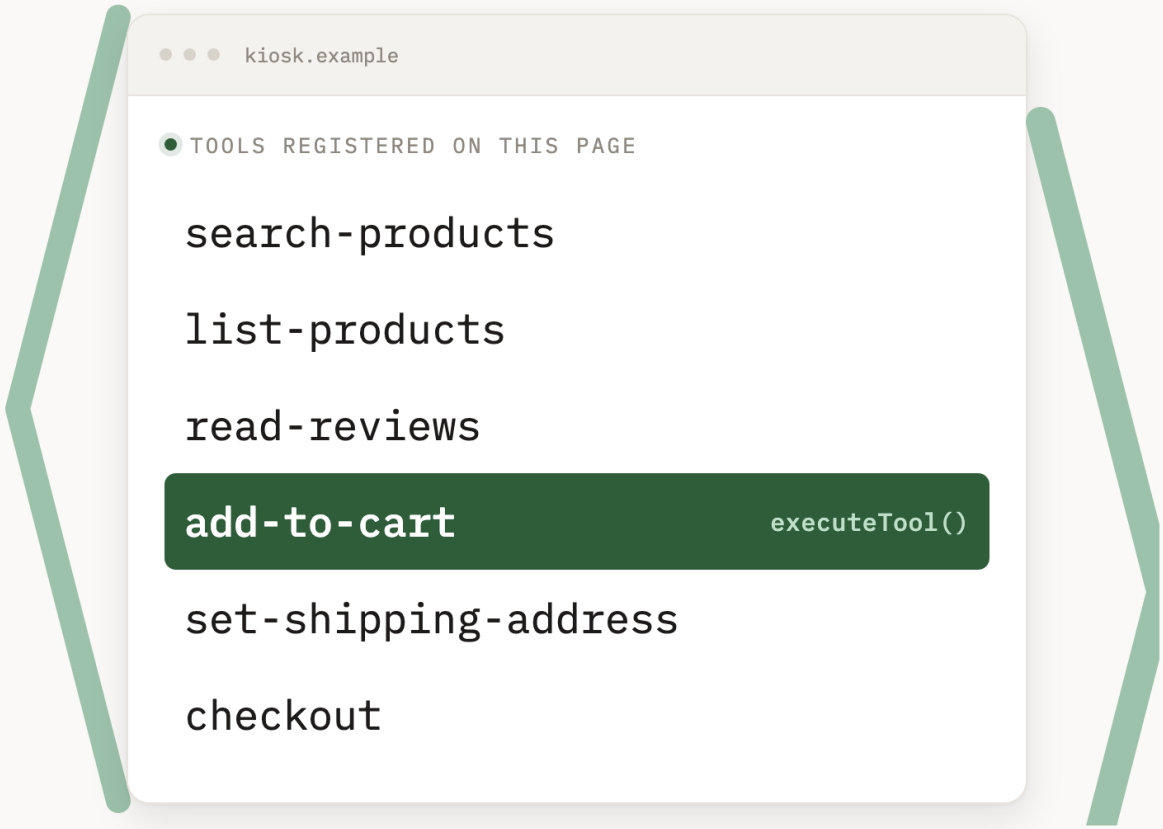
WebMCP in Practice

A short guide to building websites AI agents can use



Sandeep Kumar Patel

A TECHNICAL BOOK

WebMCP in Practice

A short guide to building websites AI agents can use

Sandeep Kumar Patel

September 2026

COPYRIGHT

Found a stale fact, a bug or an error? Report it at **tutorialsavvy.com**.

First edition, 2026.

Contents

Foreword

I have spent seventeen years watching the web absorb one new capability after another — from jQuery smoothing over browser differences, through the rise of component frameworks, to a browser that can now render a page, hold a WebSocket open, and run a trained model, all in the same tab. Each of those shifts asked developers to learn a new way of thinking about the browser, not just a new API surface. WebMCP is the next one, and it asks something specific: that you stop thinking of your page as something only a human reads, and start thinking of it as something an agent can act on, too.

That is a real shift, and it deserves a real answer to the obvious question: should an agent get its own backend service, should it drive the browser the way a person does, or should the page itself hand the agent a list of things it can do? I did not know the answer when I started writing this book. I have one now. It came from building a small shop the third way, and from checking every claim in these pages against that running app rather than against the specification.

What follows is not a pitch for WebMCP. It is an account of what that question turned up, chapter by chapter. The next few pages set the terms for how that account was kept; if you build one real tool by the time you finish reading it, the book has done its job.

— Sandeep Kumar Patel

Preface

This is a short book about a young API.

WebMCP lets a web page hand an AI agent a list of things it can do: search this catalog, add this to the cart, set this address. The tools are described in the page's own words, they run the page's own code, and they live inside the tab the user already has open. It is a good idea. It is also a W3C Community Group draft that renamed its own entry point partway through, with one implementation, behind an origin trial — Chrome's mechanism for letting a site run an unshipped feature for real visitors, for a limited window, by serving a token. Both of those things are true at once, and this book is written for someone who wants to build on it anyway, with their eyes open.

