MX-Handbook

Designing Web Interfaces for AI Agents

Tom Cranstoun

2026-01-22

Contents

Preface **3**

Chapter 1: MX-Bible **9**

Chapter 2: How AI Reads (vs. How Humans Read) **13**

Chapter 3: Guiding Principles **19**

Chapter 4: Content Architecture for Machines **29**

Chapter 5: Metadata That Works **41**

Chapter 6: Navigation and Discovery **53**

Chapter 7: The JavaScript Challenge **67**

Chapter 8: Testing with AI Agents **81**

Chapter 9: Common Anti-Patterns **95**

Chapter 10: Implementation Roadmap **113**

Chapter 11: Business Imperative **129**

The End **137**

MX-Handbook

Designing Web Interfaces for AI Agents

Tom Cranstoun

January 2026

EARLY DRAFT

Commercial Work - Copyright Tom Cranstoun (c) 2026 - Do Not Distribute

This manuscript is in preparation for publication.

Confidential and proprietary. All rights reserved.

Preface

I didn't set out to write about AI agents. I set out to book a holiday.

It was late 2024, and I was comparing tour operators for a trip through Southeast Asia. I'd delegated the research to an AI assistant, expecting it to save me hours of clicking through brochures. Instead, it gave me confident but wrong advice about which company had the better itinerary.

The agent had looked at one tour operator's paginated day-by-day breakdown for a 14-day tour, seen only Day 1, and concluded that was the entire trip. The competitor's single-page itinerary was readable in full. Based on this, my assistant recommended the wrong company.

I caught the error. That mistake led me to investigate why AI agents fail - and what that means for businesses.

The solution became clear: Machine Experience (MX). MX is the practice of adding metadata and instructions to websites so AI agents don't have to think. Instead of agents inferring meaning from visual design, MX provides explicit structure through semantic HTML, Schema.org structured data, and clear state management. When agents must guess based on incomplete context, they hallucinate—generating confident but incorrect answers. MX prevents hallucination by making all context explicitly present in machine-readable formats.

The Pattern Emerges

The same design choices that confused my AI assistant also confused users with disabilities. People who rely on keyboards rather than mice. Those using screen readers to navigate. Voice control users who cannot make precise movements.

We'd built a web that worked brilliantly for one specific type of user: someone with good vision, working on a desktop, with focused attention and plenty of time. Everyone else had been struggling quietly for many years. Now AI agents were struggling loudly, and there was finally commercial pressure to fix the problems.

The Market Moved Fast

When I started writing, AI agents accessing websites felt like something to monitor and prepare for gradually. By early 2026, the landscape had fundamentally transformed.

January 2026: Four major platforms launched agent systems within eight days. Amazon's Alexa+ (January 5th) for voice-initiated purchasing. Microsoft's Copilot Checkout (January 8th) for transactions within the AI assistant. Google's Universal Commerce Protocol (January 11th) backed by Target and Walmart. Anthropic's Claude Cowork (January 12th) for autonomous workflows.

This isn't a distant future. Agent-mediated commerce is happening now, and businesses that aren't ready are being excluded from high-conversion transactions.

Understanding the Five Types of AI Agents

Before we can fix websites for AI agents, we need to understand what they actually are. Not all agents work the same way, and these technical differences matter.

Server-Side Agents (ChatGPT, Claude) run on remote servers and fetch your website as raw text. They cannot execute JavaScript or render CSS. They see HTML structure and text content, but miss everything that loads dynamically. If your product catalogue loads via JavaScript, server-side agents see an empty page.

In-Browser Agents (Microsoft Copilot, Browser Extensions) run within web browsers and have full access to rendered pages. They can execute JavaScript and access the DOM after it updates. They see dynamic content but miss visual hierarchy from CSS. If your form submission shows a toast notification that vanishes in 3 seconds, in-browser agents might miss it.

Browser Automation Agents (Perplexity, Playwright-based) control full browsers programmatically and can take screenshots. They can see layout, colour, and size relationships through computer vision. They can see your "Add to Cart" button is red and prominent, but still need proper HTML button elements to click reliably.

Local Agents (Ollama, On-Device LLMs) run on users' personal computers with limited resources. They use smaller models with limited context windows. If your page serves 10,000 words of marketing copy before getting to product details, local agents may never reach the important content.

Agentic Operating Systems (Anthropic Cowork) orchestrate multiple agents working in parallel. They combine capabilities of other agent types - managing files, executing browser automation, processing server-side content - whilst coordinating complex multi-step workflows. If your checkout works for browser agents but breaks for server-side agents, agentic operating systems cannot reliably complete purchases.

Each type has different capabilities. But all five need the same fundamental patterns: semantic HTML, explicit state management, and structured data. When you fix your website for one agent type, you improve it for all five.

What this book is

Ten focused chapters on implementation. How AI agents read HTML. What patterns work. How to test. What breaks. You'll find code examples, real comparisons, and practical tests you can run today.

Chapter 1 explains why this matters (quickly) and gives you an immediate test for your site.

Chapters 2-3 cover the technical foundation - how AI agents parse content and the principles that make content readable.

Chapters 4-7 provide patterns for common content types: articles, products, navigation, JavaScript-heavy interfaces.

Chapter 8 shows you how to test properly, because assumptions about AI compatibility are usually wrong.

Chapter 9 covers the mistakes I see most often.

Chapter 10 gives you a roadmap for phased implementation, because you probably can't fix everything immediately.

Chapter 11 covers the business imperative - why this matters commercially, ROI calculations, and strategic decisions about protocol integration.

What this book isn't

This isn't a comprehensive exploration of AI technology, business models, or strategic planning. For that, read "The MX Bible" (the complete technical reference). This book assumes you're convinced and ready to implement.

This isn't about abandoning modern frameworks or returning to 1990s HTML. The patterns here work with React, Vue, complex SPAs, and JavaScript-heavy interfaces. You're fixing specific problems, not rebuilding from scratch.

This isn't theoretical. Every pattern comes from real client work - sites that were invisible to AI agents and needed fixing quickly.

The two-book ecosystem

This practical guide (The MX Handbook) is part of a two-book series, all focused on Machine Experience (MX) implementation:

"The MX Handbook" (this book) - Platform-specific MX implementations across major CMS systems, content strategies, and detailed testing frameworks for practitioners managing larger deployments. Practical MX patterns for developers and UX designers who need to make sites agent-compatible quickly.

"The MX Bible" - Comprehensive technical reference covering MX principles, business models, organisational change, legal implications, and strategic positioning for architects and consultants implementing Machine Experience at scale.

Both books share continuously updated appendices at https://allabout.network/invisible-users/ including implementation examples, pattern libraries, and protocol updates.

Read either book standalone, or use them together: the Handbook for immediate implementation guidance, the Bible for comprehensive strategic context.

How to use this book

This book works from both ends. Business leaders and technical leads should start at the back (Chapters 11 and 10) to understand ROI and planning, then work forwards through implementation details. Developers should start at the front (Chapters 2-4) for technical patterns, then work backwards to understand business context.

Critical: Developers must not skip the business chapter. You'll be asked to justify this work. Chapter 11 gives you the numbers and strategic context you need.

If you're a frontend developer: Start with Chapter 2 (how AI reads) then jump to Chapter 4 (patterns). Reference Chapter 8 for testing as you implement. Read Chapter 11 before presenting implementation plans to leadership.

If you're a UX designer: Read Chapter 3 (principles) to understand the constraints, then Chapter 7 (JavaScript) for dynamic interfaces. Chapter 11 explains why this affects conversion rates.

If you're a QA engineer: Go straight to Chapter 8 (testing methodology) then work backwards through the patterns to understand what you're testing for. Chapter 11 shows why these tests matter commercially.

If you're a technical lead or business leader: Start at the back. Read Chapter 11 (business imperative) for ROI justification and competitive urgency, then Chapter 10 (roadmap) for implementation planning. Work forwards through Chapters 2-9 as needed for technical depth.

What you'll gain

By the end of this book, you'll be able to:

- Test any page for AI readability in minutes
- Spot the patterns that break AI comprehension
- Fix common problems without major refactoring
- Validate that your fixes actually work
- Plan implementation across your site systematically

The techniques here make sites better for screen reader users, mobile visitors, and anyone who needs to find information quickly. AI agents are demanding, but they're demanding the same things accessibility guidelines have been recommending for many years. Fix it for one audience, help three others.

The Convergence Principle

Here's the key insight: patterns that help AI agents also help humans with disabilities.

Semantic HTML that helps screen readers also helps server-side agents. Explicit state management that helps keyboard users also helps browser-based agents. Structured data that helps voice assistants also helps all types of agents extract accurate information.

You're not choosing between audiences. You're not trading off business goals against accessibility compliance. One implementation serves everyone: users with disabilities, search engines, and AI agents.

The work compounds across multiple channels. The same patterns that improve AI agent compatibility also improve search engine rankings, accessibility compliance, and user experience for people in non-ideal conditions.

Why This Matters Now

In January 2025, Tailwind CSS laid off 75% of its team after traffic-dependent revenue collapsed. When AI agents started answering developer questions without sending traffic to documentation sites, companies relying on pageviews found their monetisation broken.

This isn't theoretical. Companies are failing right now because their business models assume human browsing behaviour that no longer dominates discovery.

Acknowledgements

This book exists because problems became visible when I looked closely at failures I'd typically have ignored. I'm grateful to everyone who has written about web accessibility over many years; their work laid the foundations on which this analysis builds.

Thank you to colleagues, clients, and collaborators who reviewed early drafts and asked uncomfortable questions.

A note on timing

This field moves fast. By the time you read this, new AI agents will have launched, new protocols will be competing, and the landscape will have shifted. The patterns in this book focus on fundamentals: semantic HTML, explicit state, structured data. Those foundations work regardless of which specific agents dominate.

For updates and new patterns, visit https://allabout.network/invisible-users/news.html.

Now let's make your site visible.

York, England

Q1 2026

Chapter 1: MX-Bible

A visitor you'll never see

Right now, while you're reading this, an AI agent is probably visiting your website. You won't see it in your analytics as a conversion. It won't fill out your contact form. It won't click your call-to-action button. But it might be deciding whether your company gets recommended to its user—or your competitor does.

These visitors don't behave like humans. They don't admire your hero image, they don't notice your carefully crafted colour scheme, and they definitely don't appreciate that clever JavaScript animation you spent three days perfecting. They're reading your site in ways you never designed for, making decisions based on information you didn't realize you were publishing.

Welcome to the era of invisible users.

A note on perspective: This book isn't a criticism of AI technology. I genuinely admire what these systems have achieved - the ability to generate coherent responses, understand context, and complete complex tasks across countless domains represents remarkable engineering. My focus isn't on pointing out AI failures or edge cases. Instead, this book explores how better-structured websites create better outcomes for everyone. When we provide well-crafted inputs (semantic HTML, explicit metadata, clear state management) and AI creators implement appropriate guardrails, the results improve dramatically. Hallucinations decrease. Citations become accurate. Commerce transactions complete successfully. This is about collaboration, not criticism.

Who are they?

When I talk about AI agents, I'm not describing some distant future. I'm talking about systems working right now:

ChatGPT, Claude, and other conversational AI that answer questions by searching and reading web content. When someone asks "Which accountancy firm in Manchester specializes in construction companies?" these systems are reading multiple websites and choosing which ones to recommend.

Voice assistants that need to extract specific information from your site. "Alexa, what are the opening hours for the museum?" requires structured, parseable content to answer correctly.

Browser automation agents that can interact with websites on behalf of users. These can fill forms, make purchases, and compare prices across multiple sites without human intervention.

Search engine crawlers that increasingly use AI to understand context and intent. Google's not just counting keywords anymore—it's trying to understand what your page actually means.

Specialized agents for travel booking, price comparison, research assistance, and hundreds of other tasks. Each one reading your site differently, looking for different signals.

The critical point: these aren't all the same type of visitor. Some fetch your raw HTML and parse it without ever executing JavaScript. Others run full browsers and can see your site as humans do. Some take screenshots and use vision models to extract information. Many use combinations of these approaches.

Why this matters (the economic bit)

Let me give you a concrete example from my consulting work.

A client—a global automotive company—spent £2 million on a website redesign. Beautiful work. Won design awards. But when I tested it with AI agents, they couldn't find basic information like dealer locations or service pricing. The data was there, hidden in JavaScript state and visual layouts, but structured in ways that made it invisible to parsers.

Meanwhile, their smaller competitors had basic sites with proper semantic HTML. When someone asked an AI "Where can I get my Land Rover serviced near Birmingham?" the AI confidently recommended the competitors. Not because their sites looked better—they didn't—but because their content was readable.

My client was losing enquiries to companies they should have been beating easily. The invisible users were choosing the competitors.

This pattern repeats across industries:

A law firm with a gorgeous site couldn't get mentioned when AI was asked for "employment lawyers in London" because their practice areas were rendered client-side with no semantic structure.

An e-commerce site lost product comparison visibility because their specifications lived in a JavaScript object, not in marked-up HTML.

A SaaS company's documentation—beautiful to look at—was useless to AI agents trying to help developers integrate their API.

The shift in traffic sources

For the past 25 years, we've optimized for two sources of traffic: search engines and direct visits. We've built entire industries around SEO and paid search. We've A/B tested our way to better conversion rates for human visitors.

Now there's a third source that's growing rapidly: **AI recommendations**.

When someone asks an AI for help, they're not clicking through ten blue links and comparing options. They're getting one answer, maybe three recommendations. If you're not in that answer, you don't exist.

This isn't speculative. The data is already showing the shift:

- Conversational AI queries are growing exponentially
- Traditional search traffic is plateauing for many sectors
- Voice search (which relies heavily on AI parsing) accounts for increasing mobile queries
- Browser automation is handling routine tasks like price comparison and appointment booking

The visitors you see in your analytics are becoming less representative of the decisions being made about your business.

The visibility paradox

Here's the strange part: making your site work for AI often makes it better for humans too.

When you structure content semantically—proper headings, meaningful lists, clear relationships between elements—you're not just helping AI parsers. You're helping:

- Screen reader users who navigate by heading structure
- Mobile users who need scannable content
- Busy users who want to find information quickly
- Search engines that reward clarity
- Your own development team who maintain the site

The techniques that make content AI-readable aren't tricks or hacks. They're often just good web development practices that we've been ignoring because we could get away with it. Humans are forgiving. They'll click around, use browser search, even view page source if desperate. AI agents won't. They'll move on to a site that makes information accessible.

What this book will do

I'm not going to tell you to rebuild your site from scratch. I'm not going to insist you abandon modern frameworks or stop using JavaScript. I'm going to show you how to make your existing site work for both human and AI visitors, because the two audiences have more overlap than you might think.

We'll look at:

- How AI agents actually read and interpret web content
- Practical patterns that work for both audiences
- Simple tests you can run today to see how AI sees your site
- Common mistakes that make you invisible to AI
- Real examples of sites doing this well (and badly)

Most importantly, I'll show you that this isn't another item on an endless checklist of web best practices. This is about remaining visible to users who are increasingly making decisions about your business without you knowing.

The morning-after test

Steve Krug taught us to watch humans use our sites to find usability problems. I'm suggesting something simpler for AI readability: feed your page to ChatGPT or Claude and ask it questions.

Try this right now:

1. View source on your most important page
2. Copy the HTML (not the rendered version, the actual source)
3. Paste it into ChatGPT or Claude
4. Ask: "What is this page about? What are the main services or products offered? What actions can I take?"

If the AI can't answer accurately, neither can the invisible users visiting your site.

I tested this with a corporate client's homepage. The AI's response: "This appears to be a website but I cannot determine what the company does or what services they offer. I can see some navigation elements and what might be placeholder text."

The actual rendered page was clear to humans. But the content was loaded via JavaScript, the headings were div elements with CSS classes, and the semantic structure was non-existent. To a parser, it was gibberish.

That's a £50 million revenue company, invisible to AI.

What's next

The rest of this book breaks down into practical sections:

- **Understanding how AI reads** (spoiler: very differently to humans)
- **Content architecture patterns** that work for both audiences
- **Testing and validation** you can do without specialist tools
- **Common problems and solutions** from real sites

Each chapter is short. Each has examples you can implement. And each assumes you're working on a real site with real constraints—budgets, deadlines, legacy code, and stakeholders who care about design.

Because here's the thing: the invisible users don't care about your constraints. They're visiting your site right now, reading it in ways you never intended, and making decisions that affect your business.

It's time to design for them.

Coming up in Chapter 2: How AI actually reads web pages—and why your beautiful design might be making you invisible.

Chapter 2: How AI Reads (vs. How Humans Read)

The human reading experience

When you visit a website, you experience it holistically. Your eyes scan the page, picking up visual hierarchy from size, colour, and position. You notice the large image at the top, the three columns of features, the bright red call-to-action button. You understand layout: the stuff in the sidebar is supplementary, the content in the center is primary, the navigation at the top tells you where you can go.

This happens in milliseconds, unconsciously, aided by years of web browsing patterns. You know a logo usually links home. You know hamburger menus hide navigation on mobile. You know footers contain contact information and legal links.

Now forget all of that.

The AI reading experience

When an AI agent visits your page, here's what happens:

1. **Fetch the HTML** - Just the raw source code, as delivered by the server
2. **Parse it into a tree structure** - The DOM (Document Object Model)
3. **Extract text and relationships** - Following the tree from top to bottom
4. **Tokenize the content** - Breaking it into chunks for processing
5. **Build understanding** - Using patterns learned from billions of web pages

Notice what's missing? Everything visual. The AI doesn't see your page. It reads your code.

Let me show you the difference with a real example.

Example: The hero section

Here's what a human sees when visiting a typical corporate homepage:

[Large background image of modern office]
We Transform Businesses Through Innovation
Strategic consulting for the digital age
[Learn More Button] [Contact Us Button]

Clean. Professional. Clear hierarchy. The large heading obviously matters most, the subheading provides context, the buttons show available actions.

Here's what the AI sees in the HTML:

```
<div class="hero">
  <div class="hero-content">
    <div class="hero-text">
      <div class="heading-main">We Transform Businesses Through Innovation</div>
      <div class="heading-sub">Strategic consulting for the digital age</div>
    </div>
    <div class="hero-actions">
      <a href="/services" class="btn-primary">Learn More</a>
      <a href="/contact" class="btn-secondary">Contact Us</a>
    </div>
  </div>
</div>
```

The AI sees: four nested div elements with class names, two text strings of equal semantic weight, and two links. It has no idea which text is more important. The class names `heading-main` and `heading-sub` mean nothing—they're styling hooks for CSS, not semantic indicators.

Now watch what happens when we make it semantic:

```
<section aria-label="Introduction">
  <h1>We Transform Businesses Through Innovation</h1>
  <p>Strategic consulting for the digital age</p>
  <nav aria-label="Main␣actions">
    <a href="/services">Learn More</a>
    <a href="/contact">Contact Us</a>
  </nav>
</section>
```

Same visual result with CSS. Completely different meaning to AI:

- The `h1` signals "this is the primary heading for this page"
- The `p` indicates "this is descriptive text"
- The `nav` element says "these links are navigation options"
- The `aria-label` attributes provide additional context

The AI now understands structure, hierarchy, and purpose.

DOM order is reading order

Here's something that catches developers out constantly: AI reads your DOM in document order, top to bottom, regardless of how CSS positions elements visually.

Consider this common layout pattern:

Visual layout (what humans see):

```
+----------------------------------+
| [Navigation Menu]          |
+----------------------------------+
| Main Content   | Sidebar    |
|                | - Related  |
|                | - Ads      |
|                | - Popular  |
+----------------------------------+
```

HTML order (what AI reads):

```
<nav>Navigation Menu</nav>
<aside>
  <div>Related Articles</div>
  <div>Advertisements</div>
  <div>Popular Posts</div>
```

```
</aside>
<main>
  <h1>Main Article Title</h1>
  <p>Your actual content...</p>
</main>
```

The developer put the sidebar before the main content in the HTML because it was easier with their grid system. Visually, with CSS, this looks fine to humans. But AI reads:

1. Navigation (fine)
2. Sidebar content including ads (noise)
3. Finally the actual article

By the time the AI reaches your main content, it's already processed dozens of unrelated links and promotional material. For an LLM with a token budget, those sidebar items might crowd out the actual article content.

The fix is simple—put main content first in the DOM:

```
<nav>Navigation Menu</nav>
<main>
  <h1>Main Article Title</h1>
  <p>Your actual content...</p>
</main>
<aside>
  <div>Related Articles</div>
  <div>Advertisements</div>
  <div>Popular Posts</div>
</aside>
```

Use CSS to position the sidebar where you want visually. The content order now matches the priority order.

Token budgets and working memory

When an LLM processes your page, it has a working memory limit measured in tokens. Think of tokens as roughly chunks of text—a word might be one or two tokens, depending on length and language.

Current AI models typically have context windows of:

- GPT-4: 128,000 tokens
- Claude: 200,000 tokens

- Gemini: 2,000,000 tokens

Sounds like plenty, right? Consider this: an average web page, with all its HTML, might use 10,000-50,000 tokens. But that's just the raw page. Add in:

- Navigation (repeated on every page)
- Footer content (copyright, links, legal)
- Headers and scripts (metadata, tracking codes)
- Advertisements and related content
- Comments sections

You can easily hit 100,000 tokens for a single article page when you include all the surrounding scaffolding.

Now the AI is making choices about what to pay attention to. If your main content is buried after 30,000 tokens of navigation, sidebar links, and ads, it's fighting for attention.

What gets lost

Let me be specific about what AI agents typically cannot perceive:

CSS-based information:

```
<div class="urgent">This requires immediate attention</div>
```

A human sees red text, bold weight, maybe a warning icon via CSS. The AI sees a div with some text. The urgency is invisible.

JavaScript-rendered content:

```
<div id="product-details"></div>
<script>
  loadProductDetails(12345);
</script>
```

A parser fetching raw HTML gets an empty div. The actual product details never appear unless the JavaScript executes. Many AI agents don't execute JavaScript—they're reading the source as delivered.

Visual hierarchy through styling:

```
<div style="font-size:␣48px">Big Important Text</div>
<div style="font-size:␣12px">Less important detail</div>
```

To AI, these are just two text strings. The size difference is meaningless. Use `h1` and `p` instead, and the hierarchy becomes semantic.

Images without text alternatives:

```
<img src="process-diagram.png">
```

The AI knows an image exists. It doesn't know what the image shows. If that diagram explains your entire service process, the AI has no idea. Add `alt="Five-step service process: consultation, planning, implementation, testing, and launch"` and now it does.

Content in canvas elements:

```
<canvas id="chart"></canvas>
<script>
  drawComplexChart(salesData);
</script>
```

The AI sees… a canvas element. Whatever chart you've drawn might as well not exist.

The three types of AI readers

Understanding how AI reads requires recognizing there are actually three distinct approaches in use:

1. Raw parsers (the minimalists)

These fetch HTML and parse it without executing JavaScript or loading CSS. They're looking at your source code directly. Search engine crawlers often work this way for initial discovery. Many LLM-based tools fetch raw HTML to save processing costs.

What they see: Pure HTML structure
What they miss: Anything requiring JavaScript, CSS-based visual cues
Example: Early-stage web crawlers, some API-based tools

2. Browser-based agents (the executors)

These run full browsers—Chrome, Firefox, or headless versions. They execute JavaScript, load CSS, run animations. They can interact: clicking buttons, filling forms, scrolling pages. Some can even wait for content to load asynchronously.

What they see: The rendered page, as a human would
What they miss: Nothing technical, but they still interpret via DOM, not visual perception
Example: Browser automation tools, some AI assistants with web interaction

3. Vision models (the screenshot readers)

These take screenshots of rendered pages and use computer vision to extract information. They can see visual hierarchy, notice that red warning text, understand layout through spatial relationships.

What they see: The visual presentation
What they miss: Links (can see buttons but not href values), underlying structure, non-visual metadata
Example: GPT-4 Vision, Claude with image analysis, multimodal AI systems

The practical implication

Your site needs to work for all three types. That sounds daunting, but there's good news: proper semantic HTML serves all three effectively.

The raw parser gets clear structure from semantic elements.
The browser-based agent can execute your JavaScript but doesn't depend on it for basic content.
The vision model sees proper visual hierarchy because you've styled semantic elements appropriately.

