if you want to build on it:

- Harbor repo + getting started
- Web Agent API Specification
- Building on the Web Agent API

We want feedback from:

- **Web developers** — Would you add `<link rel="mcp-server">` to your site? What tools would you expose? What's missing from the API?
- **AI tool builders** — Does this architecture work for your use case? What would you build on top of it?

- **Security and privacy folks** — Does the permission model hold up? What attack vectors are we missing?
- **Standards people** — Is the Web Agent API on the right track to become a real standard?

We're not trying to own this. We're trying to figure out what AI on the user's side looks like. If the pattern is right, other browsers will implement it — just as they implemented Service Workers after one browser shipped first.