Who this is for

A web developer who knows the DOM and can read JavaScript. You don't need to have used the Model Context Protocol or worked with agents. The book argues its case and shows the costs, and a skeptical reader is a welcome one. What you do need is a willingness to run things, because this book was written by running things.

What you need

The book is hands-on, so three things need to be in place before Chapter 2. None costs anything.

- **A browser.** Chrome 149 through 156, with two flags turned on, shows everything in this book natively (§2.2). In any other browser, the polyfill that Chapter 9 describes stands in for native support.
- **Node.js and npm.** Node.js 18, 20 or newer runs Kiosk and its tests. `npm install` fetches the rest.
- **JavaScript and the DOM.** That is the whole prerequisite. You don't need an AI account or an API key, and no chapter calls a model.

Getting the code

You will get more from this book if you run Kiosk yourself. Kiosk and the scripts around it are one download, MIT-licensed:

```
https://drive.google.com/file/d/1CH8WsLG42B--O3aKYJJHHypAxwa6Rs_i/view
```

Unzip it, run `npm install` and `npm run dev` inside `code/kiosk`, and open `http://localhost:5173`. Appendix B is the guided tour of what's in it.

The one rule behind everything

Every runnable listing here was extracted from a real, running app and checked against it. Every claim about how the API behaves was tested in a real browser, against the real polyfill, and where possible against native Chrome, not reasoned from the spec text alone. That discipline caught the book being wrong more than once, and where it did, the correction was left visible rather than quietly edited away.

It also means the book is full of dated facts: version numbers, Chrome milestones, trial windows, "as of this writing." Every one of those is a photograph, not a promise. Where a number matters, the book tells you the command to check it yourself.

What's in this book

Here is the book on one page.

- **Chapter 1, Why WebMCP,** compares three ways to connect an agent to a site and says where the spec stands.
- **Chapter 2, Your First Tool with HTML,** turns a search form into a tool with attributes alone.
- **Chapter 3, Registering Tools with JavaScript,** covers `registerTool`, `executeTool`, and why the entry point is `document.modelContext`.
- **Chapter 4, Writing Good Tool Descriptions,** is about descriptions, schemas, return values and errors an agent can use.
- **Chapter 5, Changing Tools as the Page Changes,** makes tools appear and disappear with the contents of the cart.
- **Chapter 6, Prompt Injection,** runs a real attack against Kiosk's own tools.
- **Chapter 7, Tool Annotations and Limits,** covers the three hints, `exposedTo`, what none of them enforces, and how a page can ask the shopper.
- **Chapter 8, Testing Tools,** tests against the real platform instead of a mock, smoke-tests in a real browser, and logs tool calls.

- **Chapter 9, Shipping to Production,** covers feature detection, origin trials, the polyfill, running without tools, and what production adds.
- **Chapter 10, What's Next,** sorts the settled from the open.
- **Appendix A** is the API surface in one place, dated. **Appendix B** is the tour of Kiosk's own code. **Appendix C** answers the exercises at the end of Chapters 2 to 9. A glossary, further reading and an index follow.

How to read it

Straight through. It is built around one app, **Kiosk**, a small online shop that grows as the chapters go, and each chapter assumes the last. Chapter 2 gives you a working tool by adding a couple of HTML attributes to a form you already have. If that form submits to the server, that is the whole job. If it answers in the page the way Kiosk's does, it needs a little more wiring. Either way it is the fastest payoff in the book, and a fine place to stop.

Chapters 6 and 7 are the security core, and they are not optional reading. The whole design rests on the agent running as the logged-in user, which is both the best argument for WebMCP and the whole of its threat model.

Chapters 2 to 9 each end with exercises on Kiosk. Try each before you read its answer in Appendix C. `npm test` in `code/kiosk` runs every answer.

Conventions Used in This Book

Code listings are shown in a monospace font. Every runnable listing was extracted from the companion code download and verified there; nothing is pseudocode. Appendix B tells you where to get it. Where a listing is shown at an unusual indentation, it's because it's a verbatim excerpt from a larger file (nested inside a function, say), and a note under it says so.

Figures are numbered by chapter — Figure 2-1 is the first figure in Chapter 2 — with the caption below the image. Every screenshot in this book is a real capture, of Kiosk or of DevTools inspecting it, in Chrome with native WebMCP enabled, taken by a script in the repository rather than staged by hand.

Cross-references use `§` for a section and "Chapter N" for a chapter. `§4.4` is section 4 of Chapter 4; "see Chapter 6" is the whole chapter. Every one of them points at something real.

`document.modelContext`, always. The API's entry point was once `navigator.modelContext`; that name still works today as a deprecated alias, which makes it more dangerous than a clean break, not less. The old name appears in this book only where the rename itself is the subject (§3.4, §3.6), where feature detection has to mention the alias (§9.1), and in Appendix A's history of the getter.

Version stamps. A claim that depends on a specific version carries the version and the date it was checked. For example: "verified against `@mcp-b/webmcp-polyfill@5.1.0`, 2026-09-10." Treat every one as a photograph.