Here's a real example from my documentation work with Adobe Edge Delivery Services. I needed documentation that both developers could read and AI could parse.

Bad approach (how many docs work):

```
<div class="doc-section">
  <div class="doc-title">Installation</div>
  <div class="doc-content">
    <div class="code-block">npm install @adobe/aem-cli</div>
    <div class="note">Requires Node.js 16 or higher</div>
  </div>
</div>
```

Humans could read this fine with CSS. AI just saw nested divs with no semantic meaning.

Better approach (semantic structure):

```
<section>
  <h2>Installation</h2>
  <pre><code>npm install @adobe/aem-cli</code></pre>
  <aside role="note">
    <strong>Requirements:</strong> Node.js 16 or higher
  </aside>
</section>
```

Now:

- The `h2` signals a major section
- The `pre` and `code` elements identify executable code
- The `aside` with `role="note"` marks supplementary information
- The `strong` element indicates emphasis

All three AI reader types can extract meaningful information. And humans? Still see exactly what they saw before, because CSS still controls the presentation.

The scanning pattern fallacy

Steve Krug taught us that humans scan in an F-pattern or Z-pattern, and we should design accordingly. But AI doesn't scan—it processes sequentially or uses attention mechanisms to focus on semantically marked content.

This creates an interesting design challenge. Humans benefit from visual patterns that guide the eye. AI benefits from semantic patterns that guide parsing. The solution isn't choosing one over the other—it's recognizing that they're implemented in different layers.

Visual layer (CSS): Create the F-pattern, guide the eye with colour and size, make buttons look clickable
Semantic layer (HTML): Create clear hierarchy with headings, mark up navigation as `nav`, identify main content with `main`

The two layers work together. Remove the CSS and the semantic layer still makes sense. That's the test: does your content have meaning without styling?

What's coming

I've shown you how AI reads. Now you might be thinking: "This means I need to rebuild everything!"

Not necessarily. In the next chapter, we'll look at the principles that guide AI-friendly design—the equivalent of Krug's usability heuristics, but for invisible users. You'll see that many of these principles align with accessibility, mobile-first design, and progressive enhancement.

Making your site AI-readable isn't a separate project. It's often just doing what we should have been doing anyway, but with renewed purpose now that the stakes are higher.

Coming up in Chapter 3: The guiding principles for dual-audience design—how to make sites that work for both humans and AI without compromise.

Chapter 3: Guiding Principles

The don't make me think equivalent

Steve Krug's core principle was simple: make it obvious what things are and how to use them. For AI, we need something equally straightforward, but the "obvious" part works differently.

Humans rely on visual cues and learned patterns. AI relies on explicit structure and semantic meaning. The good news? You can serve both with the same HTML if you follow a few guiding principles.

Here are the four principles that underpin everything else in this book:

1. **Semantic clarity over visual clarity**
2. **Structure reveals intent**

3. **Metadata makes promises explicit**
4. **Redundancy serves different consumers**

Let's look at what each means in practice.

Principle 1: Semantic clarity over visual clarity

Visual clarity is making something look like what it is. A button looks clickable, a heading looks important, an error message looks urgent.

Semantic clarity is marking up something as what it is. A button uses `<button>`, a heading uses `<h1>`, an error message uses appropriate ARIA attributes.

The critical insight: you can have both. They're not competing approaches—they're complementary layers.

Example: The pricing table**

I worked with a SaaS company whose pricing page was driving good conversion from humans but was invisible to AI. When users asked chatbots "What does CompanyX charge for their pro plan?" the AI couldn't answer, despite the information being prominently displayed.

Here's what they had:

```
<div class="pricing-grid">
  <div class="plan-card␣plan-pro">
    <div class="plan-name">Professional</div>
    <div class="plan-price">£99<span class="period">/month</span></div>
    <div class="plan-features">
      <div class="feature">Up to 100 users</div>
      <div class="feature">Unlimited projects</div>
      <div class="feature">Priority support</div>
    </div>
    <div class="plan-cta">
      <a href="/signup?plan=pro" class="btn-primary">Get Started</a>
    </div>
  </div>
</div>
```

Visually: perfect. The price was large, the features were clearly listed, the call-to-action stood out.

Semantically: meaningless. The AI saw divs containing text strings. Nothing indicated that £99 was a price, that "Professional" was a plan tier, or that those features were a list of capabilities.

Here's the semantic version:

```
<section aria-labelledby="pricing-heading">
  <h2 id="pricing-heading">Pricing Plans</h2>
  <article itemscope itemtype="https://schema.org/Product">
    <h3 itemprop="name">Professional</h3>
    <p>
      <data itemprop="price" value="99">£99</data>
      <span itemprop="priceCurrency" content="GBP"></span>
      <span>/month</span>
    </p>
    <h4>Features:</h4>
    <ul>
      <li>Up to 100 users</li>
      <li>Unlimited projects</li>
      <li>Priority support</li>
    </ul>
    <p>
      <a href="/signup?plan=pro">Get Started</a>
    </p>
  </article>
</section>
```

Now we have:

- `section` and `article` elements defining document structure
- `h2`, `h3`, `h4` creating a clear hierarchy
- `ul` indicating a list (not just vertical text)
- Schema.org markup making the price explicit
- Proper semantic elements throughout

The AI can now answer: "CompanyX's Professional plan costs £99 per month and includes up to 100 users, unlimited projects, and priority support."

And humans? With CSS, they see exactly the same visual design as before. Nothing changes for them.

Principle 2: Structure reveals intent

Your HTML structure tells AI what matters and how things relate. This goes beyond just using semantic elements—it's about the relationships between them.

Heading hierarchy as content outline**

Think of your heading levels (h1 through h6) as a table of contents that AI can parse. This isn't just good practice—it's how AI builds a mental model of your content.

Bad structure:

```
<h1>Welcome to Our Site</h1>
<h3>Our Services</h3>
<h2>What We Do</h2>
<h3>Contact Information</h3>
<h2>About Us</h2>
```

The heading levels jump around without logic. AI can't tell what's a major section versus a subsection. The structure suggests chaos.

Good structure:

```
<h1>Welcome to Our Site</h1>
<h2>Our Services</h2>
  <h3>Web Development</h3>
  <h3>Design Consulting</h3>
<h2>About Us</h2>
  <h3>Our Team</h3>
  <h3>Our History</h3>
<h2>Contact Information</h2>
```

Now there's a clear hierarchy. The h1 is the page title. Each h2 starts a major section. Each h3 is a subsection under its h2. AI can build a proper content outline.

This mirrors how a human would create a document outline—and that's not coincidental. Good document structure serves both audiences.

Lists show relationships**

When you have related items, use list elements. This seems obvious, but you'd be surprised how often developers use div elements with CSS to create visual lists.

```
<!-- Don't do this -->
<div class="feature-list">
  <div class="feature">Fast performance</div>
  <div class="feature">Secure by default</div>
  <div class="feature">Easy to integrate</div>
</div>

<!-- Do this -->
<ul>
  <li>Fast performance</li>
  <li>Secure by default</li>
  <li>Easy to integrate</li>
</ul>
```

The `ul` element tells AI: "these items are related members of a group." That's semantic information the div version doesn't convey.

When order matters, use `ol`:

```
<ol>
  <li>Create an account</li>
  <li>Verify your email</li>
  <li>Complete your profile</li>
  <li>Start using the service</li>
</ol>
```

Now AI knows this is a sequence, not just a collection. If someone asks "What's the first step?" the AI can answer correctly.

Definition lists for key-value pairs**

Here's an under-used HTML element that's perfect for certain content patterns:

```
<dl>
  <dt>Location</dt>
  <dd>Manchester, UK</dd>

  <dt>Established</dt>
  <dd>1999</dd>

  <dt>Specialization</dt>
  <dd>Adobe Experience Manager consulting</dd>

  <dt>Day rate</dt>
  <dd>£800</dd>
</dl>
```

The `dl` (definition list) with `dt` (term) and `dd` (definition) creates explicit key-value relationships. AI can extract: "What's the location?" → "Manchester, UK". "When was it established?" → "1999".

Compare this to:

```
<div class="info-grid">
  <div class="info-row">
    <span class="label">Location</span>
    <span class="value">Manchester, UK</span>
  </div>
  <!-- etc -->
</div>
```

Same visual result with CSS. Completely different semantic meaning.

Principle 3: Metadata makes promises explicit

Sometimes content structure isn't enough. You need to make explicit promises about what content means and how it should be interpreted.

This is where metadata and structured data come in—not as SEO tricks, but as clear declarations of intent.

Schema.org: The lingua franca**

Schema.org provides a shared vocabulary for describing content. When you mark up content with Schema.org terms, you're speaking a language AI already understands.

Here's a real example from a client in the automotive sector:

```
<article itemscope itemtype="https://schema.org/LocalBusiness">
  <h1 itemprop="name">Manchester Motors</h1>
  <div itemprop="address" itemscope itemtype="https://schema.org/PostalAddress">
    <span itemprop="streetAddress">123 Oxford Road</span>,
    <span itemprop="addressLocality">Manchester</span>
    <span itemprop="postalCode">M1 7ED</span>
  </div>
  <p>
    Phone: <span itemprop="telephone">0161 123 4567</span><br>
    Email: <a href="mailto:info@manchestermotors.co.uk" itemprop="email">info@manchestermotors.
        ↪ co.uk</a>
  </p>
  <p itemprop="description">
    Authorized Land Rover service center specializing in maintenance and repairs.
  </p>
</article>
```

This microdata makes explicit promises:

- This is a local business
- The name is “Manchester Motors”

- The address components are structured and labeled
- The phone and email are marked as contact information
- The description explains what the business does

When an AI encounters this, it doesn’t have to guess or infer. The information is labeled and structured.

JSON-LD: The AI-friendly format**

While microdata works, JSON-LD (JavaScript Object Notation for Linked Data) is often easier to implement and maintain:

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "LocalBusiness",
  "name": "Manchester␣Motors",
  "address": {
    "@type": "PostalAddress",
    "streetAddress": "123␣Oxford␣Road",
    "addressLocality": "Manchester",
    "postalCode": "M1␣7ED"
  },
  "telephone": "0161␣123␣4567",
  "email": "info@manchestermotors.co.uk",
  "description": "Authorized␣Land␣Rover␣service␣center␣specializing␣in␣maintenance␣and␣repairs."
}
</script>
```

This sits in your page’s `<head>` or anywhere in the `<body>`. It doesn’t affect visual presentation at all. It’s purely for machine readers.

The advantage? You can implement structured data without touching your existing HTML. Your developers can maintain it separately from your design system.

Don’t overdo it**

A word of caution: Schema.org has hundreds of types and properties. You don’t need them all. Focus on:

- **Organization/LocalBusiness**: For company information
- **Product**: For products you sell
- **Article**: For blog posts and news
- **FAQPage**: For FAQ sections

- **HowTo**: For step-by-step instructions
- **Event**: For events and webinars

These cover 90% of use cases. Adding more rarely helps and can clutter your code.

Principle 4: Redundancy serves different consumers

Here's where dual-audience design gets interesting: sometimes you need to say the same thing in multiple ways because different consumers need different formats.

This feels wrong to developers trained in DRY (Don't Repeat Yourself), but remember—you're not serving just one consumer. You're serving humans who scan visually, screen readers that navigate by landmark, and AI that parses structure.

Visual + semantic + metadata**

Consider a product page:

```
<main>
  <article itemscope itemtype="https://schema.org/Product">
    <h1 itemprop="name">Professional Standing Desk</h1>

    <img src="desk.jpg"
         alt="White␣adjustable␣standing␣desk␣with␣electric␣height␣control␣showing␣desk␣raised␣to␣
              ↪ standing␣position"
         itemprop="image">

    <p itemprop="description">
      Electric height-adjustable desk with memory presets,
      cable management, and whisper-quiet motor.
      Suitable for home offices and professional workspaces.
    </p>

    <dl>
      <dt>Price</dt>
      <dd>
        <data itemprop="price" value="599">£599</data>
        <meta itemprop="priceCurrency" content="GBP">
      </dd>

      <dt>Availability</dt>
      <dd itemprop="availability" content="https://schema.org/InStock">
        In Stock
      </dd>

      <dt>Dimensions</dt>
      <dd>120cm × 60cm (adjustable height 70cm - 120cm)</dd>
    </dl>

    <button type="button" aria-label="Add␣Professional␣Standing␣Desk␣to␣shopping␣cart">
      Add to Cart
    </button>
  </article>
</main>
```

Look at what we're doing:

For humans:

- The image shows what it looks like
- The layout creates visual hierarchy
- The button is visually obvious

For screen readers:

- The `alt` text describes the image
- The `aria-label` makes the button action explicit

- The structure allows navigation by heading

For AI parsers:

- The Schema.org markup identifies this as a product
- The price is marked with actual value and currency
- The availability is explicitly stated
- The heading provides the product name

For all audiences:

- The semantic HTML creates clear structure
- The content is readable without CSS or JavaScript
- The information is complete and self-contained

This is redundancy with purpose. Each layer serves specific consumers without harming others.

The alt text principle**

Alt text is a perfect example of useful redundancy. Visual users see the image. Screen reader users hear the alt text. AI parsers read the alt text.

Bad alt text treats this as a chore:

```
<img src="product-001.jpg" alt="product">
```

Good alt text serves all audiences:

```
<img src="product-001.jpg"
    alt="Standing␣desk␣raised␣to␣maximum␣height␣showing␣the␣motor␣housing␣and␣control␣panel">
```

The visual user gets additional context if the image fails to load. The screen reader user understands what's depicted. The AI parser knows what the image shows even though it can't see it.

Navigation patterns**

Navigation is another area where redundancy helps:

```
<nav aria-label="Main␣navigation">
  <ul>
    <li><a href="/" aria-current="page">Home</a></li>
    <li><a href="/services">Services</a></li>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
  </ul>
</nav>

<!-- Later in page footer -->
<nav aria-label="Footer␣navigation">
  <h2>Quick Links</h2>
  <ul>
    <li><a href="/privacy">Privacy Policy</a></li>
    <li><a href="/terms">Terms of Service</a></li>
    <li><a href="/sitemap">Sitemap</a></li>
  </ul>
</nav>
```

Multiple navigation blocks? That's fine if they're labeled differently. The `aria-label` attributes distinguish them. AI can tell there's a main navigation and a footer navigation. Screen readers can jump between them. Visual users see them styled and positioned differently.

When principles conflict with reality

I can hear the objections already:

"Our design system uses div elements for everything."
"We can't change the HTML without breaking the CSS."
"The framework we use generates the markup."
"We don't have budget to rebuild everything."

All valid concerns. Here's the thing: you don't need perfection. You need improvement.

Start with high-value pages:

- Homepage
- Top landing pages
- Product/service pages
- Contact and about pages

Fix the most glaring issues:

- Add proper heading hierarchy
- Mark navigation with `nav` elements
- Use `main` for primary content
- Add alt text to images

Then incrementally improve:

- Add Schema.org to key content types
- Convert visual lists to semantic lists
- Improve link text ("click here" → "view pricing details")
- Add ARIA labels where needed

You don't need to touch every page. The 80/20 rule applies: fixing 20% of your pages (the high-traffic ones) solves 80% of the AI-readability problem.

The testing mindset

These principles aren't rules to memorize—they're guidelines to test against. For each page you create or modify, ask:

1. **If I removed all CSS, would the structure still make sense?**
2. **Can I outline the page using just the headings?**
3. **Are relationships between items explicit? (lists, definition lists, semantic elements)**
4. **Would an AI know what the key information is? (Schema.org, metadata)**
5. **Does redundancy serve different consumers without creating confusion?**

If you can answer yes to most of these, you're on the right track.

What's next

I've given you principles. Now you need patterns—specific implementations for common scenarios.

In the next chapter, we'll look at content architecture: how to structure different types of pages so that both humans and AI can navigate and understand them. We'll cover navigation patterns, article structure, product pages, and forms.

Each pattern will show you the before (what doesn't work) and after (what does), with real examples you can adapt to your own sites.

Coming up in Chapter 4: Content architecture patterns for common page types—the building blocks of AI-readable sites.

Chapter 4: Content Architecture for Machines

The building blocks

If the previous chapter gave you principles, this one gives you patterns. These are the specific structural approaches that work for both human and AI readers across common page types.

I'm not going to show you every possible variation. Instead, I'll focus on the patterns I see most often in client work—the ones that, when done badly, cause the most AI-readability problems.

Pattern 1: The article or blog post

Articles are where many sites get their first AI traffic. Someone asks a question, an AI searches for answers, and your article either gets recommended or doesn't.

The structure that fails:

```
<div class="post">
  <div class="post-header">
    <div class="post-title">How to Choose the Right CMS</div>
    <div class="post-meta">Posted by John Smith on March 15, 2024</div>
  </div>
  <div class="post-body">
    <div class="intro">
      Choosing a content management system is one of the most important
      decisions you'll make for your website...
    </div>
    <div class="section">
      <div class="section-title">Consider your requirements</div>
      <div class="section-content">...</div>
    </div>
    <!-- More sections -->
  </div>
</div>
```

This might look fine visually, but AI sees:

- No clear hierarchy (everything's a div)
- No indication of document structure
- No semantic metadata about authorship or publication date
- No way to distinguish the introduction from the main content

The structure that works:

```
<article itemscope itemtype="https://schema.org/Article">
  <header>
    <h1 itemprop="headline">How to Choose the Right CMS</h1>
    <p>
      By <span itemprop="author">John Smith</span> |
      <time itemprop="datePublished" datetime="2024-03-15">15 March 2024</time>
    </p>
  </header>

  <div itemprop="articleBody">
    <p class="lede">
      Choosing a content management system is one of the most important
      decisions you'll make for your website...
    </p>

    <h2>Consider your requirements</h2>
    <p>Start by listing what you actually need...</p>

    <h3>Content authoring needs</h3>
    <p>How many people will create content?...</p>

    <h3>Technical requirements</h3>
    <p>What integrations do you need?...</p>

    <h2>Evaluate the options</h2>
    <p>Once you know what you need...</p>

    <h3>Traditional CMS platforms</h3>
    <p>WordPress, Drupal, and similar systems...</p>

    <h3>Headless CMS solutions</h3>
    <p>Modern headless systems like Contentful...</p>

    <h2>Making the decision</h2>
    <p>With your requirements clear and options evaluated...</p>
  </div>
</article>
```

What changed:

Document structure:

- `article` element wraps the entire post
- `header` contains metadata
- Proper `h1` through `h3` hierarchy creates an outline:
  - How to Choose the Right CMS
    - Consider your requirements
      - Content authoring needs
      - Technical requirements
    - Evaluate the options
      - Traditional CMS platforms
      - Headless CMS solutions
    - Making the decision

Semantic metadata:

- Schema.org Article type
- `headline` property for the title
- `author` and `datePublished` with proper datetime format
- `articleBody` wrapping the content

For AI, this provides:

- Clear topic from the h1

- Outline structure from headings
- Author attribution
- Publication date (useful for determining recency)
- Distinct introduction paragraph

For humans:

- Same visual presentation with CSS
- Better accessibility (screen readers can navigate by heading)
- Clearer document structure when styles fail

Pattern 2: The product page

Product pages are where the economic stakes are highest. If AI can't parse your product information, it can't recommend your products.

The common anti-pattern:

```
<div class="product-page">
  <div class="product-gallery">
    <img src="product-main.jpg">
    <div class="thumbnails">
      <img src="product-1.jpg">
      <img src="product-2.jpg">
    </div>
  </div>

  <div class="product-info">
    <div class="product-name">Wireless Keyboard</div>
    <div class="product-price">£79.99</div>
    <div class="product-rating">
      <span class="stars">5 stars</span>
      <span class="review-count">127 reviews</span>
    </div>
  </div>

  <div class="product-details">
    <div class="tab-content" id="specs" style="display:none">
      Battery life: 6 months
      Connectivity: Bluetooth 5.0
      Layout: UK QWERTY
    </div>
  </div>
</div>
```

Problems:

- Product specs hidden in JavaScript tabs (display:none means parsers might skip it)
- No structured data about price or availability
- Rating shown visually but not semantically
- Images without descriptive alt text
- No clear indication this is a product

The pattern that works:

```
<main>
  <article itemscope itemtype="https://schema.org/Product">
    <h1 itemprop="name">Wireless Keyboard</h1>

    <div class="product-images">
      <img itemprop="image"
```

```
         src="product-main.jpg"
         alt="Wireless␣keyboard␣with␣backlit␣keys␣in␣dark␣grey␣finish">
    <img src="product-1.jpg"
         alt="Close-up␣view␣of␣keyboard␣keys␣showing␣backlighting">
    <img src="product-2.jpg"
         alt="Side␣view␣showing␣keyboard␣profile␣and␣USB␣receiver">
  </div>

  <section>
    <h2>Overview</h2>
    <p itemprop="description">
      Professional wireless keyboard with LED backlighting,
      six-month battery life, and Bluetooth 5.0 connectivity.
      Features UK QWERTY layout with dedicated media keys.
    </p>

    <dl>
      <dt>Price</dt>
      <dd itemprop="offers" itemscope itemtype="https://schema.org/Offer">
        <data itemprop="price" value="79.99">£79.99</data>
        <meta itemprop="priceCurrency" content="GBP">
        <link itemprop="availability" href="https://schema.org/InStock">
        <span>In Stock</span>
      </dd>

      <dt>Customer Rating</dt>
      <dd itemprop="aggregateRating" itemscope itemtype="https://schema.org/AggregateRating">
        <data itemprop="ratingValue" value="4.8">4.8</data> out of 5 stars
        (<data itemprop="reviewCount" value="127">127</data> reviews)
      </dd>
    </dl>
  </section>

  <section>
    <h2>Technical Specifications</h2>
    <dl>
      <dt>Battery Life</dt>
      <dd>6 months (typical use)</dd>

      <dt>Connectivity</dt>
      <dd>Bluetooth 5.0, 2.4GHz USB receiver included</dd>

      <dt>Layout</dt>
      <dd>UK QWERTY with number pad</dd>

      <dt>Dimensions</dt>
      <dd>440mm × 130mm × 25mm</dd>

      <dt>Weight</dt>
      <dd>600g (without batteries)</dd>
    </dl>
  </section>

  <section>
    <h2>Purchase</h2>
    <form action="/cart/add" method="post">
      <input type="hidden" name="product_id" value="WK-100">
      <label for="quantity">Quantity:</label>
      <input type="number" id="quantity" name="quantity" value="1" min="1">
      <button type="submit">Add to Cart</button>
    </form>
  </section>
</article>
```

```
</main>
```

What this provides:

For AI parsing:

- Clear Product schema with name, description, image
- Structured pricing with currency and availability
- Rating as structured data, not just visual stars
- Specifications in definition lists (key-value pairs)
- All content visible regardless of JavaScript state

For humans:

- All the same information, styled however you want
- Tabs can still work (progressive enhancement)
- Same visual design achievable with CSS

For search engines:

- Rich snippets showing price, rating, availability
- Better understanding of product attributes
- Clear product identity

Pattern 3: The navigation menu

Navigation is interesting because it serves multiple purposes: humans use it to move around, AI uses it to understand site structure.

The problematic pattern:

```
<div class="nav">
  <a href="/">Home</a>
  <div class="dropdown">
    <a href="/services">Services</a>
    <div class="dropdown-content">
      <a href="/services/web">Web Development</a>
      <a href="/services/mobile">Mobile Apps</a>
      <a href="/services/consulting">Consulting</a>
    </div>
  </div>
  <a href="/about">About</a>
  <a href="/contact">Contact</a>
</div>
```

Issues:

- Not marked as navigation
- Dropdown structure unclear (just nested divs)
- No indication which page you’re on
- Relationship between parent and child items unclear

The better pattern:

```
<nav aria-label="Main␣navigation">
  <ul>
    <li><a href="/" aria-current="page">Home</a></li>
    <li>
      <a href="/services" aria-expanded="false">Services</a>
      <ul>
        <li><a href="/services/web">Web Development</a></li>
        <li><a href="/services/mobile">Mobile Apps</a></li>
        <li><a href="/services/consulting">Consulting</a></li>
      </ul>
    </li>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
  </ul>
</nav>
```

Improvements:

Semantic structure:

- `nav` element marks this as navigation
- `aria-label` distinguishes this navigation from others (footer nav, breadcrumbs)
- Nested `ul` elements show hierarchical relationships
- `aria-current` indicates the current page
- `aria-expanded` shows dropdown state

For AI:

- Clear site structure (Services has three sub-areas)
- Navigation relationships explicit
- Current location indicated

For humans:

- Still works without JavaScript (nested lists)
- Progressive enhancement adds hover/click behaviour
- Screen readers can navigate effectively

Bonus: Breadcrumbs**

While we're on navigation, breadcrumbs are particularly valuable for AI:

```
<nav aria-label="Breadcrumb">
  <ol itemscope itemtype="https://schema.org/BreadcrumbList">
    <li itemprop="itemListElement" itemscope itemtype="https://schema.org/ListItem">
      <a itemprop="item" href="/">
        <span itemprop="name">Home</span>
      </a>
      <meta itemprop="position" content="1">
    </li>
    <li itemprop="itemListElement" itemscope itemtype="https://schema.org/ListItem">
      <a itemprop="item" href="/services">
        <span itemprop="name">Services</span>
      </a>
      <meta itemprop="position" content="2">
    </li>
    <li itemprop="itemListElement" itemscope itemtype="https://schema.org/ListItem">
      <span itemprop="name">Web Development</span>
      <meta itemprop="position" content="3">
    </li>
  </ol>
</nav>
```

This tells AI:

- Where this page sits in the hierarchy
- The path from homepage to current location
- The relationship between levels

Search engines often display this as breadcrumb navigation in results, and AI can use it to understand context.

Pattern 4: The FAQ section

FAQ pages are goldmine content for AI. People ask questions, AI looks for answers—FAQ pages should be the perfect match.

The missed opportunity:

```
<div class="faq-section">
  <div class="faq-item">
    <div class="question">How long does delivery take?</div>
    <div class="answer">Standard delivery takes 3-5 business days...</div>
  </div>
  <div class="faq-item">
    <div class="question">What payment methods do you accept?</div>
    <div class="answer">We accept all major credit cards...</div>
  </div>
</div>
```

This is readable but provides no semantic structure.

The optimized pattern:

```
<section itemscope itemtype="https://schema.org/FAQPage">
  <h2>Frequently Asked Questions</h2>

  <article itemscope itemprop="mainEntity" itemtype="https://schema.org/Question">
    <h3 itemprop="name">How long does delivery take?</h3>
    <div itemscope itemprop="acceptedAnswer" itemtype="https://schema.org/Answer">
      <p itemprop="text">
        Standard delivery takes 3-5 business days. Express delivery
        (next business day) is available for orders placed before 2pm.
      </p>
    </div>
  </article>

  <article itemscope itemprop="mainEntity" itemtype="https://schema.org/Question">
    <h3 itemprop="name">What payment methods do you accept?</h3>
    <div itemscope itemprop="acceptedAnswer" itemtype="https://schema.org/Answer">
      <p itemprop="text">
        We accept all major credit cards (Visa, Mastercard, American Express),
        PayPal, and bank transfer for business accounts.
      </p>
    </div>
  </article>

  <article itemscope itemprop="mainEntity" itemtype="https://schema.org/Question">
    <h3 itemprop="name">Can I return items?</h3>
    <div itemscope itemprop="acceptedAnswer" itemtype="https://schema.org/Answer">
      <p itemprop="text">
        Yes, we offer a 30-day return policy. Items must be unused and in
        original packaging. Contact our support team to arrange a return.
      </p>
    </div>
  </article>
</section>
```

This pattern:

Makes questions discoverable:

- Each question is a proper heading (h3)
- Schema.org FAQPage type marks the section
- Each Question/Answer pair is structured

Enables direct answers:

- AI can extract question and answer pairs
- Answers are complete and self-contained
- Structure makes it clear which text answers which question

Works for humans:

- Visual styling unchanged
- Can add accordion JavaScript if desired
- Screen readers navigate by heading

When someone asks an AI "How long does delivery take from CompanyX?" the AI can find this FAQ, extract the answer, and respond confidently.

Pattern 5: The contact page

Contact pages seem simple but often fail AI parsing because information is scattered or embedded in images.

The common failure:

```
<div class="contact-page">
  <div class="contact-info">
    <div class="office">
      <div>Manchester Office</div>
      <div>123 Oxford Road</div>
      <div>Manchester M1 7ED</div>
      <div>0161 123 4567</div>
    </div>
  </div>
  <div class="contact-form">
    <input placeholder="Name">
    <input placeholder="Email">
    <textarea placeholder="Message"></textarea>
    <button>Send</button>
  </div>
</div>
```

Problems:

- No semantic markup for address components
- Phone number not marked as such
- Form fields lack proper labels
- No indication what the form does

The structured approach:

```
<main>
  <h1>Contact Us</h1>

  <section itemscope itemtype="https://schema.org/LocalBusiness">
    <h2>Our Office</h2>
    <div itemprop="name">Digital Domain Technologies Ltd</div>
    <address itemprop="address" itemscope itemtype="https://schema.org/PostalAddress">
      <span itemprop="streetAddress">123 Oxford Road</span><br>
      <span itemprop="addressLocality">Manchester</span>
      <span itemprop="postalCode">M1 7ED</span><br>
      <span itemprop="addressCountry">United Kingdom</span>
    </address>

    <dl>
      <dt>Phone</dt>
      <dd><a href="tel:+441611234567" itemprop="telephone">0161 123 4567</a></dd>

      <dt>Email</dt>
      <dd><a href="mailto:info@digitaldomain.co.uk" itemprop="email">info@digitaldomain.co.uk</a
          ↪ ></dd>

      <dt>Business Hours</dt>
      <dd itemprop="openingHours" content="Mo-Fr␣09:00-17:00">
        Monday to Friday, 9am - 5pm
      </dd>
    </dl>
  </section>

  <section>
    <h2>Send us a message</h2>
    <form action="/contact/submit" method="post">
      <div>
        <label for="name">Your Name</label>
        <input type="text" id="name" name="name" required>
      </div>

      <div>
        <label for="email">Email Address</label>
        <input type="email" id="email" name="email" required>
      </div>

      <div>
        <label for="phone">Phone Number (optional)</label>
        <input type="tel" id="phone" name="phone">
      </div>

      <div>
        <label for="subject">Subject</label>
        <input type="text" id="subject" name="subject" required>
      </div>

      <div>
        <label for="message">Message</label>
        <textarea id="message" name="message" rows="6" required></textarea>
      </div>

      <button type="submit">Send Message</button>
    </form>
  </section>
</main>
```

This provides:

Structured contact information:

- `address` element for the postal address
- Schema.org LocalBusiness with address components
- Phone as clickable link (works on mobile)
- Email as mailto link
- Opening hours in machine-readable format

Proper form structure:

- Every input has a `label` explicitly associated via `id`
- Input types match data (email, tel, text)
- Required fields marked
- Clear purpose for the form

For AI:

- Can extract phone number to answer “What’s the phone number?”
- Can determine opening hours
- Can parse the complete address
- Understands what information the form collects

Pattern 6: The data table

Tables are often misused (for layout) or under-marked (missing semantic information). When used correctly, they’re brilliant for AI.

The bare minimum:

```
<table>
  <tr>
    <td>Plan</td>
    <td>Price</td>
    <td>Users</td>
  </tr>
  <tr>
    <td>Basic</td>
    <td>£29/month</td>
    <td>5</td>
  </tr>
  <tr>
    <td>Professional</td>
    <td>£99/month</td>
    <td>25</td>
  </tr>
</table>
```

AI sees rows and cells but doesn’t know which row is headers versus data.

The semantic version:

```
<table>
  <caption>Pricing Plans Comparison</caption>
  <thead>
    <tr>
      <th scope="col">Plan</th>
      <th scope="col">Price</th>
      <th scope="col">Maximum Users</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th scope="row">Basic</th>
      <td>£29/month</td>
      <td>5</td>
    </tr>
    <tr>
      <th scope="row">Professional</th>
      <td>£99/month</td>
      <td>25</td>
    </tr>
    <tr>
      <th scope="row">Enterprise</th>
      <td>£299/month</td>
      <td>Unlimited</td>
    </tr>
  </tbody>
</table>
```

Now:

- `caption` provides context
- `thead` separates headers from data
- `th` elements with `scope` indicate what they describe
- `tbody` contains the actual data

AI can now answer: “How much is the Professional plan?” by finding the row where the row header equals “Professional” and extracting the price cell.

Patterns vs. templates

These patterns aren’t rigid templates. They’re starting points. Your specific needs might require variations:

- Different Schema.org types
- Additional ARIA attributes
- More or less nesting
- Different heading levels

The key is understanding the principles behind each pattern:

1. **Use semantic elements** (`article`, `section`, `nav`, `header`, etc.)
2. **Create clear hierarchies** (proper heading levels)
3. **Make relationships explicit** (lists, definition lists, tables)
4. **Add structured data** where it provides value
5. **Label everything** properly (ARIA labels, form labels, image alt text)

Testing these patterns

For each pattern, run the morning-after test:

1. View the page source
2. Copy the HTML
3. Paste into ChatGPT or Claude
4. Ask: “What information is on this page? What actions can I take?”

If the AI can extract the key information accurately, your structure works. If it misses things or gets confused, you’ve found problems to fix.

Next chapter, we’ll move from content structure to metadata—the layer that makes promises explicit about what your content means and how it should be understood.

Coming up in Chapter 5: Metadata that works—Schema.org, JSON-LD, and making your content explicitly understandable to AI.

Chapter 5: Metadata That Works

The promise layer

In the previous chapter, we looked at content structure—using semantic HTML to create clear hierarchies and relationships. But sometimes structure alone isn't enough. Sometimes you need to make explicit promises about what your content means.

That's what metadata does. It's a layer of explicit declarations: "This text string is a price in British pounds." "This date is when the article was published." "This person is the author, not just mentioned in passing."

Think of metadata as annotations in the margins of your HTML. The content is still there for humans to read, but the annotations help AI understand context and meaning that might otherwise be ambiguous.

The metadata spectrum

Before we dive into implementation, let's acknowledge that metadata exists on a spectrum from "barely worth the effort" to "critical for function."

Low value:

- Metadata that duplicates obvious structure
- Over-specification that doesn't change AI behaviour
- Deprecated or ignored schemas

Medium value:

- Basic Schema.org for standard content types
- OpenGraph for social sharing
- Standard meta tags

High value:

- Structured data that enables rich results
- Metadata that disambiguates content
- Machine-readable formats for data AI needs to extract

I'll focus on the medium-to-high value range. There's no point implementing metadata that doesn't move the needle.

Schema.org: The practical bits

Schema.org provides a shared vocabulary of types and properties. The full vocabulary contains hundreds of types—Organization, Person, Product, Event, Article, Recipe, Movie, and on and on.

You don't need most of them. Here are the types I actually use in client work, ranked by frequency:

1. **Organization / LocalBusiness** For company information, contact details, locations.

2. **Article / BlogPosting / NewsArticle**
For content marketing, blogs, news sections.

3. **Product / Offer** For e-commerce or any product pages.

4. **FAQPage / Question / Answer** For support content and FAQ sections.

5. **HowTo** For tutorials and step-by-step guides.

6. **WebPage / WebSite** General page metadata, site search.

These six types cover about 90% of what most sites need. If you're doing something more specialized—recipes, events, courses, job postings—there are types for those too. But start with these.

JSON-LD vs. microdata vs. RDFa

There are three ways to add Schema.org markup to your pages. Let me save you some time: use JSON-LD.

Microdata embeds markup directly in your HTML attributes:

```
<div itemscope itemtype="https://schema.org/Person">
  <span itemprop="name">Tom Cranstoun</span>
  <span itemprop="jobTitle">Principal Consultant</span>
</div>
```

RDFa also embeds in HTML, with different attribute names:

```
<div vocab="https://schema.org/" typeof="Person">
  <span property="name">Tom Cranstoun</span>
  <span property="jobTitle">Principal Consultant</span>
</div>
```

JSON-LD separates structured data from your HTML:

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "Person",
  "name": "Tom␣Cranstoun",
  "jobTitle": "Principal␣Consultant"
}
</script>
```

Why JSON-LD wins:

- **Separation of concerns**: Your markup doesn't change
- **Easier to maintain**: Update JSON without touching HTML
- **Easier to generate**: Your CMS can output JSON from structured fields
- **Less error-prone**: Simpler syntax, easier to validate
- **Google's preference**: Explicitly recommended by Google

The only downside: you might duplicate information that's already in your HTML. But that redundancy is worthwhile for the maintainability gains.

Pattern: Organization markup

Every site should have basic organization markup. This goes in your site-wide template, typically on the homepage:

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "Organization",
  "name": "Digital␣Domain␣Technologies␣Ltd",
  "url": "https://www.digitaldomain.co.uk",
  "logo": "https://www.digitaldomain.co.uk/logo.png",
  "description": "Enterprise␣content␣management␣and␣web␣development␣consultancy",
  "address": {
    "@type": "PostalAddress",
    "streetAddress": "123␣Oxford␣Road",
    "addressLocality": "Manchester",
    "postalCode": "M1␣7ED",
    "addressCountry": "GB"
  },
  "contactPoint": {
    "@type": "ContactPoint",
    "telephone": "+44-161-123-4567",
    "contactType": "customer␣service",
    "email": "info@digitaldomain.co.uk",
    "availableLanguage": ["en"]
  },
  "sameAs": [
    "https://www.linkedin.com/in/tomharris",
    "https://twitter.com/digitaldomain"
  ]
}
</script>
```

This tells AI (and search engines):

- Your official company name
- Your web address
- Where to find your logo
- Physical location with structured address
- How to contact you
- Social media profiles

If you're a local business (restaurant, shop, service provider), use `LocalBusiness` instead of `Organization` and add opening hours:

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "LocalBusiness",
  "name": "Manchester␣Motors",
  "image": "https://example.com/photo.jpg",
  "address": {
    "@type": "PostalAddress",
    "streetAddress": "123␣Oxford␣Road",
    "addressLocality": "Manchester",
    "postalCode": "M1␣7ED",
    "addressCountry": "GB"
  },
  "geo": {
    "@type": "GeoCoordinates",
    "latitude": 53.4808,
    "longitude": -2.2426
  },
  "telephone": "+44-161-123-4567",
  "openingHoursSpecification": [
    {
      "@type": "OpeningHoursSpecification",
      "dayOfWeek": ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"],
      "opens": "09:00",
      "closes": "17:00"
    },
    {
      "@type": "OpeningHoursSpecification",
      "dayOfWeek": "Saturday",
      "opens": "09:00",
      "closes": "13:00"
    }
  ]
}
</script>
```

Now AI can answer: "What time does Manchester Motors close on Friday?" → "17:00" (5pm).

Pattern: Article markup

For blog posts, articles, case studies—any editorial content:

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "Article",
  "headline": "How␣to␣Migrate␣from␣Legacy␣CMS␣to␣Modern␣Web␣Architecture",
  "description": "A␣practical␣guide␣to␣migrating␣from␣traditional␣content␣management␣systems␣to␣
      ↪ modern␣web␣architectures,␣including␣common␣pitfalls␣and␣solutions.",
  "image": "https://example.com/article-image.jpg",
  "author": {
    "@type": "Person",
    "name": "Tom␣Cranstoun",
    "url": "https://www.digitaldomain.co.uk/about"
  },
  "publisher": {
    "@type": "Organization",
    "name": "Digital␣Domain␣Technologies",
    "logo": {
      "@type": "ImageObject",
      "url": "https://www.digitaldomain.co.uk/logo.png"
    }
  },
  "datePublished": "2024-03-15T09:00:00Z",
  "dateModified": "2024-03-20T14:30:00Z"
}
</script>
```

Key points:

Use proper ISO 8601 dates: `2024-03-15T09:00:00Z` not "March 15, 2024"

Include both datePublished and dateModified: AI cares about recency

Link author and publisher: These can be full objects or just references

Add an image: Helps with social sharing and rich results

If your site has multiple authors, you can use an array:

```
"author": [
  {
    "@type": "Person",
    "name": "Tom Cranstoun"
  },
  {
    "@type": "Person",
    "name": "Jane Smith"
  }
]
```

Pattern: Product markup

Product pages need the most detailed metadata because AI often makes purchasing recommendations:

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "Product",
  "name": "Professional␣Standing␣Desk",
  "image": [
    "https://example.com/desk-1.jpg",
    "https://example.com/desk-2.jpg",
    "https://example.com/desk-3.jpg"
  ],
  "description": "Electric␣height-adjustable␣desk␣with␣memory␣presets,␣cable␣management,␣and␣
      ↪ whisper-quiet␣motor.",
  "sku": "SD-PRO-100",
  "mpn": "SD-PRO-100",
  "brand": {
    "@type": "Brand",
    "name": "OfficeElite"
  },
  "offers": {
    "@type": "Offer",
    "url": "https://example.com/products/standing-desk",
    "priceCurrency": "GBP",
    "price": "599.00",
    "priceValidUntil": "2024-12-31",
    "availability": "https://schema.org/InStock",
    "itemCondition": "https://schema.org/NewCondition",
    "seller": {
      "@type": "Organization",
      "name": "Digital␣Domain␣Technologies"
    }
  },
  "aggregateRating": {
    "@type": "AggregateRating",
    "ratingValue": "4.8",
    "reviewCount": "127",
    "bestRating": "5",
    "worstRating": "1"
  }
}
</script>
```

The `offers` section is critical. Notice:

price as a string: “599.00” not 599
priceCurrency: ISO 4217 code (GBP, USD, EUR)

availability: Use Schema.org URLs (InStock, OutOfStock, PreOrder, etc.)
priceValidUntil: When does this price expire?

If you have multiple offers (different sizes, colours, configurations), use an array:

```
"offers": [
  {
    "@type": "Offer",
    "name": "120cm width",
    "price": "599.00",
    "priceCurrency": "GBP",
    "availability": "https://schema.org/InStock"
  },
  {
    "@type": "Offer",
    "name": "140cm width",
    "price": "699.00",
    "priceCurrency": "GBP",
    "availability": "https://schema.org/InStock"
  }
]
```

Pattern: FAQ markup

FAQ pages are perfect for AI because they're already in question-and-answer format:

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "FAQPage",
  "mainEntity": [
    {
      "@type": "Question",
      "name": "How␣long␣does␣delivery␣take?",
      "acceptedAnswer": {
        "@type": "Answer",
        "text": "Standard␣delivery␣takes␣3-5␣business␣days.␣Express␣delivery␣(next␣business␣day)␣
             ↪ is␣available␣for␣orders␣placed␣before␣2pm."
      }
    },
    {
      "@type": "Question",
      "name": "What␣payment␣methods␣do␣you␣accept?",
      "acceptedAnswer": {
        "@type": "Answer",
        "text": "We␣accept␣all␣major␣credit␣cards␣(Visa,␣Mastercard,␣American␣Express),␣PayPal,␣
             ↪ and␣bank␣transfer␣for␣business␣accounts."
      }
    },
    {
      "@type": "Question",
      "name": "Can␣I␣return␣items?",
      "acceptedAnswer": {
        "@type": "Answer",
        "text": "Yes,␣we␣offer␣a␣30-day␣return␣policy.␣Items␣must␣be␣unused␣and␣in␣original␣
             ↪ packaging.␣Contact␣our␣support␣team␣to␣arrange␣a␣return."
      }
    }
  ]
}
</script>
```

This enables:

- Direct answers when AI is queried
- Rich results in search showing expandable FAQs
- Clear question-answer pairing

Keep answers complete and self-contained. Don’t say “See our returns policy” when you can state the policy directly.

Pattern: HowTo markup

Step-by-step guides benefit from explicit structure:

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "HowTo",
  "name": "How␣to␣Install␣Node.js␣CLI␣Tools",
  "description": "Step-by-step␣guide␣to␣installing␣and␣configuring␣Node.js␣command-line␣tools.",
  "totalTime": "PT10M",
  "estimatedCost": {
    "@type": "MonetaryAmount",
    "currency": "GBP",
    "value": "0"
  },
  "tool": [
    {
      "@type": "HowToTool",
      "name": "Node.js␣16␣or␣higher"
    },
    {
      "@type": "HowToTool",
      "name": "npm␣or␣yarn␣package␣manager"
    }
  ],
  "step": [
    {
      "@type": "HowToStep",
      "name": "Check␣Node.js␣version",
      "text": "Verify␣you␣have␣Node.js␣16␣or␣higher␣installed␣by␣running:␣node␣--version",
      "url": "https://example.com/guide#step1"
    },
    {
      "@type": "HowToStep",
      "name": "Install␣the␣CLI",
      "text": "Install␣the␣CLI␣tool␣globally␣using:␣npm␣install␣-g␣example-cli",
      "url": "https://example.com/guide#step2"
    },
    {
      "@type": "HowToStep",
      "name": "Verify␣installation",
      "text": "Confirm␣successful␣installation␣by␣running:␣example-cli␣--version",
      "url": "https://example.com/guide#step3"
    }
  ]
}
</script>
```

The `totalTime` uses ISO 8601 duration format:

- PT10M = 10 minutes
- PT1H30M = 1 hour 30 minutes

- P2D = 2 days

This helps AI answer: "How long does it take to install the CLI tool?" → "10 minutes"

When NOT to use metadata

Metadata isn't always worthwhile. Skip it when:

1. The content is already unambiguous

If your heading hierarchy is clear and your semantic HTML is solid, adding metadata might be redundant:

```
<!-- This doesn't need Schema.org -->
<article>
  <h1>Our Services</h1>
  <section>
    <h2>Web Development</h2>
    <p>We build fast, accessible websites...</p>
  </section>
</article>
```

The structure speaks for itself. Adding WebPage schema here adds no value.

2. You can't maintain it

Stale metadata is worse than no metadata. If your prices change frequently and you can't keep Schema.org in sync, skip it. Wrong information is worse than no explicit information.

3. The content type doesn't matter for AI

Not everything needs to be machine-readable. Your privacy policy? Probably fine without structured data. Your terms of service? Unlikely anyone's asking AI about it.

4. You're adding metadata for metadata's sake

I've seen sites with Person schema for every mentioned name, Place schema for every location reference, Thing schema for… well, things. This is overkill. Focus on content that answers user queries.

The metadata generation problem

Here's a practical challenge: how do you actually add this metadata to your pages?

Option 1: Manual JSON-LD**

Write it by hand, paste it in. Fine for static sites with few pages. Doesn't scale.

Option 2: Template-based generation**

Your CMS or site generator creates JSON-LD from page fields:

```
// Pseudo-code for a blog post template
const articleSchema = {
  "@context": "https://schema.org",
  "@type": "Article",
  "headline": page.title,
  "description": page.excerpt,
  "datePublished": page.publishDate.toISOString(),
  "dateModified": page.modifiedDate.toISOString(),
  "author": {
    "@type": "Person",
    "name": page.author.name
  }
};
```

This works well if your CMS has structured fields for metadata.

Option 3: Build-time generation**

For static sites, generate JSON-LD during build:

```
// Example using a simple builder
function generateArticleSchema(article) {
  return {
    "@context": "https://schema.org",
    "@type": "Article",
    "headline": article.title,
    "datePublished": article.date,
    "author": {
      "@type": "Person",
      "name": article.author
    }
  };
}

// In your build script
const schema = generateArticleSchema(pageData);
const scriptTag = `<script type="application/ld+json">${JSON.stringify(schema, null, 2)}</
    ↪ script>`;
```

Option 4: Dynamic generation**

For sites with user-generated content or frequent updates, generate JSON-LD server-side on each request:

```
// Express.js example
app.get('/products/:id', async (req, res) => {
  const product = await getProduct(req.params.id);

  const schema = {
    "@context": "https://schema.org",
    "@type": "Product",
    "name": product.name,
    "offers": {
      "@type": "Offer",
      "price": product.price.toString(),
      "priceCurrency": "GBP",
      "availability": product.inStock
        ? "https://schema.org/InStock"
        : "https://schema.org/OutOfStock"
    }
  };

  res.render('product', { product, schema });
});
```

Choose based on your site architecture and update frequency.