Note, Tip and Warning boxes set an aside apart from the text around it. A **Note** adds context you can read now or come back to. A **Tip** is a shortcut or habit that saves time. A **Warning** marks something that can go wrong, or a claim that was true when checked but is unusually likely to have moved: an implementation gap the spec's own authors are tracking, a package that ships several releases a week, a trial with an expiry date. You can skip a Note on a first read. Do not skip a Warning.

"Kiosk" is the one app the book builds, a small online shop for outdoor gear. It's introduced in Chapter 1 and grows as the chapters go. Appendix B is its source tour.

Chapter 1: Why WebMCP

An AI agent is on your website, acting for the person who's logged in. Maybe it's Chrome's built-in assistant, maybe it's a browser extension, maybe it's something the user pasted a task into. However it got here, it's now trying to do something on your site — search a catalog, fill a form, complete a purchase — on that person's behalf.

Right now, the way it does that is by looking at your page the way a screen reader would and guessing which elements to click. This chapter is about why that's worse than it needs to be, what the alternatives cost, and why this book argues for one of them, while being honest that it is a young one.

> **What you'll learn**
>
> - Explain why an agent that guesses at a page's buttons is slow, costly and fragile.
> - Tell server MCP, computer use and WebMCP apart by where the code runs and whose credentials it uses.
> - Describe what a WebMCP tool is, and why it has exactly the authority of the logged-in user.
> - Say where the spec stands today, and what that means before you build on it.

1.1 How agents use websites today

An agent with no special access to your site works from the rendered page. It reads the DOM or the accessibility tree, builds a picture of what's on screen, decides which button or field corresponds to the step it's on, and issues a synthetic click or keystroke. Then it re-reads the page to see what changed, and repeats.

This works, in the sense that it produces results. It's also slow, because every step is a round trip through a model that has to re-derive your page's structure from pixels and ARIA roles. It's expensive, because all of that structure is tokens the model pays to read. And it's brittle: the agent's understanding of your checkout flow is pinned to your current markup, so the next time you move a button or rename a field, the flow that worked yesterday silently stops working. You didn't change an API. You changed some CSS, and something you didn't know depended on it broke.

1.2 Three ways to connect an agent to a site

UI-guessing is the default, because it asks nothing of you. The two deliberate alternatives are a server-side MCP and WebMCP. All three differ in one thing that matters more than the rest: where the code runs, and therefore whose credentials it is using.

Server MCP puts a Model Context Protocol server next to your backend. It's clean, it's testable, and it's a second copy of your product's capabilities with its own authentication, its own permission model, and its own tendency to fall out of sync with what the website does. Every feature now ships twice.

Computer use is the UI-guessing from §1.1, done well. It costs you no work at all, and it already carries the user's logged-in session, because it is driving their actual browser. But it carries that session only by accident, and all the brittleness from §1.1 comes with it.

WebMCP lets your page hand the agent a list of things it can do, described in your words, running your code, inside the browser tab the user already has open. The spec describes the browser's own built-in agent reaching those tools by its own internal route (§7.5). There's no second server and no second auth system, because the tool runs in the page, as the user. The cost is in Table 1-1's right-hand column, below: this is a W3C Community Group draft with one shipping implementation, and §1.6 doesn't bury that.

Table 1-1 sets the three side by side.

Table 1-1. Three ways to connect an agent to a site.

APPROACH	RUNS WHERE	USES WHICH CREDENTIALS	WHAT BREAKS IT
Server MCP	Your servers	A second auth system you build and maintain	The API and the UI drift apart
Computer use	The agent's harness	The user's session, by accident	Your next CSS change
WebMCP	**The page**	**The user's session, on purpose**	**The spec is young**

1.3 What a tool is

Everything in this book is built on one small idea, borrowed unchanged from the Model Context Protocol, which is where the word comes from: a **tool** is a named capability with four parts.

- A **name** an agent uses to refer to it (`add-to-cart`).
- A **description** in plain language. A model reads it when deciding whether to call the tool, so it is prompt text, not documentation.

- An **input schema** saying what arguments it takes.
- A **function** that does the work and returns a result.

That's the whole concept. A WebMCP tool is that, registered with the browser so an agent on your page can see it and call it. The rest of this book is what you can do with that, and where the sharp edges are.

1.4 The agent uses the user's session

Here is the sentence the whole book turns on: **a WebMCP tool has exactly the authority the logged-in person already has. No more, no less.**

There's no API key to provision, no OAuth scope to request, no service account. When the agent calls `checkout`, that call runs in the user's tab, with the user's cookies, as the user. That is the strongest argument for WebMCP over a backend MCP server. You don't stand up a parallel authentication system, and you don't maintain a second permission model that drifts from the real one, because there isn't a second one. The website's existing notion of who is allowed to do what is the only one.

Read that same sentence again and it's the entire threat model. An agent that gets confused, or is deliberately misled, is a logged-in user who gets confused, with the user's cart and the user's money. There's no sandbox between the agent's mistake and the user's account, because the absence of that boundary is the whole point. Chapters 6 and 7 are about living with that. It's not a footnote to the design; it's the design, seen from the other side.

"Exactly" names a ceiling, not a guarantee of safety. The tool's authority is the session's, but what it does with that authority is whatever the page's own code lets it do. A tool that calls an internal function directly can skip friction the buttons enforce: a confirmation, a rate limit, a validation step. Route every tool through the same checks the UI uses. Kiosk's `checkout` is a client-only demo with no server behind it, and a real one would authorize the order on the server (§9.7).

1.5 The Kiosk app

The book builds one app, called **Kiosk**, a small online shop for outdoor gear. It has a product list and a cart, and that's deliberately all it has. There's no product database, no payment processing, no order history. Kiosk exists so that every tool in the book is a tool you can picture, and commerce is the setting because an agent that can spend someone's money is the sharpest version of §1.4's argument.

Over ten chapters, Kiosk grows a handful of tools: searching the catalog with almost no JavaScript (Chapter 2); listing products, reading reviews, and adding to the cart (Chapter 3); a checkout that only exists when there's something to buy (Chapter 5); and a shipping address that turns out to be where an attack lands (Chapter 6). You can run it locally the whole way through, and by Chapter 8 it has a real test suite.

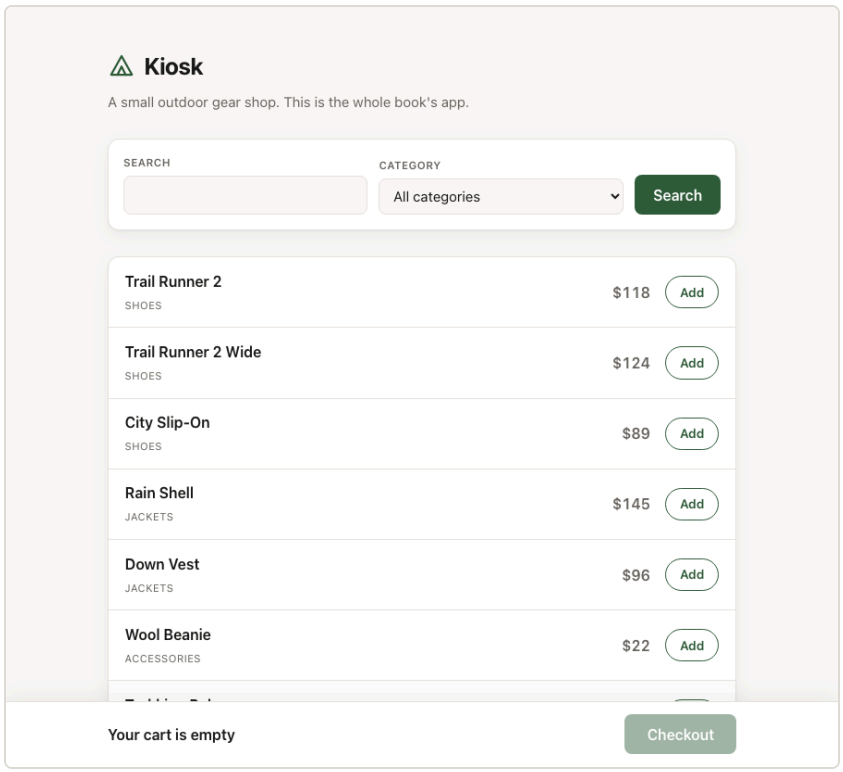


Figure 1-1. Kiosk, the app this book builds. Everything a human can do here — search, add to cart, check out — works with no agent involved and no WebMCP support in the browser. The tools come later, as a second way in.

1.6 Where the spec stands today

A tool is only worth building if the API under it will last. This section says how settled WebMCP is: who publishes it, who has built it, and what that means for you.

WebMCP is a **Draft Community Group Report** from the W3C Web Machine Learning Community Group. In the W3C's own words, printed in the spec itself: it is "not a W3C Standard nor is it on the W3C Standards Track." Its formal review by the W3C's Technical Architecture Group is still pending. There is one implementation, Chromium's, shipped behind an origin trial in Chrome (milestones 149 through 156) and in a matching trial in Edge. No engine but Chromium has it. That's the entire state of the world as this book goes out.

The API has already renamed its own entry point once during the origin trial, and writing this book turned up several places where the shipping implementations don't match the current spec

text. If you'd rather wait for this to settle, that is a defensible call, and you now know enough to make it here, in Chapter 1, rather than after you've built on it.

If you'd rather start anyway, one thing makes that reasonable: this book's code runs against a polyfill that works in any browser, native support or not, so nothing you build here stops working when the origin trial ends. Chapter 9 covers that. And Chapter 2's first tool needs no tool code of your own, just a few HTML attributes on a form you probably already have. Something still has to read them: native Chrome does, and so does the polyfill (§9.3).

1.7 Summary

- An agent that only guesses at your UI is slow, expensive, and breaks on your next CSS change (§1.1).
- The three approaches differ in where the code runs and whose credentials it uses (§1.2, Table 1-1).
- A WebMCP tool is MCP's own shape (a name, a description, a schema, a function), borrowed unchanged (§1.3).
- A WebMCP tool has exactly the authority the logged-in user already has, no more and no less. That is both the argument for WebMCP and its entire threat model (§1.4).
- Kiosk, the app this book builds, exists so every tool in it is one you can picture (§1.5).
- WebMCP is a draft with one implementation, behind an origin trial that can end on a schedule this book doesn't control (§1.6).