Validation and testing

Don't trust your Schema.org markup without validation. Use these tools:

Google's Rich Results Test:
https://search.google.com/test/rich-results

Paste your URL or HTML and see if Google can parse your structured data. Shows warnings and errors.

Schema.org Validator:
https://validator.schema.org/

More lenient than Google's tool. Good for checking if your JSON-LD is valid Schema.org, even if it won't trigger rich results.

Your own AI test:

The simplest check:

1. Copy your page HTML (including the JSON-LD script)
2. Paste into ChatGPT or Claude
3. Ask specific questions: "What's the price?" "Who's the author?" "When was this published?"

If the AI can extract the information correctly, your metadata works.

Common mistakes

Mistake 1: Invalid JSON**

```
{
  "@context": "https://schema.org",
  "@type": "Product",
  "name": "Standing Desk",
  "price": "599" // Missing comma
  "currency": "GBP"
}
```

JSON is strict. Missing commas, trailing commas, unescaped quotes all break parsing. Use a JSON validator or let your code editor highlight errors.

Mistake 2: Wrong data types**

```
{
  "price": 599, // Should be string "599" or "599.00"
  "ratingValue": "4.8 stars", // Should be number 4.8
  "reviewCount": "127" // Should be number 127 or string "127"
}
```

Check Schema.org documentation for expected types. Price is text, ratings are numbers, dates are ISO strings.

Mistake 3: Incomplete required properties**

For Product schema, you need:

- name
- image

- offers (with price, priceCurrency, availability)

Missing any of these and Google won't show rich results. The validator will tell you what's required.

Mistake 4: Mismatched content**

```
<h1>Professional Standing Desk</h1>
<script type="application/ld+json">
{
  "@type": "Product",
  "name": "Adjustable␣Desk␣Pro" // Doesn't match visible name
}
</script>
```

Your metadata should match your visible content. Google may ignore structured data that contradicts what's on the page.

Mistake 5: Over-nesting**

```
{
  "@type": "Product",
  "name": "Desk",
  "offers": {
    "@type": "Offer",
    "price": "599",
    "seller": {
      "@type": "Organization",
      "name": "Company",
      "address": {
        "@type": "PostalAddress",
        "streetAddress": "123 Main St",
        "addressLocality": "Manchester",
        "postalCode": "M1 7ED",
        "geo": {
          "@type": "GeoCoordinates",
          "latitude": 53.4808,
          "longitude": -2.2426
        }
      }
    }
  }
}
```

This works but is unnecessarily complex for a product page. Keep it as simple as possible while including necessary information.

Progressive implementation

You don't need to add metadata to every page tomorrow. Start with high-value pages:

Week 1: Organization schema
Add to your homepage. One page, big impact.

Week 2: Article schema
Add to your most-trafficked blog posts or articles.

Week 3: Product schema
If you sell products, add to top-selling items.

Week 4: FAQ schema
Add to your FAQ or support pages.

Then expand gradually. Track which pages get mentioned by AI, and prioritize adding metadata to those.

The payoff

A client in the automotive sector added proper Schema.org markup to 20 high-traffic pages—product pages, service locations, FAQ section. Within three months:

- AI recommendations increased by 40% (tracked via referral sources)
- Google rich results appeared for 15 of the 20 pages
- Voice search traffic doubled (Alexa, Google Assistant queries)
- Support queries decreased as AI could answer common questions

The implementation took about a week of developer time. The ROI was clear.

What's next

We've covered content structure (Chapter 4) and metadata (Chapter 5). Together, these give AI the information it needs. But there's still the question of navigation—how AI discovers content and understands site organization.

Next chapter tackles navigation patterns, sitemaps, internal linking, and the overall information architecture that helps AI understand the relationship between your pages.

Coming up in Chapter 6: Navigation and discovery—helping AI find and understand your site structure.

Chapter 6: Navigation and Discovery

How AI discovers your content

Before AI can read your content, it needs to find it. This seems obvious but is often overlooked. You might have perfectly structured pages with excellent metadata, but if AI can't discover them or understand how they relate to each other, you need to optimize navigation and discovery.

AI discovers content through three main mechanisms:

1. Direct URL requests
Someone asks: "What does the Digital Domain Technologies service page say about Edge Delivery?" The AI already knows the URL (perhaps from previous searches or a sitemap) and fetches that specific page.

2. Following links
AI starts at your homepage (or a known page) and follows links to discover other content. This is how search engine crawlers work, and many AI agents use similar patterns.

3. Sitemaps
Your sitemap.xml provides a structured list of URLs. AI can parse this to understand your site's scope and priority pages.

Each mechanism has different requirements and failure modes. Let's look at how to support all three effectively.

Pattern: The XML sitemap

Your sitemap.xml tells AI (and search engines) what pages exist and how they're organized. This is the foundation of discoverability.

Minimal sitemap:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://www.example.com/</loc>
    <lastmod>2024-03-20</lastmod>
    <priority>1.0</priority>
  </url>
  <url>
    <loc>https://www.example.com/services</loc>
    <lastmod>2024-03-15</lastmod>
    <priority>0.8</priority>
  </url>
  <url>
```

```
    <loc>https://www.example.com/about</loc>
    <lastmod>2024-02-10</lastmod>
    <priority>0.6</priority>
  </url>
</urlset>
```

The elements that matter:

loc - The full URL (required)
lastmod - When the page was last changed (ISO 8601 date)
priority - Relative importance (0.0 to 1.0)
changefreq - How often it changes (daily, weekly, monthly, yearly)

Forget `changefreq`. Google ignores it, and it's too hard to maintain accurately. Focus on `lastmod` and `priority`.

Better sitemap with organization:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <!-- Homepage -->
  <url>
    <loc>https://www.example.com/</loc>
    <lastmod>2024-03-20</lastmod>
    <priority>1.0</priority>
  </url>

  <!-- Main sections -->
  <url>
    <loc>https://www.example.com/services</loc>
    <lastmod>2024-03-15</lastmod>
    <priority>0.9</priority>
  </url>
  <url>
    <loc>https://www.example.com/case-studies</loc>
    <lastmod>2024-03-18</lastmod>
    <priority>0.9</priority>
  </url>

  <!-- Service pages -->
  <url>
    <loc>https://www.example.com/services/edge-delivery</loc>
    <lastmod>2024-03-10</lastmod>
    <priority>0.8</priority>
  </url>
  <url>
    <loc>https://www.example.com/services/aem-consulting</loc>
    <lastmod>2024-03-12</lastmod>
    <priority>0.8</priority>
  </url>

  <!-- Blog posts -->
  <url>
    <loc>https://www.example.com/blog/migrating-to-eds</loc>
    <lastmod>2024-03-20</lastmod>
    <priority>0.7</priority>
  </url>

  <!-- Supporting pages -->
  <url>
    <loc>https://www.example.com/about</loc>
    <lastmod>2024-02-10</lastmod>
    <priority>0.5</priority>
  </url>
  <url>
    <loc>https://www.example.com/contact</loc>
    <lastmod>2024-01-15</lastmod>
    <priority>0.6</priority>
  </url>
</urlset>
```

Notice the priority gradation:

- Homepage: 1.0
- Main sections: 0.9
- Service pages: 0.8
- Blog content: 0.7
- Supporting pages: 0.5-0.6

This tells AI which pages matter most when it's deciding what to crawl or recommend.

Sitemap index for large sites:

If you have hundreds or thousands of pages, split into multiple sitemaps:

```
<?xml version="1.0" encoding="UTF-8"?>
<sitemapindex xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <sitemap>
    <loc>https://www.example.com/sitemap-pages.xml</loc>
    <lastmod>2024-03-20</lastmod>
  </sitemap>
  <sitemap>
    <loc>https://www.example.com/sitemap-posts.xml</loc>
    <lastmod>2024-03-20</lastmod>
  </sitemap>
  <sitemap>
    <loc>https://www.example.com/sitemap-products.xml</loc>
    <lastmod>2024-03-15</lastmod>
  </sitemap>
</sitemapindex>
```

Each individual sitemap should stay under 50,000 URLs and 50MB uncompressed.

Dynamic sitemap generation:

For sites with changing content, generate sitemaps dynamically:

```
// Example: Express.js sitemap route
app.get('/sitemap.xml', async (req, res) => {
  const pages = await getPages();
  const posts = await getBlogPosts();

  let xml = '<?xml version="1.0" encoding="UTF-8"?>\n';
  xml += '<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">\n';

  // Homepage
  xml += ' <url>\n';
  xml += `  <loc>https://www.example.com/</loc>\n`;
  xml += `  <lastmod>${new Date().toISOString().split('T')[0]}</lastmod>\n`;
  xml += `  <priority>1.0</priority>\n`;
  xml += ' </url>\n';

  // Dynamic pages
  pages.forEach(page => {
    xml += ' <url>\n';
    xml += `  <loc>https://www.example.com${page.url}</loc>\n`;
    xml += `  <lastmod>${page.modified.toISOString().split('T')[0]}</lastmod>\n`;
    xml += `  <priority>${page.priority || 0.8}</priority>\n`;
    xml += ' </url>\n';
  });

  // Blog posts
  posts.forEach(post => {
    xml += ' <url>\n';
    xml += `  <loc>https://www.example.com/blog/${post.slug}</loc>\n`;
    xml += `  <lastmod>${post.modified.toISOString().split('T')[0]}</lastmod>\n`;
    xml += `  <priority>0.7</priority>\n`;
    xml += ' </url>\n';
  });

  xml += '</urlset>';

  res.header('Content-Type', 'application/xml');
  res.send(xml);
});
```

This ensures your sitemap is always current without manual updates.

Link text matters

When AI follows links, the link text provides critical context. The href tells AI where the link goes, but the link text tells AI why it goes there and what to expect.

Bad link text:

```
<p>
  We offer Edge Delivery Services consulting.
  <a href="/services/eds">Click here</a> for more information.
</p>

<p>
  Read our latest blog post
  <a href="/blog/migration-guide">here</a>.
</p>

<nav>
  <a href="/services">Services</a>
  <a href="/services/web">More</a>
  <a href="/services/consulting">Details</a>
</nav>
```

Problems:

- "Click here" and "here" are meaningless out of context
- "More" and "Details" don't indicate what they link to
- AI can't build useful mental model from these links

Better link text:

```
<p>
  We offer Edge Delivery Services consulting.
  <a href="/services/eds">Learn about our EDS consulting services</a>.
</p>

<p>
  Read our latest blog post:
  <a href="/blog/migration-guide">Migrating from AEM Classic to Edge Delivery Services</a>.
</p>
```

```
<nav>
  <a href="/services">Our Services</a>
  <a href="/services/web">Web Development</a>
  <a href="/services/consulting">Strategic Consulting</a>
</nav>
```

Each link text:

- Describes the destination
- Makes sense when read aloud
- Provides value even without surrounding context

This matters because AI (and screen readers) often process links separately from surrounding text. The link text needs to be self-explanatory.

The link text test:

Extract all links from your page and read just the link text in a list. Does each one make sense? Can you tell where it goes?

Your page:

- Click here
- Read more
- Learn more
- See details
- More information

Better:

- EDS consulting services
- Migration guide: AEM to Edge Delivery
- Case study: Automotive industry transformation
- Contact our team
- Download pricing guide

The second list works without any surrounding context.

Internal linking strategy

Internal links serve two purposes for AI:

1. **Discovery:** AI finds new pages by following links
2. **Context:** Links establish relationships between topics

Your internal linking should create a semantic web that helps AI understand how your content relates.

Hub and spoke pattern:

```
Homepage
  +-- Services (hub page)
  |   +-- Edge Delivery Services
  |   +-- AEM Consulting
  |   +-- Training
  +-- Case Studies (hub page)
  |   +-- Automotive Industry
  |   +-- Financial Services
  |   +-- Retail Sector
  +-- Blog (hub page)
      +-- Latest Posts
      +-- Guides
      +-- Technical Articles
```

Each hub page links to its spokes. Each spoke links back to its hub and to related spokes.

Example hub page:

```
<main>
  <h1>Our Services</h1>
  <p>
    We provide consultancy across Adobe Experience Manager and
    Edge Delivery Services implementations.
  </p>

  <section>
    <h2>Edge Delivery Services</h2>
    <p>
      Modern web delivery using Adobe's Edge Delivery Services platform.
      <a href="/services/edge-delivery">Explore EDS consulting</a>
    </p>
  </section>

  <section>
    <h2>AEM Consulting</h2>
    <p>
      Strategic advisory for Adobe Experience Manager implementations.
      <a href="/services/aem-consulting">Learn about AEM consulting</a>
    </p>
  </section>

  <section>
    <h2>Training and Support</h2>
    <p>
      Upskill your team with hands-on training and ongoing support.
      <a href="/services/training">View training programmes</a>
    </p>
  </section>

  <aside>
    <h2>Related Resources</h2>
    <ul>
      <li><a href="/case-studies">View our case studies</a></li>
      <li><a href="/blog/guides">Read our implementation guides</a></li>
      <li><a href="/contact">Discuss your requirements</a></li>
    </ul>
  </aside>
</main>
```

This creates multiple paths for discovery:

- From homepage to services hub
- From services hub to specific services
- From services to case studies and guides
- Cross-links between related content

Contextual links in content:

```
<article>
  <h1>Migrating from AEM Classic to Edge Delivery Services</h1>

  <p>
```

```
    The migration process involves several stages. First, you'll need to
    <a href="/guides/content-audit">audit your existing content</a> to
    determine what can be migrated automatically versus what needs manual
    restructuring.
  </p>

  <p>
    Once your content is organized, the next step is
    <a href="/guides/block-development">developing custom blocks</a> for
    any unique functionality your site requires.
  </p>

  <p>
    Throughout this process, understanding
    <a href="/guides/eds-architecture">EDS architecture fundamentals</a>
    will help you make better decisions about implementation approach.
  </p>
</article>
```

These contextual links:

- Help AI understand topic relationships
- Provide natural discovery paths
- Show which topics are prerequisites or related concepts

Breadcrumb navigation

Breadcrumbs are more than a UX nicety—they tell AI exactly where a page sits in your information hierarchy.

Basic breadcrumbs:

```
<nav aria-label="Breadcrumb">
  <ol>
    <li><a href="/">Home</a></li>
    <li><a href="/services">Services</a></li>
    <li><a href="/services/edge-delivery">Edge Delivery Services</a></li>
    <li aria-current="page">Implementation Guide</li>
  </ol>
</nav>
```

With Schema.org markup:

```
<nav aria-label="Breadcrumb">
  <ol itemscope itemtype="https://schema.org/BreadcrumbList">
    <li itemprop="itemListElement" itemscope itemtype="https://schema.org/ListItem">
      <a itemprop="item" href="/">
        <span itemprop="name">Home</span>
      </a>
      <meta itemprop="position" content="1">
    </li>
    <li itemprop="itemListElement" itemscope itemtype="https://schema.org/ListItem">
      <a itemprop="item" href="/services">
        <span itemprop="name">Services</span>
      </a>
      <meta itemprop="position" content="2">
    </li>
    <li itemprop="itemListElement" itemscope itemtype="https://schema.org/ListItem">
      <a itemprop="item" href="/services/edge-delivery">
        <span itemprop="name">Edge Delivery Services</span>
      </a>
      <meta itemprop="position" content="3">
    </li>
    <li itemprop="itemListElement" itemscope itemtype="https://schema.org/ListItem">
      <span itemprop="name">Implementation Guide</span>
      <meta itemprop="position" content="4">
    </li>
  </ol>
</nav>
```

This creates explicit parent-child relationships. AI now knows:

- Implementation Guide is under Edge Delivery Services
- Edge Delivery Services is under Services
- Services is under Home

When someone asks "Does Digital Domain offer EDS consulting?" AI can navigate from the homepage to services to EDS to find detailed information.

Site structure patterns

How you organize your site affects AI's ability to understand it. Here are patterns that work:

Topical clusters:

```
/services/edge-delivery/
  - index.html (overview)
  - /implementation
  - /migration
  - /support
  - /case-studies

/services/aem-consulting/
  - index.html (overview)
  - /strategy
  - /architecture
  - /training
```

Each topic has a hub page with related sub-pages. The URL structure makes relationships explicit.

Content by type:

```
/blog/
  - /guides/ (how-to content)
  - /case-studies/ (client work)
  - /news/ (company updates)
  - /technical/ (deep dives)
```

Clear categorization helps AI understand content purpose.

Date-based for news:

```
/blog/2024/03/migration-guide
/blog/2024/02/release-notes
```

Date in URL signals recency. AI can prioritize recent content for time-sensitive queries.

Flat vs. nested:

```
<!-- Too flat -->
/services-edge-delivery-implementation
/services-edge-delivery-migration
/services-aem-consulting-strategy
```

```
<!-- Too nested -->
/services/consulting/adobe/experience-manager/implementation/enterprise/
```

Aim for 2-4 levels deep. Flat enough for clarity, nested enough to show hierarchy.

The navigation menu structure

Your main navigation creates the mental model AI builds of your site organization.

Simple but effective:

```
<nav aria-label="Main␣navigation">
  <ul>
    <li><a href="/">Home</a></li>
    <li>
      <a href="/services">Services</a>
      <ul>
        <li><a href="/services/edge-delivery">Edge Delivery Services</a></li>
        <li><a href="/services/aem-consulting">AEM Consulting</a></li>
        <li><a href="/services/training">Training</a></li>
      </ul>
    </li>
    <li><a href="/case-studies">Case Studies</a></li>
    <li><a href="/blog">Blog</a></li>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
  </ul>
</nav>
```

This tells AI:

- You have six main sections
- Services has three sub-categories
- The structure is relatively flat

With added semantic information:

```
<nav aria-label="Main␣navigation" role="navigation">
  <h2 class="visually-hidden">Main menu</h2>
  <ul>
    <li><a href="/" aria-current="page">Home</a></li>
    <li>
      <a href="/services" aria-expanded="false" aria-haspopup="true">Services</a>
      <ul aria-label="Services␣submenu">
        <li>
          <a href="/services/edge-delivery">
            Edge Delivery Services
            <span class="nav-description">Modern web delivery platform</span>
          </a>
        </li>
        <li>
          <a href="/services/aem-consulting">
            AEM Consulting
            <span class="nav-description">Strategic advisory and implementation</span>
          </a>
        </li>
        <li>
          <a href="/services/training">
            Training
            <span class="nav-description">Upskill your team</span>
          </a>
         </li>
       </ul>
     </li>
     <!-- Additional menu items -->
   </ul>
 </nav>
```

The `nav-description` class can be visually hidden but provides context for AI and screen readers. AI now knows what each service offering involves.

Footer navigation

Don't neglect footer links. They often contain important pages that aren't in main navigation:

```
<footer>
  <nav aria-label="Footer␣navigation">
    <h2>Quick Links</h2>
    <ul>
      <li><a href="/sitemap">Sitemap</a></li>
      <li><a href="/privacy">Privacy Policy</a></li>
      <li><a href="/terms">Terms of Service</a></li>
      <li><a href="/accessibility">Accessibility Statement</a></li>
    </ul>
  </nav>

  <nav aria-label="Legal␣information">
    <h2>Company</h2>
    <ul>
      <li><a href="/about/company">About Us</a></li>
      <li><a href="/about/team">Our Team</a></li>
      <li><a href="/careers">Careers</a></li>
      <li><a href="/contact">Contact</a></li>
    </ul>
  </nav>
</footer>
```

Notice:

- Separate `nav` elements with `aria-label` to distinguish them
- Headings for each section
- Links to utility pages (privacy, terms, etc.)

AI can now find your privacy policy, contact information, and other important pages even if they're not in main navigation.

Related content links

Within your content, suggest related pages:

```
<article>
  <h1>Migrating from AEM Classic to Edge Delivery Services</h1>
  <!-- Article content -->

  <aside class="related-content">
    <h2>Related Guides</h2>
    <ul>
      <li>
        <a href="/guides/eds-architecture">
          Understanding EDS Architecture
        </a>
        <p>Learn the fundamentals before migrating</p>
      </li>
      <li>
        <a href="/guides/content-modeling">
          Content Modeling for Edge Delivery
        </a>
        <p>Structure your content for optimal performance</p>
      </li>
      <li>
        <a href="/case-studies/automotive-migration">
          Case Study: Automotive Industry Migration
        </a>
        <p>See how we migrated a major automotive site</p>
      </li>
    </ul>
  </aside>
</article>
```

This provides:

- Discovery paths to related content
- Context about what each link offers
- Semantic grouping of related resources

AI can follow these links to build a more complete picture of your expertise and offerings.

The robots.txt consideration

Your robots.txt file tells crawlers what they can and can't access:

```
User-agent: *
Allow: /

Sitemap: https://www.example.com/sitemap.xml

# Block admin and utility pages
Disallow: /admin/
Disallow: /login/
Disallow: /checkout/
Disallow: /api/
```

Keep it simple. Only block:

- Admin interfaces
- Authentication pages
- Payment processing
- API endpoints not meant for crawling

Don't block:

- CSS or JavaScript files (some crawlers need them)
- Images (AI may need to fetch them)
- Public documentation
- Any content you want AI to discover

Common mistake:

```
User-agent: *
Disallow: /services/ # Oops, blocking all services pages
```

Test your robots.txt with Google's robots.txt Tester or similar tools before deploying.

Navigation and understanding

Good navigation doesn't just help AI find pages—it helps AI understand your business:

What you do: Clear service pages linked from main navigation
Your expertise: Case studies and guides showing depth
Your priorities: What you link to most prominently
Content relationships: How topics connect to each other

Poor navigation creates confusion:

- AI can't determine what you specialize in
- Pages seem unrelated to each other
- No clear entry points for different topics
- Important pages are hidden or hard to find

Good navigation creates clarity:

- Primary services are obvious
- Content relationships are explicit
- Entry points match common queries
- Everything is discoverable within 3-4 clicks from homepage

Testing navigation effectiveness

The discovery test:

1. Start at your homepage HTML
2. Extract all links
3. Follow each link and extract their links
4. Repeat for 3-4 levels
5. Map what's discoverable

Can AI reach all your important pages? Are there orphaned pages with no inbound links?

The context test:

Read all your link text in isolation. Does each link clearly indicate its destination and purpose?

The structure test:

Draw your site as a tree diagram based on navigation structure. Does the hierarchy make sense? Are there sections that seem misplaced?

Real-world example

I worked with a financial services client whose navigation was organized around internal departments:

```
- Corporate Solutions
- Retail Banking
- Investment Management
- Operations
```

This made sense internally but not to AI or customers. When someone asked "Does CompanyX offer mortgages?" AI couldn't find the information because mortgages were under "Retail Banking."

We restructured to customer-facing categories:

```
- Personal Banking
  - Current Accounts
  - Savings
  - Mortgages
  - Loans
- Business Banking
  - Business Accounts
  - Business Loans
  - Merchant Services
- Investments
  - ISAs
  - Pensions
  - Investment Portfolios
```

Same content, different organization. AI could now answer "Does CompanyX offer mortgages?" by navigating to Personal Banking > Mortgages.

The lesson: organize navigation around how people (and AI) think about your offerings, not around how your company is structured internally.

What's next

We've covered how AI discovers content and navigates sites. But even well-structured, easily discovered content can be undermined by one thing: JavaScript that hides content from parsers.

The next chapter tackles the JavaScript challenge—when client-side rendering helps, when it hurts, and how to make JavaScript-heavy sites work for AI.

Coming up in Chapter 7: The JavaScript challenge—making modern web apps readable by AI without sacrificing interactivity.

Chapter 7: The JavaScript Challenge

The rendering divide

Modern web development loves JavaScript frameworks. React, Vue, Angular, Svelte—they enable rich interactivity and smooth user experiences. But they've created a problem: content that only exists after JavaScript executes.

This matters because, as we established in Chapter 2, not all AI agents execute JavaScript. Some fetch raw HTML and parse it directly. Others run headless browsers and execute everything. A third category uses vision models that see rendered output.

Your site needs to work for all three. That's the JavaScript challenge.

What gets lost

Let me show you a real example from a client project. Their homepage looked like this to humans:

[**Hero section with company name**]
Leading provider of sustainable energy solutions
[**Three service cards with icons and descriptions**]
[**Client logos grid**]
[**Contact form**]

Beautiful. Interactive. Completely invisible to parsers.