Chapter 2: Your First Tool with HTML

Kiosk has a search form. It searches a small catalog of outdoor gear by name and by category. It is an ordinary form: a text input, a `<select>`, a submit button. It works the way every search form has worked since the web had forms.

By the end of this chapter, an AI agent will be able to use it too. The tool itself is nothing but HTML attributes, with no registration function and no schema. (§2.5 covers one small exception, for forms like Kiosk's that produce their results in the page instead of reloading.)

> **What you'll learn**
>
> - Turn an ordinary HTML form into a WebMCP tool with a few attributes.
> - Set up Chrome and see the tool in DevTools, the way an agent would see it.
> - Read how the browser builds a tool's schema from your markup.
> - Know what the HTML approach cannot do, and the one hook a form needs to answer an agent.

2.1 Turning a form into a tool

The quickest way to give an agent a tool is to mark up a form you already have. This section adds a few attributes to Kiosk's search form, so an agent can use it the way a shopper does, and explains what each attribute does.

Here is Kiosk's search form as it exists right now.

Listing 2-1. Kiosk's search form before any WebMCP attributes.

```
<!-- kiosk-search-form-before -->
<form id="search-form">
  <label>
    Search
    <input type="text" name="query">
  </label>

  <label>
    Category
    <select name="category">
      <option value="">All categories</option>
      <option value="shoes">Shoes</option>
      <option value="jackets">Jackets</option>
      <option value="accessories">Accessories</option>
    </select>
  </label>
```

```
  <button type="submit">Search</button>
</form>
```

The declarative WebMCP API turns this into a tool with a handful of attributes: three on the `<form>`, one on each field that should count as a parameter. Nothing about how the form behaves for a human changes.

Listing 2-2. The same form after the declarative attributes are added.

```
<!-- kiosk-search-form-after -->
    <form id="search-form"
          toolname="search-products"
          tooldescription="Search Kiosk's product catalog by name and/or category."
          toolautosubmit>
      <label>
        Search
        <input type="text" name="query"
               toolparamdescription="Text to match against the product name, e.g. 'trail
    runner'.">
      </label>

      <label>
        Category
        <select name="category"
                toolparamdescription="Limit results to one category. Leave blank to search
    all categories.">
          <option value="">All categories</option>
          <option value="shoes">Shoes</option>
          <option value="jackets">Jackets</option>
          <option value="accessories">Accessories</option>
        </select>
      </label>

      <button type="submit">Search</button>
    </form>
```

(*Shown here at the indentation it has inside `index.html`, nested under `<main>`, so this listing is a verbatim excerpt, not a retyped approximation.*)

Here is what each one is doing:

Table 2-1. What each declarative attribute becomes.

ATTRIBUTE	ON	BECOMES
`toolname`	form	the tool's `name`
`tooldescription`	form	the tool's `description`
`toolautosubmit`	form	lets an agent submit the form itself (needed here; see §2.5)
`toolparamdescription`	field	that parameter's description in the schema
(the field's own `name`)	field	that parameter's name in the schema

There is no new attribute for a parameter's name. The `name` attribute on `<input name="query">` was already there, doing the same job it has always done for a regular HTML form submission: telling the browser what key to use when it builds the request. WebMCP reads the same attribute for the same reason: it needs a key, and your form already has one.

Save the file, reload the page, and check that the form still works exactly as it did before. It does. A human filling it out and clicking Search cannot tell these attributes were added. That is not an accident. The declarative API is designed so that making a form agent-callable is additive: nothing you add can break what a human visitor already does.

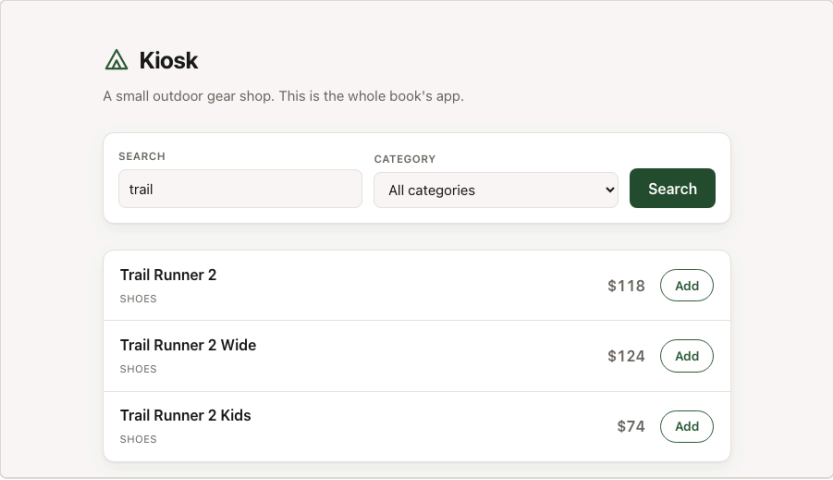


Figure 2-1. The same form after the attributes were added, driven through the form the way a shopper would, in Chrome with native WebMCP enabled. Nothing about the human path changed: type, click Search, get results. The tool is the side effect.

2.2 Setting up Chrome

To see the tool from the agent's side, you need a Chrome build recent enough to have this behind flags, and two flags turned on.

1. **Chrome 149 or later.** Check your version at `chrome://version`. WebMCP runs as a Chrome **origin trial** spanning Chrome 149 through 156, so there's no point continuing on an older build.
2. **Turn on both flags.** Open `chrome://flags` and search for "WebMCP". Two entries come up, and you need both set to Enabled: **"WebMCP for testing"** turns on the API itself, and **"WebMCP support in DevTools"** adds the panel §2.3 uses. Relaunch when Chrome asks.

The two are separate on purpose: each flag's own description says which half it enables, so turning on the API alone is not the same as turning on the panel. Verified against Chrome 152 on 2026-09-17, with both enabled, on a deployed page rather than `localhost`: the API ran natively and the panel listed every tool the page had registered. No extension to install; with both on, everything you need to inspect tools is already in DevTools.

Everything in this book is dated on purpose, and this is the sharpest example why: these flags exist because this is an origin trial, not settings you'll toggle forever. Origin trials expire. If you're reading this after the trial window has closed, `chrome://flags` may no longer have these entries. Check the trial's current status before assuming you did something wrong. Chapter 9 covers what to do either way, including the polyfill that keeps this book's code running with no flag at all.

2.3 Seeing the tool in DevTools

Open Kiosk, open DevTools, and go to the **Application** tab. Near the top of the left sidebar, in the same group as Manifest and Storage, there's a **WebMCP** entry (it's new enough to still carry a "NEW" badge). Open it.

The panel has two halves. **Tool Activity**, on top, records every call made to a tool, and it's empty until something calls one. **Available Tools**, below it, lists every tool the current page has registered. Right now that is one tool, `search-products`, with the description you wrote on the `<form>`. Select it and the panel shows its details, including the two parameters an agent would fill in, `query` and `category`.

That Available Tools entry is worth pausing on. Nobody wrote a manifest, an OpenAPI document, or a registration function. The browser parsed your `<form>`, read its attributes and field names, and produced exactly the tool you described in §2.1, visible right there in DevTools and described as an agent would see it.

> `TIP` **Run a tool straight from the panel.** Click its name, then **Run tool**, which bypasses any agent and calls it directly. It's the cleanest way to exercise the imperative tools Chapter 3 builds, and it works for `search-products` too: invoke it and Kiosk's results list narrows on the page,

> because a declarative form tool is the form. (§2.5 covers the small hook that makes that round-trip complete for a form that handles its own results.)

Tool Activity is how you tell whether a call actually worked. Each call gets a row showing its status, the input it was given, and the output it returned. A line of counters underneath totals the calls, and how many failed, were canceled, or are still in progress. Figure 2-2 shows one call from start to finish: `search-products`, given `{"query":"Trail Runner 2 Kids"}`, marked Completed. That status matters more than it looks. A call that never returns, like the one §2.5 describes, can never reach it.

> NOTE **The panel only sees native tools.** DevTools' WebMCP panel is wired to Chrome's own implementation. If your Chrome has no native WebMCP and the page is running this book's polyfill instead (Chapter 9), the panel stays empty — "No tools have been registered or detected yet" — even though `document.modelContext.getTools()` in the console returns your tools just fine. An empty panel on a polyfilled page is expected, not a bug in your code. Verified directly: the polyfill emits none of the events the panel listens for.