Here's what the HTML source contained:

```
<!DOCTYPE html>
<html>
<head>
  <title>EnergyTech Solutions</title>
  <script src="/bundle.js"></script>
</head>
<body>
  <div id="root"></div>
</body>
</html>
```

That's it. Everything else—the company description, the services, the contact form—was generated by JavaScript. The `bundle.js` file fetched data from an API, built the DOM, and rendered the content.

To a parser reading raw HTML: an empty page.

When someone asked an AI “What does EnergyTech Solutions do?” the AI couldn’t answer. The information existed, but only after JavaScript execution.

The spectrum of JavaScript usage

Not all JavaScript is problematic. Let’s look at the spectrum:

Harmless JavaScript:

- Visual effects (animations, transitions)
- User interactions (dropdown menus, modals)
- Progressive enhancements (form validation, autocomplete)
- Analytics and tracking

These don’t affect content visibility. The underlying HTML contains all the information; JavaScript just makes it more interactive.

Problematic JavaScript:

- Content loaded via AJAX after page load
- Entire page rendered client-side from JSON
- Navigation that updates content without changing URL
- Information hidden behind tabs that use `display: none`

These hide content from parsers or make it hard to discover.

The critical question:

If JavaScript fails or doesn’t execute, is your content still readable?

Testing without JavaScript

The simplest test:

1. **Visit your page**
2. **Disable JavaScript** (Developer Tools → Settings → Disable JavaScript)
3. **Refresh the page**

What do you see?

If the answer is “nothing” or “a blank page” or “a loading spinner that never completes,” you have a problem.

If the answer is “the content, just without interactivity,” you’re in good shape.

The view source test:

View page source (right-click → View Page Source). Can you read your content in the HTML? Not the rendered version—the actual source code.

If your content is there, parsers can read it. If it’s not there, only browser-based agents can access it.

Progressive enhancement: the solution

Progressive enhancement is an old idea that’s newly relevant: start with working HTML, enhance with CSS and JavaScript.

Base layer (HTML): Content and structure
Enhancement layer (CSS): Visual presentation
Interaction layer (JavaScript): Enhanced behaviour

Each layer works without the next. JavaScript adds polish, but the foundation is solid HTML that works on its own.

Example: Product cards**

Client-side only approach:

```
<div id="products-container"></div>

<script>
  fetch('/api/products')
    .then(res => res.json())
    .then(products => {
      const html = products.map(p => `
        <div class="product-card">
          <h3>${p.name}</h3>
          <p>${p.description}</p>
          <span class="price">£${p.price}</span>
        </div>
      `).join('');
      document.getElementById('products-container').innerHTML = html;
    });
</script>
```

Parser sees: empty div.

Progressive enhancement approach:

```
<section id="products-container">
  <article class="product-card">
    <h3>Standing Desk Pro</h3>
    <p>Electric height-adjustable desk with memory presets.</p>
    <data class="price" value="599">£599</data>
  </article>

  <article class="product-card">
    <h3>Ergonomic Chair</h3>
    <p>Fully adjustable office chair with lumbar support.</p>
    <data class="price" value="399">£399</data>
  </article>

  <article class="product-card">
    <h3>Monitor Arm</h3>
    <p>Dual monitor arm with gas spring adjustment.</p>
    <data class="price" value="129">£129</data>
  </article>
</section>

<script>
  // JavaScript can still enhance this with filtering, sorting, etc.
  // But the base content is already in the HTML
</script>
```

Parser sees: three products with names, descriptions, and prices.

The JavaScript can still add functionality:

- Filter by price range
- Sort by different criteria

- Add to cart without page reload
- Show/hide out of stock items

But the foundation works without it.

Server-side rendering (SSR)

If you're using a JavaScript framework and want the benefits of both approaches, server-side rendering is the answer.

How it works:

1. Server executes your JavaScript framework
2. Generates HTML with content
3. Sends complete HTML to browser
4. Browser loads JavaScript
5. JavaScript "hydrates" the page (adds interactivity to existing HTML)

Traditional client-side rendering:

```
Browser requests page→
  Server sends minimal HTML + JavaScript→
  Browser downloads JavaScript→
  JavaScript executes→
  JavaScript fetches data→
  JavaScript builds DOM→
  Page visible
```

Server-side rendering:

```
Browser requests page→
  Server executes JavaScript→
  Server fetches data→
  Server generates HTML→
  Server sends complete HTML→
  Page visible→
  Browser loads JavaScript→
  JavaScript adds interactivity
```

The page is visible faster for humans, and the content is present in HTML for parsers.

Next.js example:

```
// pages/products.js
export async function getServerSideProps() {
  // Fetch data at request time on the server
  const res = await fetch('https://api.example.com/products');
  const products = await res.json();

  return {
    props: { products }
  };
}

export default function Products({ products }) {
  return (
    <section>
      {products.map(product => (
        <article key={product.id}>
          <h3>{product.name}</h3>
          <p>{product.description}</p>
          <data value={product.price}>£{product.price}</data>
        </article>
      ))}
    </section>
  );
}
```

The HTML sent to the browser contains all product information. Parsers can read it. JavaScript then adds interactive features.

Static site generation (SSG)

For content that doesn't change frequently, generate HTML at build time:

```
// Build-time generation
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/products');
  const products = await res.json();

  return {
    props: { products },
    revalidate: 3600 // Regenerate every hour
  };
}
```

This produces static HTML files with all content baked in. Perfect for AI readability, fast for users, simple to deploy.

Handling dynamic content

Some content genuinely needs to be dynamic—user-specific data, real-time information, personalised experiences. How do you make this AI-readable?

Pattern 1: Skeleton content

Include placeholder or default content in HTML, replace with personalised content via JavaScript:

```
<section class="user-dashboard">
  <h1>Welcome back</h1>
  <div class="account-summary">
    <p>Account balance: <data class="balance">Loading...</data></p>
    <p>Last login: <time class="last-login">Loading...</time></p>
  </div>
</section>

<script>
  // Fetch user data and update the existing elements
  fetch('/api/user/dashboard')
    .then(res => res.json())
    .then(data => {
      document.querySelector('.balance').textContent = `£${data.balance}`;
      document.querySelector('.last-login').textContent = data.lastLogin;
    });
</script>
```

AI sees the structure and knows this is an account dashboard, even if the specific values aren't available.

Pattern 2: Separate public and private content

Keep AI-relevant content in static HTML. Make user-specific content a separate concern:

```
<!-- Public product information (in HTML) -->
<article>
  <h1>Professional Standing Desk</h1>
  <p>Electric height-adjustable desk with memory presets.</p>
  <data value="599">£599</data>

  <dl>
    <dt>Dimensions</dt>
    <dd>120cm × 60cm</dd>

    <dt>Weight Capacity</dt>
    <dd>80kg</dd>
  </dl>
</article>

<!-- User-specific content (loaded via JavaScript) -->
<aside id="user-context">
  <!-- Populated with: "You␣viewed␣this␣3␣times", "In␣your␣saved␣items", etc. -->
</aside>
```

The product information is available to AI. The personalisation is added for logged-in users but doesn't affect discoverability.

Single-page applications (SPAs)

SPAs are particularly challenging. The entire application runs in JavaScript, changing content without page loads. URLs might not change, or they might use hash fragments that aren't sent to servers.

The problem:

```
<!-- URL: https://example.com/#/products/desk-pro -->
<!DOCTYPE html>
<html>
<head>
  <title>Our Shop</title>
</head>
<body>
  <div id="app"></div>
  <script src="/app.js"></script>
</body>
</html>
```

Every URL in your SPA returns this same HTML. AI can't distinguish between the homepage, product pages, or the about page.

Solution 1: Use proper URLs (no hash fragments)

Modern routers support proper URLs with the History API:

```
// Bad: Hash-based routing
https://example.com/#/products/desk-pro

// Good: History API routing
https://example.com/products/desk-pro
```

Solution 2: Server-side rendering for public routes

Render public-facing pages server-side, use SPA for authenticated areas:

```
// routes/products/[id].js
export async function getServerSideProps({ params }) {
  const product = await fetchProduct(params.id);

  return {
    props: { product }
  };
}

export default function ProductPage({ product }) {
  return (
    <main>
      <h1>{product.name}</h1>
      <p>{product.description}</p>
      {/* Product details */}
    </main>
  );
}
```

Public product pages get SSR (AI can read them). Your admin dashboard can still be a full SPA.

Solution 3: Pre-rendering

Generate static HTML for each route at build time:

```
# Using a pre-rendering tool
npx react-snap
```

This crawls your SPA and generates static HTML files for each route. Deploy these alongside your JavaScript app.

Infinite scroll and pagination

Infinite scroll is problematic for AI because:

- Initial HTML only contains first page of results
- Additional content loads as user scrolls
- No way for AI to access page 2, 3, etc. without scrolling

The fix: Hybrid approach

Provide traditional pagination for AI, enhance with infinite scroll for humans:

```
<section class="product-list">
  <!-- Products 1-20 in HTML -->
  <article>Product 1</article>
  <article>Product 2</article>
  <!-- ... -->
  <article>Product 20</article>
</section>

<nav class="pagination" aria-label="Product␣pages">
  <a href="/products?page=1" aria-current="page">1</a>
  <a href="/products?page=2">2</a>
  <a href="/products?page=3">3</a>
  <a href="/products?page=4">4</a>
  <a href="/products?page=5">5</a>
</nav>

<script>
  // Progressive enhancement: replace pagination with infinite scroll
  // But keep pagination links functional for AI and no-JS users
</script>
```

AI can follow pagination links to discover all products. JavaScript can replace this with infinite scroll for a better user experience.

Modal dialogs and hidden content

Content in modals or hidden behind tabs is discoverable if it's in the HTML:

```
<div class="tab-container">
  <nav class="tabs" role="tablist">
    <button role="tab" aria-selected="true" aria-controls="specs">Specifications</button>
    <button role="tab" aria-selected="false" aria-controls="reviews">Reviews</button>
    <button role="tab" aria-selected="false" aria-controls="shipping">Shipping</button>
  </nav>

  <section id="specs" role="tabpanel">
    <h2>Technical Specifications</h2>
    <dl>
      <dt>Dimensions</dt>
      <dd>120cm × 60cm × 75-120cm (adjustable height)</dd>
      <!-- More specs -->
    </dl>
  </section>

  <section id="reviews" role="tabpanel" hidden>
    <h2>Customer Reviews</h2>
    <!-- Reviews content -->
  </section>

  <section id="shipping" role="tabpanel" hidden>
    <h2>Shipping Information</h2>
    <!-- Shipping content -->
  </section>
</div>
```

All three tabs exist in the HTML. They're just hidden with CSS or the `hidden` attribute. AI can read all the content.

The `display: none` concern:

Some people worry that `display: none` content might be ignored by crawlers. In practice:

- Google explicitly states they process hidden content
- Screen readers handle it correctly with proper ARIA
- The content is in the HTML, so parsers can access it

The key: don't hide content to deceive (keyword stuffing in hidden divs). Hide content to improve UX (tabs, accordions).

Loading states and fallbacks

When JavaScript loads content asynchronously, provide meaningful fallback content:

Bad approach:

```
<div class="product-reviews">
  <div class="spinner"></div>
</div>
```

Parser sees: a div with a spinner div. No indication what this section is for.

Better approach:

```
<section class="product-reviews" aria-busy="true">
  <h2>Customer Reviews</h2>
  <p>Loading reviews...</p>
  <noscript>
    <p>Please enable JavaScript to view customer reviews, or
      <a href="/products/desk-pro/reviews">view reviews page</a>.</p>
  </noscript>
</section>

<script>
  loadReviews().then(reviews => {
    // Replace loading message with actual reviews
    // Remove aria-busy attribute
  });
</script>
```

Now:

- The section has a heading indicating its purpose
- There's fallback text explaining what should load
- `noscript` provides an alternative for no-JS browsers
- AI understands this section contains reviews

AJAX navigation

Sites that fetch content via AJAX and update the page without reloading need special handling:

The pattern:

```
<!-- Initial page load: full HTML -->
<main id="content">
  <article>
    <h1>Welcome to Our Blog</h1>
    <p>Latest posts about web development...</p>
  </article>
</main>

<nav>
  <a href="/blog/javascript-tips" data-ajax="true">JavaScript Tips</a>
  <a href="/blog/css-tricks" data-ajax="true">CSS Tricks</a>
</nav>

<script>
  // Enhance links with AJAX
  document.querySelectorAll('[data-ajax]').forEach(link => {
    link.addEventListener('click', async (e) => {
      e.preventDefault();

      // Fetch new content
      const response = await fetch(link.href);
      const html = await response.text();
      // Update content area
      const parser = new DOMParser();
      const doc = parser.parseFromString(html, 'text/html');
      document.getElementById('content').innerHTML =
        doc.getElementById('content').innerHTML;

      // Update URL
      history.pushState({}, '', link.href);
    });
  });
</script>
```

This works because:

- Each link has a real URL
- The URL returns full HTML (not just a fragment)
- JavaScript intercepts clicks to make it smoother
- If JavaScript fails, links still work

The server response:

When `/blog/javascript-tips` is requested, return full HTML:

```
<!DOCTYPE html>
<html>
<head>
  <title>JavaScript Tips | Blog</title>
</head>
<body>
  <main id="content">
    <article>
      <h1>JavaScript Tips</h1>
      <p>Here are some useful JavaScript patterns...</p>
    </article>
  </main>
  <!-- Navigation, footer, etc. -->
</body>
</html>
```

Don’t return just the `<article>` fragment. Return the whole page. JavaScript can extract what it needs, but crawlers get complete context.

Testing JavaScript-dependent sites

Test 1: Disable JavaScript

We covered this earlier, but it’s worth repeating: this is your first test. If the site is unusable without JavaScript, you need progressive enhancement.

Test 2: Network throttling

Slow down your connection (Developer Tools → Network → Throttling → Slow 3G). Watch what happens:

- How long until content appears?
- What’s visible before JavaScript loads?
- Are there layout shifts as content loads?

If the page is blank for 5+ seconds on slow connections, AI might time out before content appears.

Test 3: The HTML snapshot test

1. Visit your page
2. View source (the actual HTML sent by the server)
3. Save it as an HTML file
4. Open that file in a browser

Does it look anything like your actual site? If it's blank or broken, that's what parsers see.

Test 4: Headless browser simulation

Use Puppeteer or Playwright to simulate what a browser-based AI agent sees:

```
const puppeteer = require('puppeteer');

(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();

  await page.goto('https://example.com/products/desk-pro');

  // Wait for content to load
  await page.waitForSelector('h1');

  // Extract content
  const content = await page.evaluate(() => {
    const heading = document.querySelector('h1')?.textContent;
    const description = document.querySelector('article p')?.textContent;
    const price = document.querySelector('[data-price]')?.textContent;

    return { heading, description, price };
  });

  console.log(content);
  await browser.close();
})();
```

This shows what a JavaScript-executing crawler can extract from your page.

Real-world example: The e-commerce rebuild

A client had a React-based e-commerce site. Beautiful interface, terrible AI visibility. Product pages returned:

```
<!DOCTYPE html>
<html>
<head>
  <title>Shop</title>
</head>
<body>
  <div id="root"></div>
  <script src="/bundle.js"></script>
</body>
</html>
```

Every product page. Same HTML.

When someone asked "What's the price of the standing desk at CompanyX?" AI couldn't answer. The price existed, but only in JavaScript.

We rebuilt with Next.js and SSR:

```
// pages/products/[slug].js
export async function getStaticPaths() {
  const products = await fetchAllProducts();
  const paths = products.map(p => ({
    params: { slug: p.slug }
  }));

  return { paths, fallback: 'blocking' };
}

export async function getStaticProps({ params }) {
  const product = await fetchProduct(params.slug);

  return {
    props: { product },
    revalidate: 3600 // Regenerate every hour
  };
}

export default function ProductPage({ product }) {
  return (
    <main itemScope itemType="https://schema.org/Product">
      <h1 itemProp="name">{product.name}</h1>
      <img itemProp="image" src={product.image} alt={product.name} />
      <p itemProp="description">{product.description}</p>

      <div itemProp="offers" itemScope itemType="https://schema.org/Offer">
        <data itemProp="price" value={product.price}>
          £{product.price}
        </data>
        <meta itemProp="priceCurrency" content="GBP" />
      </div>

      {/* Rest of product page */}
    </main>
  );
}
```

Now each product page had:

- Unique URL
- Complete HTML with product information
- Schema.org markup
- All content visible to parsers

The site still felt like a SPA to users (fast navigation, smooth transitions), but every page was fully formed HTML that AI could read.

Results after three months:

- AI recommendations increased by 60%
- Organic search traffic up 35%
- Voice search queries (requiring structured data) doubled

Same visual design, same user experience, completely different AI visibility.

The hybrid approach summary

The pattern that works:

1. **Start with HTML** - Content in the initial response

2. **Add CSS** - Visual presentation
3. **Enhance with JavaScript** - Interactivity and improved UX
4. **Use SSR/SSG when possible** - For public content
5. **Provide fallbacks** - For when JavaScript fails or doesn't execute
6. **Test without JavaScript** - Regularly

This isn't "JavaScript is bad." It's "JavaScript should enhance, not replace, HTML."

Your site can be modern, interactive, and framework-based while still being AI-readable. You just need to ensure the foundation—the HTML—contains the content.

What's next

We've covered content structure, metadata, navigation, and JavaScript. These are the building blocks of AI readability.

The next chapter brings it all together with testing methodologies. How do you verify your site works for AI? What tools can you use? How do you catch problems before they affect your visibility?

Coming up in Chapter 8: Testing with AI agents—practical methods for validating AI readability.

Chapter 8: Testing with AI Agents

Why testing matters

You can implement perfect semantic HTML, comprehensive Schema.org markup, and thoughtful navigation—but if you don't test, you won't know if it actually works.

Testing AI readability is different from traditional testing. You're not checking if buttons click or forms submit. You're checking if AI can understand, extract, and use your content to answer questions.

The good news: testing AI readability is simpler than you might think. You don't need specialist tools or expensive services. You need a methodical approach and some basic techniques.

The morning-after test

This is your first and most important test. I mentioned it briefly in Chapter 1—now let's look at it properly.

The basic version:

1. Open your page in a browser
2. Right-click → View Page Source (not Inspect Element, the actual source)
3. Select all and copy the HTML
4. Open ChatGPT, Claude, or another LLM
5. Paste the HTML
6. Ask questions about the page

Questions to ask:

- "What is this page about?"
- "What products or services are offered?"
- "What actions can I take on this page?"
- "What is the price of [specific item]?"
- "Who is the contact person or company?"
- "When was this content last updated?"

The AI should answer accurately using only the HTML you pasted. If it can't, you've found a problem.

Example conversation:

```
You: [paste HTML of product page]

You: What is this page about?

AI: This appears to be a product page for a standing desk called "Professional Standing Desk".
    ↪ It's an electric height-adjustable desk priced at £599.
```

```
You: What are the main features?

AI: The key features include:
- Electric height adjustment with memory presets
- Whisper-quiet motor
- Cable management system
- Dimensions: 120cm × 60cm with adjustable height from 70cm to 120cm
- Weight capacity: 80kg

You: Is it in stock?

AI: Yes, according to the page it is currently in stock.
```

This tells you the AI can extract basic information. Now test edge cases:

```
You: What's the warranty period?

AI: I don't see warranty information in the HTML provided.
```

That’s a problem. You offer a warranty, but it’s not in the HTML—perhaps it’s loaded via JavaScript or hidden in a modal.

The advanced version:

Test with multiple AI systems:

- ChatGPT (OpenAI)
- Claude (Anthropic)
- Gemini (Google)

Each processes content slightly differently. If all three can extract the information, you’re in good shape. If one struggles, it might indicate an edge case worth addressing.

LLM-based extraction testing

Go beyond simple questions. Test if AI can extract structured information:

Test 1: Data extraction

```
You: [paste product page HTML]

You: Extract the following information in JSON format:
- Product name
- Price
- Availability
- Main features (as an array)

AI: {
  "productName": "Professional Standing Desk",
  "price": "599",
  "currency": "GBP",
  "availability": "In Stock",
  "features": [
    "Electric height adjustment",
    "Memory presets",
    "Cable management",
    "Whisper-quiet motor"
  ]
}
```

If the AI can do this accurately, your structured data is working. If it gets confused or misses information, you need to improve your markup.

Test 2: Relationship understanding

```
You: [paste service page HTML]

You: What services does this company offer, and how are they organized?

AI: The company offers three main service categories:

1. Edge Delivery Services
   - Implementation
   - Migration
   - Support

2. AEM Consulting
   - Strategy
   - Architecture
   - Training

3. Training and Support
   - Workshops
   - Ongoing support
   - Documentation

The page uses a clear hierarchy with h2 headings for main services and h3 headings for sub-
    ↪ services.
```

This tests whether your heading hierarchy and content organization are clear.

Test 3: Context awareness

```
You: [paste blog post HTML]

You: Who wrote this article and when was it published?

AI: The article was written by Tom Cranstoun and published on 15 March 2024. It was last
    ↪ modified on 20 March 2024.

You: Is this person qualified to write about this topic?

AI: Based on the author bio in the HTML, Tom Cranstoun is described as "Principal Consultant at
    ↪  Digital Domain Technologies with 25 years of CMS experience", which suggests relevant
    ↪ qualifications for the topic of AEM migration.
```

This checks if contextual information (author, dates, credentials) is properly marked up.

Headless browser testing

Some AI agents use headless browsers. Simulate this approach with Puppeteer or Playwright:

Basic Puppeteer test:

```
const puppeteer = require('puppeteer');

async function testPageReadability(url) {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();

  // Navigate to page
  await page.goto(url, { waitUntil: 'networkidle0' });

  // Extract key content
  const content = await page.evaluate(() => {
    // Get main heading
    const h1 = document.querySelector('h1')?.textContent?.trim();

    // Get all headings to understand structure
    const headings = Array.from(document.querySelectorAll('h1, h2, h3'))
      .map(h => ({
        level: h.tagName,
        text: h.textContent.trim()
      }));

    // Check for Schema.org data
    const schemaScripts = Array.from(
      document.querySelectorAll('script[type="application/ld+json"]')
    ).map(s => {
      try {
        return JSON.parse(s.textContent);
      } catch (e) {
        return null;
      }
    }).filter(Boolean);

    // Get all links
    const links = Array.from(document.querySelectorAll('a'))
      .map(a => ({
        text: a.textContent.trim(),
        href: a.href
      }))
      .filter(l => l.text && l.href);

    return {
      title: document.title,
      h1,
      headings,
      schemaData: schemaScripts,
      linkCount: links.length,
      sampleLinks: links.slice(0, 5)
    };
  });

  await browser.close();
  return content;
}

// Run the test
testPageReadability('https://example.com/products/desk-pro')
  .then(result => {
    console.log('Page Title:', result.title);
    console.log('Main Heading:', result.h1);
    console.log('\nHeading Structure:');
    result.headings.forEach(h => {
      const indent = ' '.repeat(parseInt(h.level[1]) - 1);
      console.log(`${indent}${h.level}: ${h.text}`);
    });
    console.log('\nSchema.org Data:', result.schemaData.length, 'items found');
    console.log('\nLinks:', result.linkCount, 'total');
    console.log('Sample links:', result.sampleLinks);
  })
  .catch(console.error);
```

This simulates what a browser-based AI agent sees after JavaScript execution.

What to check:

- Is the h1 meaningful?
- Do headings create a logical outline?
- Is Schema.org data present and valid?
- Are links descriptive?

Red flags:

```
// Bad signs in output
{
  title: "Loading...",
  h1: null,
  headings: [],
  schemaData: [],
  linkCount: 3
}
```

This suggests JavaScript hasn't finished loading or the page structure is broken.

Structured data validation

Google provides free tools for validating Schema.org markup:

Rich Results Test:
https://search.google.com/test/rich-results

Paste your URL or HTML snippet. Google will:

- Parse Schema.org markup
- Show what rich results are eligible
- Flag errors and warnings

Common errors caught:

```
Error: Missing required field "price"
Warning: Recommended field "image" is missing
Error: Invalid date format - use ISO 8601
```

Schema.org Validator:
https://validator.schema.org/

More permissive than Google's tool. Good for checking if your JSON-LD is syntactically valid, even if it won't trigger rich results.