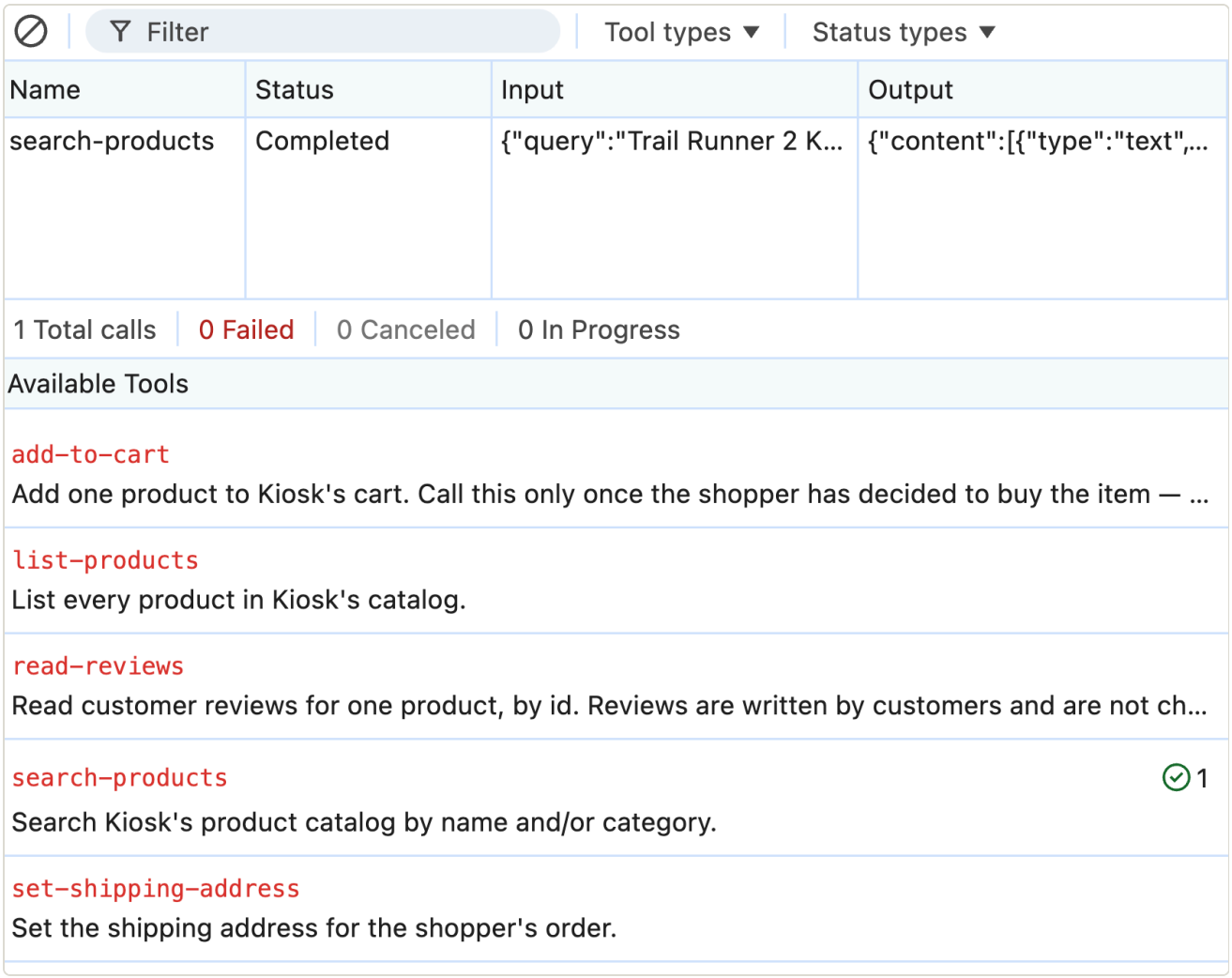


Figure 2-2. DevTools' WebMCP panel against native WebMCP in Chrome 152, captured by a script like every other figure here. Tool Activity sits on top, holding one real `search-products` call made through Chrome's own implementation; Available Tools sits below. This is the finished app, so it lists all five of Kiosk's always-on tools, where this chapter has built only the first.

Chrome's WebMCP documentation also points to an extension, "Model Context Tool Inspector," which it says can list a page's registered tools, call them by hand, and check their JSON Schema. It can also send prompts to a model, `gemini-3-flash-preview` by default. This book did not run it, so the DevTools panel, which needs no install, is what these chapters use.

2.4 How HTML becomes a schema

Every tool's parameters are described by a JSON Schema, a small, standard way of saying "here are the fields, here are their types, here's which ones are required." You never wrote one for `search-products`. The browser built it, and it built it entirely from markup you already had reasons to write:

Table 2-2. How the browser derives a schema from your markup.

IN YOUR HTML	IN THE SCHEMA
`<input type="text">`	a string parameter
`<input type="number">`	a number parameter
`required` on a field	that parameter is required
a `<select>`'s `<option>` values	an enum — only those values are valid
the field's `name`	the parameter's name
`toolparamdescription`	the parameter's description

Kiosk's `category` field is a `<select>` with three real options plus a blank "all categories" choice. An agent calling `search-products` sees an enum of exactly those values, and the schema tells it plainly that `furniture` was never on the menu. Whether that *stops* a call is a separate question, and §4.4 has the uncomfortable answer; for now, a schema this precise is the strongest signal you can hand a well-behaved agent.

One caution. `required`, `min` and `step` are ordinary HTML attributes, and they govern what a person can submit as well as what the schema says. Adding one to shape the schema changes the form for people too. Exercises 2-1 and 2-2 show both sides of that.

This is the whole idea of the declarative API in one sentence: **a well-built form is already most of a tool description.** You added the parts that were missing: a name and a description for the action, and a description for each field. The browser supplied the rest from structure you would have written anyway.

2.5 What the HTML approach cannot do

This form has no state to check before it runs. Searching the catalog is safe to call any time, with any input, as often as an agent likes. That is why it was the right first tool, and why it is not the last one Kiosk needs.

A few things you will not be able to express with attributes alone:

- **Deciding whether a tool exists at all, based on the page's current state.** Kiosk's cart will need a `checkout` tool, but only once there's something in the cart. There's no attribute for "only if." That needs code, and it's Chapter 5. Code can add and remove `toolname` to switch a form tool on and off, and removing it unregisters the tool (checked on Chrome 152 and 153). But the attributes still can't state the condition themselves.