Command-line validation:

For automated testing, use schema-dts or similar tools:

```
const { validate } = require('schema-dts');

const schema = {
  "@context": "https://schema.org",
  "@type": "Product",
  "name": "Standing Desk",
  "price": "599" // Should be in offers
};

const errors = validate(schema);
if (errors.length > 0) {
  console.log('Schema errors:', errors);
}
```

The comparison test

Before and after testing shows the impact of your improvements:

Before optimization:

```
Test: Morning-after with product page HTML

Question: What is the price?
Answer: I cannot determine the price from the HTML provided.

Question: Is it in stock?
Answer: I don't see availability information.

Question: What are the key features?
Answer: I can see some text that appears to describe features, but they're not clearly
    ↪ structured.
```

After optimization:

```
Test: Morning-after with optimized product page HTML

Question: What is the price?
Answer: £599

Question: Is it in stock?
Answer: Yes, currently in stock.

Question: What are the key features?
Answer: The main features are:
- Electric height adjustment with memory presets
- Cable management system
- Whisper-quiet motor
- Weight capacity of 80kg
- Dimensions: 120cm × 60cm, height adjustable from 70cm to 120cm
```

Document this difference. It shows stakeholders the concrete impact of AI optimization work.

Automated testing suite

Create a simple test suite you can run regularly:

```
// test-ai-readability.js
const puppeteer = require('puppeteer');

const tests = [
  {
    url: 'https://example.com/',
    checks: {
      hasH1: true,
      hasSchema: true,
      hasNav: true,
      minLinks: 5
    }
  },
  {
    url: 'https://example.com/products/desk-pro',
    checks: {
      hasH1: true,
      hasSchema: true,
      hasProductPrice: true,
      hasProductImage: true
    }
  }
];

async function runTests() {
  const browser = await puppeteer.launch();
  const results = [];

  for (const test of tests) {
    const page = await browser.newPage();
    await page.goto(test.url, { waitUntil: 'networkidle0' });

    const pageData = await page.evaluate(() => ({
      h1: document.querySelector('h1')?.textContent,
      hasNav: !!document.querySelector('nav'),
      linkCount: document.querySelectorAll('a').length,
      schemaCount: document.querySelectorAll('script[type="application/ld+json"]').length,
      hasProductPrice: !!document.querySelector('[itemprop="price"], [data-price]'),
      hasProductImage: !!document.querySelector('[itemprop="image"]')
    }));

    const testResult = {
      url: test.url,
      passed: true,
      failures: []
    };

    // Check conditions
    if (test.checks.hasH1 && !pageData.h1) {
      testResult.passed = false;
      testResult.failures.push('Missing h1 element');
    }

    if (test.checks.hasSchema && pageData.schemaCount === 0) {
      testResult.passed = false;
      testResult.failures.push('No Schema.org markup found');
    }

    if (test.checks.hasNav && !pageData.hasNav) {
      testResult.passed = false;
      testResult.failures.push('No nav element found');
    }

    if (test.checks.minLinks && pageData.linkCount < test.checks.minLinks) {
      testResult.passed = false;
      testResult.failures.push(`Only ${pageData.linkCount} links found, expected ${test.checks.
          ↪ minLinks}+`);
    }

    if (test.checks.hasProductPrice && !pageData.hasProductPrice) {
      testResult.passed = false;
      testResult.failures.push('Product price not found in expected format');
    }

    if (test.checks.hasProductImage && !pageData.hasProductImage) {
      testResult.passed = false;
      testResult.failures.push('Product image not properly marked up');
    }
    results.push(testResult);
    await page.close();
  }

  await browser.close();
  return results;
}

runTests().then(results => {
  console.log('\nAI Readability Test Results:\n');

  let allPassed = true;
  results.forEach(result => {
    const status = result.passed ? '[PASS]' : '[FAIL]';
    console.log(`${status}: ${result.url}`);

    if (!result.passed) {
      allPassed = false;
      result.failures.forEach(failure => {
        console.log(` - ${failure}`);
      });
    }
  });

  console.log('\n' + (allPassed ? 'All tests passed!' : 'Some tests failed.'));
  process.exit(allPassed ? 0 : 1);
});
```

Run this as part of your deployment process or on a schedule:

```
node test-ai-readability.js
```

The sitemap coverage test

Check if AI can discover all your important pages:

```
const axios = require('axios');
const xml2js = require('xml2js');

async function testSitemapCoverage() {
  // Fetch sitemap
  const response = await axios.get('https://example.com/sitemap.xml');
  const parser = new xml2js.Parser();
  const result = await parser.parseStringPromise(response.data);

  const urls = result.urlset.url.map(u => u.loc[0]);

  console.log(`Found ${urls.length} URLs in sitemap`);

  // Check a sample
  const sampleSize = Math.min(10, urls.length);
  const sample = urls.slice(0, sampleSize);

  for (const url of sample) {
    const pageResponse = await axios.get(url);
    const hasContent = pageResponse.data.includes('<h1');
    const hasSchema = pageResponse.data.includes('application/ld+json');

    console.log(`${url}:`);
    console.log(` - Has h1: ${hasContent}`);
    console.log(` - Has Schema.org: ${hasSchema}`);
  }
}

testSitemapCoverage().catch(console.error);
```

This verifies:

- Your sitemap is accessible
- Listed URLs are reachable
- Each page has basic semantic structure

The link text audit

Extract and review all link text from a page:

```
async function auditLinkText(url) {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();
  await page.goto(url, { waitUntil: 'networkidle0' });

  const links = await page.evaluate(() => {
    return Array.from(document.querySelectorAll('a'))
      .map(a => ({
        text: a.textContent.trim(),
        href: a.href,
        ariaLabel: a.getAttribute('aria-label')
      }))
      .filter(l => l.text || l.ariaLabel);
  });

  await browser.close();

  // Flag problematic link text
  const problematic = links.filter(l => {
    const text = l.text.toLowerCase();
    return (
      text === 'click here' ||
      text === 'here' ||
      text === 'read more' ||
      text === 'learn more' ||
      text === 'more' ||
      text.length < 3
    );
  });

  console.log(`Total links: ${links.length}`);
  console.log(`Problematic links: ${problematic.length}`);

  if (problematic.length > 0) {
    console.log('\nLinks to improve:');
    problematic.forEach(l => {
      console.log(` "${l.text}" →${l.href}`);
    });
  }

  return {
    total: links.length,
    problematic: problematic.length,
    issues: problematic
  };
}

auditLinkText('https://example.com').catch(console.error);
```

Mobile AI readability

Some AI agents access sites via mobile user agents. Test how your content appears on mobile:

```
async function testMobileReadability(url) {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();

  // Emulate mobile device
  await page.emulate({
    viewport: {
      width: 375,
      height: 667,
      isMobile: true
    },
    userAgent: 'Mozilla/5.0 (iPhone; CPU iPhone OS 14_0 like Mac OS X) AppleWebKit/605.1.15'
  });

  await page.goto(url, { waitUntil: 'networkidle0' });

  // Check content visibility
  const content = await page.evaluate(() => {
    const h1 = document.querySelector('h1');
    const main = document.querySelector('main');

    return {
      h1Visible: h1 ? h1.offsetHeight > 0 : false,
      mainVisible: main ? main.offsetHeight > 0 : false,
      h1Text: h1?.textContent?.trim(),
      viewportHeight: window.innerHeight,
      contentHeight: document.body.scrollHeight
    };
  });

  await browser.close();
  return content;
}
```

Check if:

- Content is visible on mobile viewports
- Headings aren’t hidden
- Main content isn’t pushed below the fold by oversized headers

The accessibility-AI overlap test

Many accessibility tools catch AI readability issues:

axe DevTools:
Browser extension that audits accessibility. Issues it catches that also affect AI:

- Missing alt text on images
- Improper heading hierarchy
- Forms without labels
- Links without descriptive text
- Missing ARIA labels

WAVE:
https://wave.webaim.org/

Visual accessibility checker. Highlights:

- Semantic structure problems
- Missing form labels
- Contrast issues (not AI-relevant, but good to fix)
- Document outline problems

Lighthouse:
Built into Chrome DevTools (Lighthouse tab).

The "Accessibility" section flags many AI readability issues:

- Image elements do not have alt attributes
- Heading elements are not in a sequentially-descending order
- Form elements do not have associated labels

Run Lighthouse, fix accessibility issues, and you'll improve AI readability as a side effect.

Continuous monitoring

Don't just test once. Set up ongoing monitoring:

Weekly automated tests:

```
// In your CI/CD pipeline or cron job
const schedule = require('node-schedule');

schedule.scheduleJob('0 9 * * 1', async function() {
  console.log('Running weekly AI readability tests...');

  const results = await runTests();

  if (!results.allPassed) {
    // Send alert email or Slack message
    await sendAlert({
      subject: 'AI Readability Tests Failed',
      results: results
    });
  }
});
```

Google Search Console monitoring:

Check for:

- Structured data errors (increases over time)
- Pages with missing or invalid markup
- Crawl errors that might hide content from AI

Traffic pattern monitoring:

Track:

- Referrals from AI chat interfaces (if trackable)
- Increases/decreases in organic search
- Voice search queries (often use AI extraction)

Creating a testing checklist

For each important page type, maintain a checklist:

Product Page Checklist:

- ☐ Has meaningful h1
- ☐ Product name visible in HTML (not just JavaScript)
- ☐ Price visible in HTML with proper markup
- ☐ Availability status clear
- ☐ Images have descriptive alt text
- ☐ Schema.org Product markup present
- ☐ Offers markup with price and currency
- ☐ Features listed (ul or ol, not divs)
- ☐ Specifications in definition list or table
- ☐ Links have descriptive text
- ☐ Breadcrumbs present

Blog Post Checklist:

- ☐ Has h1 with article title
- ☐ Heading hierarchy is logical (h1 → h2 → h3)
- ☐ Author name visible and marked up
- ☐ Publication date in HTML (ISO format)
- ☐ Schema.org Article markup present
- ☐ Meta description present
- ☐ Images have alt text
- ☐ Internal links use descriptive text
- ☐ Related articles linked
- ☐ Categories/tags marked up

Service Page Checklist:

- ☐ Clear h1 describing the service
- ☐ Service description in first paragraph
- ☐ Benefits/features listed properly
- ☐ Pricing information (if applicable)
- ☐ Contact information visible
- ☐ Related services linked
- ☐ Case studies or examples linked
- ☐ Schema.org Service markup (if applicable)

Run through these checklists before publishing new pages or after major updates.

The human review

Automated tests catch technical issues, but sometimes you need human judgement:

Have someone unfamiliar with your content:

1. Read just your HTML source (view page source)
2. Attempt to answer:
    - What is this page about?
    - What can I do here?
    - What's the main message?
    - How do I contact the company?

If they struggle, AI will struggle too.

Real-world testing workflow

Here's how I test client sites:

Phase 1: Initial audit (2-3 hours)

1. Run morning-after test on 5-10 key pages
2. Check sitemap exists and is valid
3. Run Lighthouse accessibility audit
4. Validate Schema.org markup
5. Review link text across main navigation
6. Document findings

Phase 2: Detailed analysis (4-6 hours)

1. Headless browser testing on key pages
2. Test with JavaScript disabled
3. Mobile viewport testing
4. Schema.org validation on all markup
5. Content structure review (heading hierarchy)
6. Internal linking audit

Phase 3: Automated setup (2-4 hours)

1. Create automated test suite
2. Set up weekly monitoring
3. Configure alerts for failures
4. Document baseline metrics

Phase 4: Ongoing (30 minutes weekly)

1. Review automated test results
2. Check for new errors
3. Test any new pages added
4. Monitor traffic patterns

Total initial investment: 10-15 hours
Ongoing: 30 minutes per week

The return: knowing your site works for AI, catching problems before they affect visibility.

What you've learned

If you've implemented the patterns from earlier chapters and validated them with these testing methods, you now have:

- Content that AI can discover (sitemaps, navigation)
- Structure that AI can parse (semantic HTML)
- Metadata that AI can extract (Schema.org)
- Confidence that it all works (testing)

But there's one more thing to address: the common mistakes that undermine all this work. The next chapter covers anti-patterns—the things to avoid.

Coming up in Chapter 9: Common anti-patterns—mistakes that make you invisible to AI and how to fix them.

Chapter 9: Common Anti-Patterns

The invisible barriers

You can do everything right—semantic HTML, Schema.org markup, proper navigation—and still make your site invisible to AI with a few specific mistakes. These anti-patterns appear repeatedly in client work, often introduced with good intentions but catastrophic results for AI readability.

Let me show you the most common ones and how to fix them.

Anti-pattern 1: Visual-only information

The problem:

Information conveyed purely through visual styling, without semantic backing.

Real example:

```
<div class="pricing-tiers">
  <div class="tier␣tier-basic">
    <div class="tier-name">Basic</div>
    <div class="tier-price">£29</div>
  </div>

  <div class="tier␣tier-pro␣tier-featured">
    <div class="tier-name">Professional</div>
    <div class="tier-price">£99</div>
    <div class="tier-badge">Most Popular</div>
  </div>

  <div class="tier␣tier-enterprise">
    <div class="tier-name">Enterprise</div>
    <div class="tier-price">£299</div>
  </div>
</div>
```

```
.tier-featured {
  border: 3px solid gold;
  background: #fffef0;
  transform: scale(1.05);
}

.tier-badge {
  background: gold;
  color: black;
  font-weight: bold;
}
```

Humans see: the Professional tier is highlighted with a gold border, larger size, and a “Most Popular” badge.

AI sees: three identical div structures. No indication that Professional is recommended.

The fix:

Make the recommendation explicit in HTML:

```
<section class="pricing-tiers">
  <article class="tier␣tier-basic">
    <h3>Basic</h3>
    <data value="29">£29</data>
    <span class="period">/month</span>
  </article>

  <article class="tier␣tier-pro" aria-label="Recommended␣plan">
    <h3>Professional <span class="badge" aria-label="Most␣popular␣plan">Most Popular</span></h3>
    <data value="99">£99</data>
    <span class="period">/month</span>
    <p><strong>Recommended for most businesses</strong></p>
  </article>

  <article class="tier␣tier-enterprise">
    <h3>Enterprise</h3>
    <data value="299">£299</data>
    <span class="period">/month</span>
  </article>
</section>
```

Now:

- The recommendation is text content, not just styling
- ARIA labels provide additional context
- The `strong` element indicates emphasis
- AI can determine this is the recommended tier

Other examples of visual-only information:

```
<!-- Bad: Urgency shown only through colour -->
<div class="message␣error">Payment failed</div>

<!-- Good: Urgency explicit -->
<div class="message␣error" role="alert" aria-live="assertive">
  <strong>Error:</strong> Payment failed. Please check your card details.
</div>

<!-- Bad: Status shown with icons only -->
<span class="icon-check"></span>
<span class="icon-cross"></span>

<!-- Good: Status in text -->
<span class="status">
  <span aria-hidden="true" class="icon-check"></span>
  Available
</span>
<span class="status">
  <span aria-hidden="true" class="icon-cross"></span>
  Out of stock
</span>
```

Anti-pattern 2: Content in images

The problem:

Text embedded in images without proper alt text or supplementary HTML.

Real example I encountered:

A client had their entire service offering described in an infographic. Beautiful design, completely opaque to AI:

```
<img src="our-services-infographic.png" alt="Our␣services">
```

The image contained:

- Web Development: Full-stack development with modern frameworks
- Mobile Apps: iOS and Android native applications
- Cloud Infrastructure: AWS and Azure deployment and management
- DevOps: CI/CD pipelines and containerization

AI saw: "Our services" - no detail whatsoever.

The fix:

Provide the information in both image and text:

```
<section>
  <h2>Our Services</h2>

  <img src="our-services-infographic.png"
       alt="Infographic␣showing␣four␣service␣categories:␣Web␣Development,␣Mobile␣Apps,␣Cloud␣
            ↪ Infrastructure,␣and␣DevOps">

  <div class="services-text">
    <article>
      <h3>Web Development</h3>
      <p>Full-stack development with modern frameworks including React, Vue, and Node.js</p>
    </article>

    <article>
      <h3>Mobile Apps</h3>
      <p>iOS and Android native applications built with Swift and Kotlin</p>
    </article>

    <article>
      <h3>Cloud Infrastructure</h3>
      <p>AWS and Azure deployment and management with infrastructure as code</p>
    </article>

    <article>
      <h3>DevOps</h3>
      <p>CI/CD pipelines and containerization with Docker and Kubernetes</p>
    </article>
  </div>
</section>
```

You can style `.services-text` to be visually hidden if you want users to see only the infographic, but the content exists for AI and screen readers.

Other common cases:

```
<!-- Bad: Screenshot of pricing table -->
<img src="pricing-screenshot.png" alt="Pricing">
```

```
<!-- Good: Actual HTML table -->
<table>
  <caption>Pricing Plans</caption>
  <thead>
    <tr>
      <th>Plan</th>
      <th>Price</th>
      <th>Features</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>Basic</th>
      <td>£29/month</td>
      <td>Up to 5 users, 10GB storage</td>
    </tr>
    <!-- More rows -->
  </tbody>
</table>

<!-- Bad: Company info in image -->
<img src="contact-card.png" alt="Contact␣us">

<!-- Good: Structured contact info -->
<address>
  <strong>Digital Domain Technologies Ltd</strong><br>
  123 Oxford Road<br>
  Manchester M1 7ED<br>
  <a href="tel:+441611234567">0161 123 4567</a><br>
  <a href="mailto:info@example.com">info@example.com</a>
</address>
```

Anti-pattern 3: Generic link text

The problem:

Links with meaningless text like “click here”, “read more”, “learn more”.

Real example:

```
<section>
  <h3>Edge Delivery Services Migration</h3>
  <p>
    We help companies migrate from traditional CMS platforms to Adobe's
    Edge Delivery Services. Our proven methodology ensures minimal downtime
    and maximum performance gains.
  </p>
  <a href="/services/eds-migration">Learn more</a>
</section>

<section>
  <h3>AEM Consulting</h3>
  <p>
    Strategic advisory for Adobe Experience Manager implementations.
    From architecture to deployment, we guide your team through the process.
  </p>
  <a href="/services/aem-consulting">Learn more</a>
</section>

<section>
  <h3>Training and Support</h3>
  <p>
```

```
    Hands-on training programmes for development teams. Customized to your
    specific needs and technical background.
  </p>
  <a href="/services/training">Learn more</a>
</section>
```

AI sees three “Learn more” links. Which is which? When extracted from context, they’re indistinguishable.

The fix:

Make link text self-descriptive:

```
<section>
  <h3>Edge Delivery Services Migration</h3>
  <p>
    We help companies migrate from traditional CMS platforms to Adobe's
    Edge Delivery Services. Our proven methodology ensures minimal downtime
    and maximum performance gains.
  </p>
  <a href="/services/eds-migration">Explore our EDS migration services</a>
</section>

<section>
  <h3>AEM Consulting</h3>
  <p>
    Strategic advisory for Adobe Experience Manager implementations.
    From architecture to deployment, we guide your team through the process.
  </p>
  <a href="/services/aem-consulting">Learn about AEM consulting and advisory</a>
</section>

<section>
  <h3>Training and Support</h3>
  <p>
    Hands-on training programmes for development teams. Customized to your
    specific needs and technical background.
  </p>
  <a href="/services/training">View training programmes and support options</a>
</section>
```

Each link now describes its destination clearly.

Alternative approach:

If you must use short link text for design reasons, use `aria-label`:

```
<a href="/services/eds-migration"
   aria-label="Learn␣more␣about␣Edge␣Delivery␣Services␣migration">
  Learn more
</a>
```

But descriptive link text is better—it helps everyone, not just AI and screen reader users.

Anti-pattern 4: Broken heading hierarchy

The problem:

Headings used for styling rather than structure, creating illogical hierarchies.

Real example:

```
<h1>Welcome to Digital Domain</h1>

<div class="services-section">
  <h3>Our Services</h3> <!-- Skipped h2 -->

  <div class="service">
    <h2>Web Development</h2> <!-- h2 under h3 -->
    <p>We build modern web applications...</p>
  </div>

  <div class="service">
    <h2>Consulting</h2> <!-- Another h2 under h3 -->
    <p>Strategic guidance for your projects...</p>
  </div>
</div>

<div class="about-section">
  <h4>About Us</h4> <!-- h4 without h2 or h3 -->
  <p>Founded in 1999...</p>
</div>
```

This hierarchy makes no sense:

- h1 (page title)
  - h3 (services) - skipped h2
    - h2 (web dev) - went backwards
    - h2 (consulting) - went backwards
  - h4 (about) - skipped h2 and h3

AI can't build a coherent outline from this.

The fix:

Use heading levels logically:

```
<h1>Welcome to Digital Domain</h1>

<section>
  <h2>Our Services</h2>

  <article>
    <h3>Web Development</h3>
    <p>We build modern web applications...</p>
  </article>

  <article>
    <h3>Consulting</h3>
    <p>Strategic guidance for your projects...</p>
  </article>
</section>

<section>
  <h2>About Us</h2>
  <p>Founded in 1999...</p>
</section>
```

Now the hierarchy is logical:

- h1 (page title)
  - h2 (services)
    - h3 (web dev)
    - h3 (consulting)

    - h2 (about)

If you need different heading sizes for visual design, use CSS:

```
.small-heading {
  font-size: 1.2rem; /* h3 size */
}

.large-heading {
  font-size: 2rem; /* h1 size */
}

<h2 class="small-heading">This looks like h3 but is semantically h2</h2>
```

Test your heading hierarchy:

Use a browser extension like HeadingsMap or run this in the console:

```
Array.from(document.querySelectorAll('h1, h2, h3, h4, h5, h6'))
  .forEach(h => {
    const level = h.tagName[1];
    const indent = ' '.repeat(parseInt(level) - 1);
    console.log(`${indent}${h.tagName}: ${h.textContent.trim()}`);
  });
```

The output should show a logical tree structure.

Anti-pattern 5: JavaScript-only navigation

The problem:

Navigation menus that only exist after JavaScript executes.

Real example:

```
<nav id="main-nav"></nav>

<script>
  const navItems = [
    { text: 'Home', url: '/' },
    { text: 'Services', url: '/services' },
    { text: 'About', url: '/about' }
  ];

  const navHTML = navItems.map(item =>
    `<a href="${item.url}">${item.text}</a>`
  ).join('');

  document.getElementById('main-nav').innerHTML = navHTML;
</script>
```

Parsers see: empty nav element.

The fix:

Put navigation in HTML, enhance with JavaScript if needed:

```
<nav id="main-nav">
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="/services">Services</a></li>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
  </ul>
```

```
</nav>

<script>
  // JavaScript can still enhance this (dropdowns, mobile menu, etc.)
  // But the base navigation works without it
</script>
```

Anti-pattern 6: Hidden content with no fallback

The problem:

Content hidden behind interactions with no alternative access.

Real example:

```
<div class="accordion">
  <button class="accordion-header">What are your opening hours?</button>
  <div class="accordion-content" style="display:␣none;">
    Monday to Friday, 9am to 5pm
  </div>
</div>

<div class="accordion">
  <button class="accordion-header">Do you offer consultations?</button>
  <div class="accordion-content" style="display:␣none;">
    Yes, we offer free 30-minute initial consultations
  </div>
</div>
```

If JavaScript fails or doesn't execute, the answers remain hidden forever.

The fix:

Make content visible by default, hide with JavaScript as enhancement:

```
<section class="accordion-section">
  <article class="accordion-item">
    <h3>
      <button class="accordion-header" aria-expanded="true">
        What are your opening hours?
      </button>
    </h3>
    <div class="accordion-content">
      Monday to Friday, 9am to 5pm
    </div>
  </article>

  <article class="accordion-item">
    <h3>
      <button class="accordion-header" aria-expanded="true">
        Do you offer consultations?
      </button>
    </h3>
    <div class="accordion-content">
      Yes, we offer free 30-minute initial consultations
    </div>
  </article>
</section>

<script>
  // JavaScript collapses items after page load
  document.querySelectorAll('.accordion-item').forEach(item => {
    const button = item.querySelector('.accordion-header');
    const content = item.querySelector('.accordion-content');

    // Collapse by default
    content.style.display = 'none';
    button.setAttribute('aria-expanded', 'false');

    // Add click handler to toggle
    button.addEventListener('click', () => {
      const expanded = button.getAttribute('aria-expanded') === 'true';
      button.setAttribute('aria-expanded', !expanded);
      content.style.display = expanded ? 'none' : 'block';
    });
  });
</script>
```

Now:

- Content is in the HTML (AI can read it)
- JavaScript adds the accordion behaviour
- If JavaScript fails, content remains visible
- This is progressive enhancement

Anti-pattern 7: No sitemap or outdated sitemap

The problem:

Missing sitemap, or sitemap that doesn't reflect actual site structure.

Real examples I've seen:

```
<!-- Sitemap last updated 2019, site extensively changed since -->
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://example.com/old-product-page</loc>
    <lastmod>2019-06-15</lastmod>
  </url>
  <!-- Links to pages that return 404 -->
</urlset>
```

Or sites with no sitemap at all, relying entirely on internal linking for discoverability.

The fix:

Generate sitemaps automatically:

```
// Example: Dynamic sitemap generation
app.get('/sitemap.xml', async (req, res) => {
  const pages = await getAllPages();
  const posts = await getAllBlogPosts();
  const products = await getAllProducts();

  let xml = '<?xml version="1.0" encoding="UTF-8"?>\n';
  xml += '<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">\n';

  // Add all pages
  [...pages, ...posts, ...products].forEach(item => {
    xml += ' <url>\n';
    xml += `  <loc>${item.url}</loc>\n`;
    xml += `  <lastmod>${item.updated.toISOString().split('T')[0]}</lastmod>\n`;
    xml += `  <priority>${item.priority || 0.8}</priority>\n`;
    xml += ' </url>\n';
  });
  xml += '</urlset>';

  res.header('Content-Type', 'application/xml');
  res.send(xml);
});
```

For static sites, regenerate during build:

```
// build-sitemap.js
const fs = require('fs');
const globby = require('globby');

async function generateSitemap() {
  const pages = await globby([
    'dist/**/*.html',
    '!dist/404.html'
  ]);

  const urls = pages.map(page => {
    const path = page.replace('dist', '').replace('/index.html', '/');
    return `https://example.com${path}`;
  });

  const xml = `<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
${urls.map(url => ` <url>
    <loc>${url}</loc>
    <lastmod>${new Date().toISOString().split('T')[0]}</lastmod>
  </url>`).join('\n')}
</urlset>`;

  fs.writeFileSync('dist/sitemap.xml', xml);
}

generateSitemap();
```

Anti-pattern 8: Inconsistent Schema.org

The problem:

Schema.org markup that contradicts visible content or is incomplete.

Real example:

```
<h1>Professional Standing Desk - White Oak Finish</h1>
<p>Price: £599 (currently on sale for £499)</p>

<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "Product",
  "name": "Standing␣Desk",
  "offers": {
    "@type": "Offer",
    "price": "599",
    "priceCurrency": "GBP"
  }
}
</script>
```

Problems:

- Name in Schema doesn’t match h1 (“Standing Desk” vs “Professional Standing Desk - White Oak Finish”)
- Price in Schema is £599 but visible price is £499
- Sale price not reflected in Schema

The fix:

Make Schema.org match visible content exactly:

```
<h1>Professional Standing Desk - White Oak Finish</h1>
<p>
  <span class="original-price">Was: £599</span>
  <strong class="sale-price">Now: £499</strong>
</p>

<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "Product",
  "name": "Professional␣Standing␣Desk␣-␣White␣Oak␣Finish",
  "offers": {
    "@type": "Offer",
    "price": "499",
    "priceCurrency": "GBP",
    "priceValidUntil": "2024-12-31",
    "availability": "https://schema.org/InStock"
  }
}
</script>
```

Better yet, generate Schema.org from the same data source as your visible content to guarantee consistency.

Anti-pattern 9: Forms without proper labels

The problem:

Form inputs identified only by placeholder text.

Real example:

```
<form>
  <input type="text" placeholder="Your␣name">
  <input type="email" placeholder="Email␣address">
  <input type="tel" placeholder="Phone␣number">
  <textarea placeholder="Your␣message"></textarea>
  <button>Send</button>
</form>
```

Problems:

- Placeholders disappear when user types
- No explicit label-input relationship
- AI can’t determine required fields or field purpose with certainty

The fix:

Use proper label elements:

```
<form>
  <div class="form-field">
    <label for="name">Your Name</label>
    <input type="text" id="name" name="name" required>
  </div>

  <div class="form-field">
    <label for="email">Email Address</label>
    <input type="email" id="email" name="email" required>
  </div>

  <div class="form-field">
    <label for="phone">Phone Number <span class="optional">(optional)</span></label>
    <input type="tel" id="phone" name="phone">
  </div>

  <div class="form-field">
    <label for="message">Your Message</label>
    <textarea id="message" name="message" rows="5" required></textarea>
  </div>

  <button type="submit">Send Message</button>
</form>
```

Now:

- Each input has an explicit label
- Required fields marked with `required` attribute
- Optional fields indicated in label
- AI can determine form purpose and structure

Anti-pattern 10: Table abuse

The problem:

Tables used for layout or data tables without proper structure.

Layout table example (don't do this):

```
<table>
  <tr>
    <td>
      <nav>
        <!-- Navigation -->
      </nav>
    </td>
    <td>
      <main>
        <!-- Main content -->
      </main>
    </td>
    <td>
      <aside>
        <!-- Sidebar -->
      </aside>
    </td>
  </tr>
</table>
```

This hasn't been acceptable since CSS gained layout capabilities. Use flexbox or grid instead.

Data table without structure:

```
<table>
  <tr>
```

```
    <td>Plan</td>
    <td>Price</td>
    <td>Users</td>
  </tr>
  <tr>
    <td>Basic</td>
    <td>£29</td>
    <td>5</td>
  </tr>
  <tr>
    <td>Pro</td>
    <td>£99</td>
    <td>25</td>
  </tr>
</table>
```

AI can't distinguish headers from data.

The fix:

Use semantic table elements:

```
<table>
  <caption>Pricing Plan Comparison</caption>
  <thead>
    <tr>
      <th scope="col">Plan Name</th>
      <th scope="col">Monthly Price</th>
      <th scope="col">Maximum Users</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th scope="row">Basic</th>
      <td>£29</td>
      <td>5</td>
    </tr>
    <tr>
      <th scope="row">Professional</th>
      <td>£99</td>
      <td>25</td>
    </tr>
    <tr>
      <th scope="row">Enterprise</th>
      <td>£299</td>
      <td>Unlimited</td>
    </tr>
  </tbody>
</table>
```

Now AI can answer: "What's the price of the Professional plan?" by finding the row where the row header is "Professional" and extracting the price cell.

Anti-pattern 11: Content in iframes

The problem:

Important content loaded in iframes from external sources.

Real example:

```
<h2>Our Latest News</h2>
<iframe src="https://news-widget.example.com/feed"></iframe>
```

AI sees: an iframe. The content inside is from a different domain, potentially inaccessible.

The fix:

Pull content server-side and render it in your HTML:

```
<section>
  <h2>Our Latest News</h2>

  <article>
    <h3><a href="/news/new-service-launch">New Service Launch</a></h3>
    <time datetime="2024-03-20">20 March 2024</time>
    <p>We're introducing Edge Delivery Services consulting...</p>
  </article>

  <article>
    <h3><a href="/news/team-expansion">Team Expansion</a></h3>
    <time datetime="2024-03-15">15 March 2024</time>
    <p>Digital Domain welcomes three new consultants...</p>
  </article>
</section>
```

If you must use an iframe (embedded maps, video players), ensure critical information is also in HTML:

```
<section>
  <h2>Our Location</h2>

  <address>
    Digital Domain Technologies Ltd<br>
    123 Oxford Road<br>
    Manchester M1 7ED<br>
    United Kingdom
  </address>

  <p>
    <a href="https://maps.google.com/?q=123+Oxford+Road+Manchester">View on Google Maps</a>
  </p>

  <!-- Map iframe as enhancement -->
  <iframe src="https://www.google.com/maps/embed?pb=..."
          width="600" height="450"
          title="Map␣showing␣our␣office␣location␣at␣123␣Oxford␣Road,␣Manchester">
  </iframe>
</section>
```

Anti-pattern 12: PDF-only content

The problem:

Important information only available as PDF downloads.

Real example:

```
<h2>Our Services</h2>
<p>Download our services brochure to learn more.</p>
<a href="/brochure.pdf">Download PDF (2.3 MB)</a>
```

AI might not parse PDFs, or might parse them poorly if they're scanned images.

The fix:

Provide information in HTML, offer PDF as alternative:

```
<section>
  <h2>Our Services</h2>

  <article>
    <h3>Edge Delivery Services</h3>
    <p>Modern web delivery using Adobe's Edge Delivery Services platform...</p>
    <ul>
      <li>Migration from legacy CMS platforms</li>
      <li>Custom block development</li>
      <li>Performance optimization</li>
    </ul>
  </article>

  <article>
    <h3>AEM Consulting</h3>
    <p>Strategic advisory for Adobe Experience Manager implementations...</p>
    <ul>
      <li>Architecture planning</li>
      <li>Implementation guidance</li>
      <li>Team training</li>
    </ul>
  </article>

  <aside>
    <p>
      <a href="/services-brochure.pdf">Download this information as PDF</a>
      (2.3 MB)
    </p>
  </aside>
</section>
```

If you must publish content primarily as PDF, ensure the PDF has:

- Text layer (not scanned images)
- Proper document structure
- Meaningful alt text for images
- Metadata (title, author, subject)

Anti-pattern 13: Auto-playing content

The problem:

Content that changes without user interaction, making specific information hard to reference.

Real example:

```
<div class="testimonials-carousel" data-autoplay="3000">
  <div class="testimonial">
    <p>"Great␣service!" - Client A</p>
  </div>
  <div class="testimonial">
    <p>"Highly␣recommend" - Client B</p>
  </div>
  <div class="testimonial">
    <p>"Outstanding␣work" - Client C</p>
  </div>
</div>
```

The carousel auto-rotates every 3 seconds. AI fetching the page might only see one testimonial, missing the others.

The fix:

Show all content in HTML, add carousel as enhancement:

```
<section class="testimonials">
  <h2>Client Testimonials</h2>

  <article class="testimonial">
    <blockquote>
      <p>Digital Domain transformed our web presence. The migration to Edge Delivery Services
          ↪ was seamless, and performance improved dramatically.</p>
    </blockquote>
    <figcaption>
      <cite>Sarah Johnson, CTO at AutoCorp</cite>
    </figcaption>
  </article>

  <article class="testimonial">
    <blockquote>
      <p>Tom's expertise in AEM saved us months of development time. His guidance on
          ↪ architecture was invaluable.</p>
    </blockquote>
    <figcaption>
      <cite>David Chen, Head of Digital at FinanceGroup</cite>
    </figcaption>
  </article>

  <article class="testimonial">
    <blockquote>
      <p>The training programme upskilled our entire team. We're now confident managing our EDS
          ↪ implementation independently.</p>
    </blockquote>
    <figcaption>
      <cite>Emma Williams, Development Manager at RetailHub</cite>
    </figcaption>
  </article>
</section>

<script>
  // JavaScript can add carousel functionality without hiding content
  // Use CSS to show one at a time visually while keeping all in DOM
</script>
```

How to audit for anti-patterns

Quick audit checklist:

1. Disable CSS - does critical info disappear?
2. Disable JavaScript - is the site still usable?
3. View source - is content in HTML or generated?
4. Extract all links - are they descriptive?
5. Check heading hierarchy - does it make sense?
6. Look for images - do they have proper alt text?
7. Find forms - do inputs have labels?
8. Check tables - are they structured properly?
9. Review sitemap - is it current and complete?
10. Validate Schema.org - does it match visible content?

Red flags:

- Empty `<div id="app">` or `<div id="root">`
- Lots of `display: none` without semantic reason

- Navigation in JavaScript only
- “Learn more” and “Click here” everywhere
- Images with `alt=""` or `alt="image"`
- Placeholder text instead of labels
- Heading levels jumping around (h1 → h4 → h2)
- Sitemap.xml returning 404
- Forms submitting but no confirmation

The quick wins

If you can only fix a few things, prioritize these:

1. Add proper heading hierarchy (30 minutes)
Big impact on AI’s ability to understand page structure.

2. Fix link text (1-2 hours)
Replace “click here” with descriptive links.

3. Add alt text to images (2-3 hours)
Describe what each image shows.

4. Create or update sitemap (1 hour)
Ensure AI can discover all your pages.

5. Add basic Schema.org to homepage (1 hour)
Organization/LocalBusiness markup gives AI your basic details.

These five fixes, taking 6-8 hours total, will solve 80% of common AI readability problems.

What’s next

You’ve learned what to avoid. Now let’s bring everything together—the patterns to embrace, the anti-patterns to avoid, and the testing to validate.

The final chapter provides a practical implementation roadmap: how to actually do this work on a real site, with real constraints, in a realistic timeframe.

Coming up in Chapter 10: Implementation roadmap—from audit to deployment, making your site AI-readable without rebuilding everything.

Chapter 10: Implementation Roadmap

From theory to practice

You've learned the principles, seen the patterns, and understand the pitfalls. Now comes the practical question: how do you actually implement this on a real site with real constraints?

This chapter provides a realistic roadmap. Not a fantasy "rebuild everything perfectly" plan, but a pragmatic approach that acknowledges you have:

- Limited budget
- Limited time
- Other priorities competing for resources
- Stakeholders who need convincing
- Legacy code you can't just delete
- A business to run while making improvements

Let's build a plan that works in the real world.

The implementation philosophy

Start small, measure impact, expand gradually.

Don't try to fix everything at once. Pick high-value pages, implement improvements, measure the results, then use those results to justify expanding the work.

This approach:

- Shows ROI quickly
- Builds stakeholder confidence
- Lets you learn what works for your specific site
- Keeps the project manageable

Phase 0: Assessment

Before you change anything, understand what you're working with.

Tasks:

1. Inventory high-value pages

Priority: Foundation (complete before other work) (Task 1)

List your 15-25 most important pages:

- Homepage
- Top 5 landing pages (by traffic)
- Top 5 product/service pages (by conversion)
- Top 5 blog posts (by traffic)
- Contact page
- About page

Don’t try to fix 500 pages. Fix the 20 that matter most.

2. Quick readability audit

Priority: Foundation (Task 2)

For each page, check:

- Is content in the HTML source? (View source, not inspect element)
- Can you see an h1?
- Is there a logical heading hierarchy?
- Are there Schema.org scripts?
- Do links have descriptive text?
- Do images have alt text?

Create a simple spreadsheet:

```
Page | Content in HTML | Has h1 | Has Schema.org | Good links | Has alt text
-----|-----------------|--------|----------------|------------ |-------------
Homepage | Yes | Yes | No | No | Partial
/services | No (JS) | Yes | No | No | Yes
/products/desk | Yes | Yes | No | No | No
```

3. Run the morning-after test

Priority: Foundation (Task 3)

Pick 5 representative pages. For each:

1. View source
2. Copy HTML
3. Paste into ChatGPT/Claude
4. Ask: “What is this page about? What can I do here?”

Document what works and what doesn’t.

4. Check technical foundations

Priority: Foundation (Task 4)

- Does sitemap.xml exist?
- Is robots.txt blocking important content?
- Does the site work with JavaScript disabled?
- Are there accessibility issues? (Run Lighthouse)

5. Prioritize by impact and effort

Priority: Foundation (Task 5)

Score each page:

- Impact: High (5), Medium (3), Low (1)

- Effort: Easy (1), Medium (3), Hard (5)
- Priority score = Impact / Effort

Focus on high impact, low effort items first.

Deliverable: A prioritized list of pages with specific issues documented.

Phase 1: Quick Wins

Fix high-impact, low-effort problems. These show immediate results and build momentum.

Tasks:

1. Add Schema.org to key pages

Priority 1: Critical Quick Win (Task 1)

Start with:

- Homepage: Organization/LocalBusiness schema
- Top 5 products/services: Product/Service schema
- Top 3 blog posts: Article schema
- FAQ page: FAQPage schema (if you have one)

Use JSON-LD. Copy the patterns from Chapter 5. Each page takes 30-60 minutes once you have a template.

Example workflow per page:

1. Identify the page type (product, article, etc.)
2. Copy the appropriate Schema.org template
3. Fill in with actual data from the page
4. Add `<script type="application/ld+json">` to page
5. Validate with Google's Rich Results Test
6. Deploy

2. Fix heading hierarchy

Priority 1: Critical Quick Win (Task 2)

For each priority page:

1. Extract all headings: `document.querySelectorAll('h1, h2, h3, h4')`
2. Check the hierarchy makes sense
3. Fix any jumps (h1 → h4) or backwards progressions (h3 → h2)
4. Ensure each page has exactly one h1

This is often just changing div classes to proper heading elements and adjusting CSS.

3. Improve link text

Priority 1: Critical Quick Win (Task 3)

Find all generic links:

```
document.querySelectorAll('a').forEach(a => {
  const text = a.textContent.toLowerCase().trim();
  if (text === 'click here' || text === 'read more' || text === 'learn more') {
    console.log(text, →'', a.href);
  }
});
```

Replace with descriptive text:

- “Learn more” → “Learn about our EDS consulting services”
- “Click here” → “View our product catalogue”
- “Read more” → “Read the full migration guide”

4. Add alt text to images

Priority 1: Critical Quick Win (Task 4)

Audit images:

```
document.querySelectorAll('img').forEach(img => {
  if (!img.alt || img.alt === 'image' || img.alt === '') {
    console.log('Missing alt:', img.src);
  }
});
```

Add descriptive alt text:

- Not: `alt="product"`
- But: `alt="Standing desk showing electric height adjustment control panel"`

Impact: These changes make your content immediately more readable to AI without changing how pages look to humans.

Expected results:

- Schema.org validation passes on key pages
- Morning-after test shows AI can extract key information
- Accessibility scores improve (Lighthouse)

Phase 2: Structural Improvements

Address deeper structural issues.

Tasks:

1. Create or update sitemap.xml

Priority 2: Essential Improvement (Task 1)

If you don’t have a sitemap, create one. If you do, ensure it’s current.

For static sites:

```
// build-sitemap.js
const fs = require('fs');
const pages = [
  { url: '/', priority: 1.0 },
  { url: '/services', priority: 0.9 },
  { url: '/services/eds', priority: 0.8 },
  // ... all your pages
];

const xml = `<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
${pages.map(p => ` <url>
    <loc>https://example.com${p.url}</loc>
    <lastmod>${new Date().toISOString().split('T')[0]}</lastmod>
    <priority>${p.priority}</priority>
  </url>`).join('\n')}
</urlset>`;

fs.writeFileSync('sitemap.xml', xml);
```

For dynamic sites, generate from your database/CMS.

2. Add semantic navigation elements

Priority 2: Essential Improvement (Task 2)

Replace divs with proper semantic elements:

```
<!-- Before -->
<div class="nav">...</div>
<div class="main-content">...</div>
<div class="sidebar">...</div>

<!-- After -->
<nav aria-label="Main␣navigation">...</nav>
<main>...</main>
<aside>...</aside>
```

3. Implement breadcrumbs

Priority 2: Essential Improvement (Task 3)

Add breadcrumbs to key sections:

```
<nav aria-label="Breadcrumb">
  <ol itemscope itemtype="https://schema.org/BreadcrumbList">
    <li itemprop="itemListElement" itemscope itemtype="https://schema.org/ListItem">
      <a itemprop="item" href="/"><span itemprop="name">Home</span></a>
      <meta itemprop="position" content="1">
    </li>
    <li itemprop="itemListElement" itemscope itemtype="https://schema.org/ListItem">
      <a itemprop="item" href="/services"><span itemprop="name">Services</span></a>
      <meta itemprop="position" content="2">
    </li>
  </ol>
</nav>
```

4. Server-side rendering for critical paths

Priority 2: Essential Improvement (Task 4)

If your site is JavaScript-heavy, implement SSR for public pages.

For Next.js:

```
// Convert from client-side
export default function ProductPage() {
  const [product, setProduct] = useState(null);

  useEffect(() => {
    fetch('/api/product/123').then(r => r.json()).then(setProduct);
  }, []);

  if (!product) return <div>Loading...</div>;
  return <div>{product.name}</div>;
}

// To server-side
export async function getServerSideProps() {
  const product = await fetchProduct('123');
  return { props: { product } };
}

export default function ProductPage({ product }) {
  return <div>{product.name}</div>;
}
```

Focus on:

- Homepage
- Product/service pages
- Blog posts
- Landing pages

Authenticated areas can stay client-side.

5. Progressive enhancement for interactive features

Priority 2: Essential Improvement (Task 5)

Make interactive features work without JavaScript:

```
<!-- Accordion: content visible by default -->
<details>
  <summary>What are your opening hours?</summary>
  <p>Monday to Friday, 9am to 5pm</p>
</details>

<!-- Or with div, show by default, hide with JavaScript -->
<div class="accordion-item">
  <h3><button>Question</button></h3>
  <div class="answer">Answer text</div>
</div>

<script>
  // JavaScript hides and adds toggle functionality
  // But content is in HTML from the start
</script>
```

Expected results:

- All priority pages discoverable via sitemap
- Content accessible without JavaScript
- Clear site structure visible to AI
- Breadcrumbs showing page relationships

Phase 3: Content Patterns

Standardize content patterns across your site.

Tasks:

1. Implement proper article structure

Priority 3: Content Optimization (Task 1)

Create templates for blog posts:

```
<article itemscope itemtype="https://schema.org/Article">
  <header>
    <h1 itemprop="headline">{{ title }}</h1>
    <p>
      By <span itemprop="author">{{ author }}</span> |
      <time itemprop="datePublished" datetime="{{␣date␣}}">{{ formattedDate }}</time>
    </p>
  </header>

  <div itemprop="articleBody">
    {{ content }}
  </div>
</article>
```

2. Convert visual lists to semantic lists

Priority 3: Content Optimization (Task 2)

Find divs that are really lists:

```
<!-- Before -->
<div class="features">
  <div class="feature">Fast performance</div>
  <div class="feature">Secure by default</div>
  <div class="feature">Easy integration</div>
</div>

<!-- After -->
<ul class="features">
  <li>Fast performance</li>
  <li>Secure by default</li>
  <li>Easy integration</li>
</ul>
```

3. Add definition lists for key-value pairs

Priority 3: Content Optimization (Task 3)

Replace custom markup with `<dl>`:

```
<!-- Before -->
<div class="specs">
  <div class="spec-row">
    <span class="label">Dimensions</span>
    <span class="value">120cm × 60cm</span>
  </div>
</div>

<!-- After -->
<dl class="specs">
  <dt>Dimensions</dt>
  <dd>120cm × 60cm</dd>
</dl>
```

4. Fix form labels

Priority 3: Content Optimization (Task 4)

Ensure every input has a label:

```
<!-- Before -->
<input type="text" placeholder="Your␣name">

<!-- After -->
<label for="name">Your Name</label>
<input type="text" id="name" name="name">
```

5. Add ARIA labels where needed

Priority 3: Content Optimization (Task 5)

For elements that need clarification:

```
<nav aria-label="Main␣navigation">...</nav>
<nav aria-label="Footer␣links">...</nav>

<button aria-label="Close␣dialog">×</button>

<a href="/products" aria-label="View␣all␣products␣in␣our␣catalogue">
  Products
</a>
```

6. Improve table structure

Priority 3: Content Optimization (Task 6)

Add proper table semantics:

```
<table>
  <caption>Pricing Comparison</caption>
  <thead>
    <tr>
      <th scope="col">Plan</th>
      <th scope="col">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th scope="row">Basic</th>
      <td>£29/month</td>
    </tr>
  </tbody>
</table>
```

Expected results:

- Consistent semantic patterns across pages
- Forms fully accessible and AI-readable
- Data relationships explicit (lists, tables)

Phase 4: Testing and Validation

Verify everything works.