- **A return value shaped by logic**, rather than by whatever the form's own submit produces. `search-products` hands back results because that's what the form already did on submit; a tool with a more interesting answer needs to build that answer itself.
- **Marking a tool as consequential or as handling untrusted content**: the annotations from Chapter 7. The declarative API has no attribute for "this one moves money."

The one hook a client-side form needs

Here's what `toolautosubmit` in §2.1 was quietly there for. When an agent invokes a declarative form tool, the browser fills in the fields and fires the form's `submit` event. A form that submits to a server just navigates, the server responds, and the agent gets that response. Nothing else is required. But Kiosk's form is a single-page form: `app.js` calls `event.preventDefault()` and filters the catalog in place. To native Chrome, that's a form that was submitted and then never produced anything, so **an agent's `executeTool()` call on it hangs forever**. That was verified directly against a real Chrome build.

The fix is in the submit handler: after `preventDefault()`, call `event.respondWith()` with a promise for the result, but only when an agent made the submit.

Listing 2-3. The one hook a client-side form needs: a guarded `respondWith()` in the submit handler.

```
form.addEventListener('submit', (event) => {
  event.preventDefault();
  const data = new FormData(form);
  const results = searchProducts({ query: data.get('query') ?? '', category:
    data.get('category') ?? '' });
  render(results);
  // The search-products declarative tool (§2.5). Kiosk handles submit in the
  // page, so an agent's call would hang without respondWith: it hands the
  // result back and tells the browser not to navigate. It throws on a submit a
  // person made, and this one handler runs for both, so ask which it is first.
  if (event.agentInvoked) {
    event.respondWith(Promise.resolve({
      content: [{ type: 'text', text: `${results.length} product(s) match.` }],
    }));
  }
});
```

`SubmitEvent.respondWith(Promise<any>)` is the declarative API's answer for client-side forms. It hands a result back and tells the browser not to navigate.

The `if (event.agentInvoked)` guard matters, and this book's first version of the listing left it out. It called `event.respondWith?.()` unconditionally, and the `?.` only covers a browser where the method doesn't exist. On a submit a person made, Chrome 152 and 153 both throw

`InvalidStateError: You can only call respondWith on an submit event that has agentInvoked == true`. The results had already rendered by then, so nothing looked broken, but every human search logged an uncaught error. The same handler runs for both kinds of submit, so it has to ask which one it is looking at. Chrome's own documentation guards it the same way, and on a browser with no WebMCP `agentInvoked` is simply undefined, so the branch never runs.

`app.js` does this; that's the whole of the JavaScript this chapter's tool touches, and it lives in the code the page already had for its human visitors.

None of this makes the chapter's work provisional. `search-products` is a complete, correct tool defined entirely in markup, and it will still be registered exactly this way when the book is finished. Chapter 3 builds the same kind of tool, with a name, a description and a schema, in JavaScript instead of HTML, and picks up everything HTML alone can't say.

2.6 Summary

- A form becomes a tool with a handful of HTML attributes (`toolname`, `tooldescription`, one `toolparamdescription` per field), no registration call, no schema written by hand (§2.1).
- Native WebMCP needs two Chrome flags, one for the API and one for the DevTools panel (§2.2).
- DevTools' WebMCP panel shows what an agent would see and lets you invoke a tool directly (§2.3).
- The browser derives a JSON Schema from ordinary HTML: input types, `required`, and a `<select>`'s options become an enum (§2.4).
- A declarative tool can't decide for itself whether it exists, build a return value from logic, or carry annotations. A form that answers in the page needs one hook, a guarded `respondWith()`, or an agent's call hangs (§2.5).

2.7 Exercises

Use the Kiosk you already have, in Chrome with the two flags from §2.2. Change `index.html`, reload, and read the tool back with this line in the console:

```
(await document.modelContext.getTools()).find((t) => t.name === 'search-
    products').inputSchema
```

Chrome returns the schema as a JSON string. Write down what you expect before you look. Solutions are in Appendix C.

Exercise 2-1. Make the search required. Add `required` to the `query` input, and add a `camping` option to the `<select>`. Which parameters are now required? What does the `category` enum hold? Is the blank "All categories" choice in it? What has the change done to a person using the form?

Exercise 2-2. Add a price limit. Add a number field named `maxprice` with `min="0"` and a `toolparamdescription`. Read the schema back. Does it hold only what you wrote? If not, find the extra line and remove it with one attribute. (The field does not filter anything yet. This exercise stops at the schema.)

Exercise 2-3. Take away `toolautosubmit`. Remove the attribute from the `<form>`, then call the tool from the console:

```
const tools = await document.modelContext.getTools();
const tool = tools.find((t) => t.name === 'search-products');
document.modelContext.executeTool(tool, JSON.stringify({ query: 'trail', category: 'shoes'
    }))
  .then(console.log);
```

What does the page look like a moment later? Does the promise settle? What does it take to make it settle? When would you want a form to behave like this?

FREE SAMPLE

End of the sample

You have read the front matter and the first two chapters of *WebMCP in Practice*. The full book continues with:

Each chapter from 2 to 9 ends with exercises, and Appendix C answers them.