Tasks:

1. Set up automated testing

Priority 4: Validation Priority 4: Validation & Monitoring Monitoring (Task 1)

Create test suite (from Chapter 8):

```
const tests = [
  {
    url: 'https://example.com/',
    checks: { hasH1: true, hasSchema: true, hasNav: true }
  },
  // ... all priority pages
];

// Run tests, generate report
```

2. Manual validation of priority pages

Priority 4: Validation Priority 4: Validation & Monitoring Monitoring (Task 2)

For each priority page:

1. Run morning-after test
2. Validate Schema.org (Google Rich Results Test)
3. Check Lighthouse accessibility score
4. Verify in multiple browsers
5. Test with JavaScript disabled

Document results in spreadsheet:

```
Page | Morning-after | Schema valid | Lighthouse | JS disabled
-----|---------------|--------------|------------|------------
Homepage | Pass | Pass | 95 | Pass
/services | Pass | Pass | 92 | Pass
/products/desk | Pass | Pass | 98 | Pass
```

3. Create documentation and standards

Priority 4: Validation Priority 4: Validation & Monitoring Monitoring (Task 3)

Document your patterns:

content-standards.md:

```
# Content Standards

## Page Types

### Product Pages
- Use h1 for product name
- Include Schema.org Product markup
- List features in <ul>
- Show specs in <dl>
- Images must have descriptive alt text

### Blog Posts
- Use h1 for article title
- Include Schema.org Article markup
- Author and date required
- Logical heading hierarchy (h1 →h2 →h3)
```

Share with your team so new content follows the patterns.

Expected results:

- Automated tests catching regressions
- All priority pages validated
- Team has documentation to maintain standards

Phase 5: Ongoing Maintenance

Keep your improvements intact.

Tasks:

Weekly (30 minutes):

1. Review automated test results
2. Check for Schema.org errors in Search Console
3. Spot-check new pages added this week

Monthly:

1. Audit 5-10 recent pages for compliance
2. Run morning-after test on new content
3. Check sitemap is current
4. Review accessibility scores

Quarterly:

1. Full audit of all priority pages
2. Update documentation if patterns have evolved
3. Review and update Schema.org markup
4. Check for new anti-patterns creeping in

Continuous:

- New pages follow documented patterns
- Code reviews check for semantic HTML
- CMS templates generate proper markup
- Build process includes validation

The real-world case study

A client in the automotive sector came to me with this situation:

Their site:

- React SPA (single page application)
- Beautiful design, won awards
- £2 million spent on recent redesign
- Every page returned `<div id="root"></div>` in HTML
- JavaScript rendered everything client-side

The problem: When someone asked an AI “Where can I get my Land Rover serviced near Birmingham?” the AI couldn’t answer. The information existed but was invisible to parsers.

The implementation:

We followed this roadmap progressively:

Phase 0: Assessment

- Audited 25 high-value pages
- Identified top conversion paths
- Documented current AI visibility (basically zero)
- Got stakeholder buy-in with morning-after test demo

Phase 1: Quick Wins

- Added Schema.org to homepage (LocalBusiness)
- Added Product schema to top 10 vehicle service pages
- Added FAQPage schema to FAQ section
- Fixed heading hierarchy (was all over the place)
- Added alt text to all images

Phase 2: Structural Improvements

- Implemented Next.js SSR for public pages
- Created dynamic sitemap generation
- Added breadcrumb navigation
- Kept client-side rendering for dealer portal (authenticated area)

Phase 3: Content Patterns

- Created templates for service pages
- Standardized location pages
- Implemented proper navigation structure
- Added semantic HTML throughout

Phase 4: Testing and Validation

- Automated test suite
- Manual validation
- Gradual rollout (10% → 50% → 100%)
- Documentation for content team

The investment:

- Consulting time (me)
- Developer time (their team)
- Total cost: ~£12,000

The results (after 3 months):

- AI recommendations increased 60%
- Organic search traffic up 35%
- Voice search queries doubled
- Rich results appearing for 15+ pages
- Support queries decreased (AI could answer common questions)

The payoff: The £2M redesign was invisible to AI. The £12K fix made it visible.

ROI was obvious within 3 months.

Scaling beyond priority pages

Once your priority pages work well, expand gradually:

Months 2-3:

- Apply patterns to next 25 pages

- Create templates for common page types
- Train content team on standards

Months 4-6:

- Expand to all public pages
- Automate Schema.org generation
- Build validation into CMS

Months 7-12:

- Optimize long-tail content
- Refine based on data
- Update documentation
- Continuous improvement

Don't try to fix 1,000 pages at once. Fix 20, measure results, then expand based on what worked.

Handling common constraints

Constraint 1: "We don't have budget for a full implementation"**

Do Phase 0 and Phase 1 only.

This gives you:

- Schema.org on key pages
- Fixed heading hierarchy
- Better link text
- Image alt text

That's 70% of the benefit for 30% of the effort.

Constraint 2: "We can't change our JavaScript framework"**

Focus on what you can control:

- Add Schema.org (doesn't require framework changes)
- Improve link text
- Fix heading hierarchy
- Add alt text
- Create proper sitemap
- Use SSR if your framework supports it (most do now)

You don't need to rebuild. You can improve what exists.

Constraint 3: "Our CMS makes it hard to add Schema.org"**

Options:

1. Add Schema.org via Google Tag Manager
2. Create a plugin/extension for your CMS
3. Generate Schema.org server-side and inject into pages
4. Use a Schema.org generation service

There's always a way. It might not be elegant, but it works.

Constraint 4: "We have thousands of pages"**

Fix the top 20 pages first. Then:

1. Create templates for page types
2. Apply templates to batches of 50-100 pages
3. Fix opportunistically (when pages are updated anyway)
4. Prioritize by traffic × conversion value

You'll never fix everything, and you don't need to. The 80/20 rule applies: 20% of pages generate 80% of value.

Constraint 5: "Our development team is fully committed"**

Options:

1. Hire a contractor for the initial implementation
2. Make it a gradual background task
3. Build it into regular feature work
4. Allocate time during quiet periods

Or demonstrate ROI with Phase 1 quick wins, then make the business case for dedicated time.

Measuring success

Track these metrics:

Technical metrics:

- % of priority pages with Schema.org
- % with proper heading hierarchy
- % with descriptive links
- % with alt text on images
- Lighthouse accessibility scores

AI readability metrics:

- Morning-after test pass rate
- Schema.org validation pass rate
- Pages in sitemap
- JavaScript-disabled functionality

Business metrics:

- Organic search traffic
- AI referral traffic (if trackable)
- Voice search queries
- Rich results impressions (Search Console)
- Conversion rate from organic

The reality check:

You won't see dramatic overnight changes. This is a 3-6 month investment that compounds over time.

Expect:

- Month 1: Technical improvements visible
- Month 2: Search engines re-index updated pages

- Month 3: Traffic patterns start changing
- Month 6: Clear ROI visible

Building stakeholder support

Stakeholders care about ROI, not semantic HTML. Frame this work appropriately:

For executives: "AI agents are increasingly making purchase recommendations. Our site is currently invisible to them. This costs us enquiries. Here's a 10-week plan to fix it with measurable ROI."

For marketing: "This improves our organic search performance, gets us rich results in Google, and makes our content AI-discoverable. It complements our SEO strategy."

For product: "This is progressive enhancement. Better for accessibility, better for performance, better for SEO, future-proof for AI."

For development: "This is mostly using HTML the way it was designed to work. Less hacky divs, more semantic elements. It's cleaner code that happens to also work better for AI."

The demo that convinces:

1. Take your homepage HTML (view source)
2. Paste into ChatGPT
3. Ask: "What does this company do?"
4. Watch as AI can't answer

Then show a competitor's site that works better.

That usually gets attention.

The anti-roadmap

What not to do:

Don't:

- Try to fix everything at once
- Rebuild from scratch
- Obsess over perfection
- Add Schema.org to every possible element
- Implement patterns you don't understand
- Ignore your existing design system
- Make changes without testing
- Forget to measure results

Do:

- Start small and focused
- Work with what you have
- Aim for "good enough"
- Focus on high-value Schema.org types
- Understand why patterns work
- Adapt patterns to your system
- Test everything
- Track and communicate results

The 10-week summary

Here's the complete timeline at a glance:

Phase 0: Assessment

- Deliverable: Prioritized list of 15-25 pages with issues

Phase 1: Quick Wins

- Deliverable: Schema.org, headings, links, alt text on priority pages

Phase 2: Structural Improvements

- Deliverable: Sitemap, semantic HTML, SSR, progressive enhancement

Phase 3: Content Patterns

- Deliverable: Standardized templates, consistent semantic markup

Phase 4: Testing and Validation

- Deliverable: Automated tests, validation, documentation

Phase 5: Ongoing Maintenance

- Deliverable: Sustained quality, continuous improvement

Total initial investment: Priority 1-4 phases

Ongoing investment: Priority: Ongoing phase

Expected ROI: Compounds over time as implementation progresses

Final thoughts

This roadmap is based on real client work. The results are achievable.

You don't need to be perfect. You need to be better than you are now, and better than your competitors.

Start with Phase 0. Do the assessment. See what you find. Then decide how much of this roadmap makes sense for your situation.

The invisible users are visiting your site right now. They're making decisions about whether to recommend you. The question is: can they understand what they're reading?

With this roadmap, you can make sure the answer is yes.

What you've accomplished

If you've read this entire book and followed along, you now understand:

- Who AI agents are and why they matter (Chapter 1)
- How AI reads differently from humans (Chapter 2)
- The principles that guide AI-friendly design (Chapter 3)
- Content architecture patterns that work (Chapter 4)
- Metadata implementation with Schema.org (Chapter 5)
- Navigation and site discovery (Chapter 6)
- The JavaScript challenge and solutions (Chapter 7)
- Testing methodologies (Chapter 8)

- Anti-patterns to avoid (Chapter 9)
- How to actually implement all of this (Chapter 10)

More importantly, you have a practical path forward. Not a theoretical framework, but a realistic plan you can execute.

The web is changing. AI agents are becoming primary consumers of web content. Sites that work for AI will thrive. Sites that don't will become invisible.

You now have everything you need to make sure your site is visible.

Good luck.

End of MX-Handbook

Chapter 11: Business Imperative

Why fixing this matters commercially.

The Platform Race

January 2026 changed everything. Four major platforms launched agent systems within eight days:

- **Amazon Alexa+** (January 5th) - Voice-initiated purchasing with partner retailers
- **Microsoft Copilot Checkout** (January 8th) - Complete transactions within the AI assistant
- **Google Universal Commerce Protocol** (January 11th) - Open standard with Target and Walmart
- **Anthropic Claude Cowork** (January 12th) - Autonomous digital colleague for complex workflows

This isn't speculation about future possibilities. Agent-mediated commerce and autonomous task execution are happening now, and businesses that aren't ready are being excluded from high-conversion transactions and modern workflows.

The Invisible Failures

The patterns that break AI agents cost real money. Here's what these failures look like in business terms:

Business Type	Failure Pattern	What Happened	Lost Revenue	Analytics Visibility
Tour Operator	Paginated itinerary (14 pages)	Agent saw only Day 1, recommended competitor with single-page layout	£2,000 per lost booking	No trace - appears as normal bounce
E-commerce Site	Toast notification on add-to-cart	Agent missed confirmation, reported “out of stock”	£150 per abandoned sale	Shows as abandoned session

Business Type	Failure Pattern	What Happened	Lost Revenue	Analytics Visibility
SaaS Platform	Hidden pricing (“Contact sales”)	Agent couldn’t provide pricing, recommended competitor with transparent pricing	£50,000 annual contract	Short session, no engagement
Restaurant Booking	SPA with no URL state changes	Agent couldn’t confirm booking succeeded	Lost reservation	Incomplete form submission

Key insight: These failures are invisible in traditional analytics. They appear as bounces, short sessions, or abandoned forms - nothing that indicates AI agent failures. Meanwhile, competitors with agent-friendly patterns capture the business without your knowledge.

Real-World Impact: Stack Overflow

Stack Overflow reduced its workforce by 28% in 2024, coinciding with ChatGPT’s rise. Developers who previously visited Stack Overflow for answers now ask AI assistants directly.

This is the business model crisis in action: traffic-dependent revenue models collapse when AI agents answer questions directly instead of sending users to source websites.

The Revenue Model Collision

Consider how most free websites make money:

Human visit:

- 8 minutes on site
- 12 page views
- £0.50 in advertising revenue

AI agent visit:

- 0.3 seconds on site
- 1 page view
- £0.04 in revenue (if ads render at all)

The mathematics are brutal:

- 87% reduction in page views
- 92% reduction in time on site
- 92% reduction in advertising revenue per visit

This isn’t a minor optimisation problem. For advertising-funded content, it’s existential.

E-commerce: Where Incentives Match

Not every business model suffers from agent traffic. Transaction-based businesses often benefit.

Traditional human shopping:

- Visits site 8 times before purchasing
- Abandons cart 69% of the time
- Average conversion rate: 2-3%
- Customer acquisition cost: £45

Agent shopping:

- Visits once with clear purchase intent
- Completes transaction 80%+ of the time
- Minimal acquisition cost
- Makes decisions based on objective criteria

For retailers, agents are dream customers. They don't abandon carts. They don't require retargeting campaigns. They convert at extraordinary rates.

Microsoft Copilot Checkout validates this: retailers including Urban Outfitters, Anthropologie, and Etsy are seeing improved conversions from agent-mediated transactions.

The First-Mover Advantage

When users ask AI assistants for recommendations, agents cite businesses they've successfully transacted with previously. The first business in each sector to implement agent-friendly patterns establishes preference in agent recommendation systems.

This creates network effects:

Hotel A (agent-friendly):

- Gets 90% of agent traffic
- Learns from agent interactions
- Improves patterns further
- Becomes the default recommendation

Hotel B (agent-hostile):

- Gets 10% of agent traffic
- Falls behind in citations
- Loses market position

First-mover citation advantage creates durable competitive moats.

Strategic Assessment: Where Are You?

Four critical questions determine your exposure:

1. Revenue model exposure

High risk:

- Advertising-funded content
- Affiliate marketing requiring clicks
- Traffic-dependent discovery

- Engagement-maximising designs

Low risk:

- Transaction-based revenue
- Direct sales
- Subscription services
- API access models

2. Customer acquisition dependency

High risk:

- Rely on organic search traffic
- Content marketing for discovery
- SEO-driven funnel
- Brand awareness through engagement

Low risk:

- Direct relationships with customers
- Known brand with purchase intent
- B2B with long sales cycles
- Offline acquisition channels

3. Information complexity

High risk:

- Multi-page information flows
- JavaScript-heavy interfaces
- Dynamic content without structure
- Visual-only layouts

Low risk:

- Single-page information presentation
- Semantic HTML structure
- Progressive enhancement
- Explicit state management

4. Competitive positioning

High risk:

- Competitors already optimised
- Commoditised products/services
- Price-sensitive market
- Agent recommendations matter

Low risk:

- Unique value proposition
- Relationship-based sales
- Complex purchasing decisions
- Human expertise required

Implementation ROI

The investment required varies by priority level:

Priority 1 (Foundation):

- Semantic HTML fixes
- Basic Schema.org markup
- Testing methodology
- Estimated effort: 2-3 developer weeks spread over 1-2 months

Priority 2 (Essential):

- Navigation improvements
- State management fixes
- Comprehensive metadata
- Estimated effort: 4-6 developer weeks spread over 2-3 months

Priority 3 (Advanced):

- Protocol integration (ACP/UCP)
- API development
- Agent-specific features
- Estimated effort: 8-12 developer weeks spread over 3-6 months

Example calculation (e-commerce site):

Investment:

- Priority 1+2 implementation: £15,000
- Testing and validation: £3,000
- Ongoing maintenance: £2,000/year

Returns (conservative estimate):

- Current monthly revenue: £100,000
- Agent traffic reaches 15% over 12 months
- Agent conversion rate: 75% (vs 3% human)
- Additional monthly revenue: £10,000
- Annual return: £120,000

ROI: 780% in year one

This assumes you're currently invisible to agents and competitors aren't optimised yet. Returns decrease as market matures.

Protocol Integration Decision

Three major approaches exist:

Agentic Commerce Protocol (ACP)

Status: Open source (Apache 2.0), OpenAI and Stripe partnership

Advantages:

- Open standard, portable across agents
- Works with existing payment providers
- No platform lock-in

Disadvantages:

- Newer, less market penetration
- Requires implementation effort
- No guaranteed distribution

Universal Commerce Protocol (UCP)

Status: Open standard, Google partnership with Target and Walmart

Advantages:

- Google Search distribution
- Major retailer endorsement
- Open specification

Disadvantages:

- Google dependency for visibility
- Competition from Google Shopping
- Standard still evolving

Microsoft Copilot Checkout

Status: Proprietary, Microsoft platform

Advantages:

- Immediate market access
- Proven conversion rates
- Platform handles complexity

Disadvantages:

- Proprietary lock-in
- Microsoft controls distribution
- Revenue sharing required

Recommendation for most businesses:

If exposure is high: Implement both ACP and UCP to maintain optionality

If exposure is medium: Choose one protocol based on customer base (Windows users → Copilot; Google Search users → UCP)

If exposure is low: Wait for market consolidation, focus on foundation patterns (semantic HTML, Schema.org)

The Discovery Problem

Protocol integration is irrelevant if agents can't discover your business in the first place.

Tailwind CSS demonstrates this: 75% of team laid off after traffic-dependent revenue model collapsed. The problem wasn't that agents couldn't buy Tailwind UI components - it's that agents couldn't find the paid products to recommend them.

Discovery precedes commerce. Fix discoverability first (semantic HTML, structured data, clear information architecture), then add commerce capability.

The patterns in Chapters 2-9 solve discovery. Protocol integration (covered in Chapter 10) enables commerce. You need both.

When Competitors Cooperate

Target and Walmart jointly endorsed Google's Universal Commerce Protocol despite being direct competitors. When direct competitors cooperate on standards, the underlying problem is urgent and industry-wide.

Your competitors are facing the same invisible failures. The question is who acts first to fix them.

What This Means for You

If you sell products:

- Agent traffic is high-conversion
- Implementation pays for itself quickly
- First-mover advantage matters
- Protocol integration becomes essential

If you publish content:

- Advertising model is under pressure
- Alternative monetisation required
- API access, subscriptions, or paid services
- Discovery remains valuable even if traffic drops

If you provide services:

- Agent-mediated booking is growing
- Clear pricing and availability required
- Structured data enables recommendations
- Identity delegation solves customer relationship

If you're B2B SaaS:

- Pricing models need rethinking
- API access becomes table stakes
- Agent-friendly is competitive advantage
- Documentation structure matters for discovery

The Timeline

The platforms have committed. The retailers have endorsed. The timeline has compressed.

Q1 2026: Major platform launches (complete)

Q2-Q3 2026: Rapid adoption by early movers

Q4 2026: Agent traffic reaches 10-15% for commerce sites

2027: Agent recommendations become primary discovery channel for routine purchases

You don't have years to prepare. You have months.

Next Steps

1. **Test your site** (Chapter 8 methodology) - Understand current visibility
2. **Implement foundation patterns** (Chapters 2-7) - Fix discovery first
3. **Measure impact** (Chapter 8 testing) - Validate improvements work
4. **Consider protocol integration** (Chapter 10) - Add commerce capability
5. **Monitor industry** - Standards are evolving, stay informed at https://allabout.network/invisible-users/news.html

The opportunity is clear. Websites that work well for AI agents will increasingly capture the conversions that poorly-designed competitors lose.

For comprehensive business analysis, strategic frameworks, and industry-specific guidance, see "MX-Bible" (the complete guide). This chapter provides the essential decision-making context for technical teams. The full book explores business models, organisational change, legal implications, and strategic positioning in depth.

The End

You now understand why modern web design fails AI agents - and more importantly, how to fix it. The patterns in this handbook work because they're not tricks or hacks. They're fundamental design principles that serve all users, whether human or machine.

The field moves quickly. January 2026 saw three major platforms launch agent commerce systems within seven days - Amazon Alexa+, Microsoft Copilot Checkout, and Google's Universal Commerce Protocol. What industry analysts predicted would take years happened in months. Adobe's Holiday 2025 data shows AI referrals surged 700% in retail and 500% in travel, with conversion rates from AI-referred users now leading human traffic by 30%. This isn't future speculation. It's current infrastructure.

Platforms evolve, new agent capabilities emerge, and best practices get refined through real-world implementation. That's why the most current resources live online rather than frozen in print.

Additional Resources Available Online

Comprehensive appendices and additional materials are maintained online to ensure they remain current and relevant. These resources include practical implementation guides, quick references, and real-world case studies.

Visit: https://allabout.network/invisible-users/web/

What You'll Find Online

Implementation Guides

Appendix A: Implementation Cookbook Quick-reference recipes for common AI agent compatibility patterns. Copy-paste solutions for forms, navigation, state management, and error handling.

Appendix B: Battle-Tested Lessons Production learnings from real-world implementations. What works, what doesn't, and why. Avoid common pitfalls.

Appendix C: Web Audit Suite User Guide Complete documentation for the Web Audit Suite analysis tool. Installation, configuration, and interpreting results.

Appendix D: AI-Friendly HTML Guide Comprehensive guide to semantic HTML patterns that work for AI agents. Detailed explanations with before/after examples.

Quick References

Appendix E: AI Patterns Quick Reference One-page reference guide for data attributes and patterns. Essential for implementation teams.

Appendix F: Implementation Roadmap Priority-based roadmap for adopting AI agent compatibility. Organised by impact and effort, not time estimates.

Appendix G: Resource Directory Curated collection of 150+ resources: standards, tools, articles, and communities. Kept up-to-date with latest developments.

Case Studies and Examples

Appendix H: Example llms.txt File Working example of an llms.txt file following the llmstxt.org specification. Template for your own implementation.

Appendix I: Pipeline Failure Case Study Detailed analysis of a £203,000 AI agent error. How poor form design caused pipeline failure and what to learn from it.

Appendix J: Industry Developments Latest news and updates about AI agents, commerce platforms, and industry shifts. Regularly updated with verified sources.

Appendix K: Common Page Patterns Production-ready HTML templates demonstrating AI-friendly patterns for common page types. This appendix provides complete, copy-paste HTML for eight essential page types: home pages, about pages, contact pages, sales pages, collection pages, article pages, FAQ pages, and form pages. Each template includes semantic HTML structure, Schema.org JSON-LD, explicit state attributes, AI meta tags, accessible markup, and real content examples. All templates follow Chapter 11 patterns and are ready for immediate deployment.

Online Features

Each appendix page includes:

- Table of contents
- Full content with syntax highlighting
- Navigation between appendices
- Responsive design (mobile-friendly)

Continuing Your Journey

The appendices provide implementation details, but you have three additional resources to deepen your expertise and take practical action.

Practical Implementation Support

Understanding the patterns is one thing. Applying them to existing sites is another.

Web Audit Suite Analysis Service

A production-ready implementation tool has been created by Digital Domain Technologies Ltd.

The Web Audit Suite implements the patterns described in this book, providing comprehensive website analysis for AI agent compatibility, SEO, accessibility, performance, and security.

Complimentary Analysis Offer I offer an analysis of four pages from your website, providing detailed insights into AI agent compatibility, accessibility issues, and practical improvement recommendations. This analysis uses the Web Audit Suite tool documented in Appendix C and demonstrates the patterns covered throughout this book.

The analysis includes:

- AI agent compatibility assessment
- WCAG 2.1 AA accessibility compliance review
- Schema.org structured data validation
- Performance and security baseline metrics
- Prioritised recommendations with implementation guidance

Contact me for details and pricing for extended analysis:

- Email: tom.cranstoun@gmail.com
- LinkedIn: https://www.linkedin.com/in/tom-cranstoun/
- Website: https://allabout.network

Deeper Learning

The Companion Book - MX-Bible

This handbook is part of a two-book series. Whilst this book focuses on practical implementation patterns for developers and designers, the companion volume - MX-Bible - provides comprehensive theoretical coverage across 13 chapters.

MX-Bible explores the full landscape of Machine Experience (MX) and AI agent compatibility, establishing MX as the master discipline that improves SEO, accessibility, and performance as side effects. The key insight: patterns that help AI agents (who cannot see visual cues) also help screen reader users, keyboard navigators, and search engines. One implementation serves multiple audiences.

The companion book covers:

- **What AI agents actually are** - Technical capabilities, limitations, and how they differ from human users
- **The 5-stage agent journey** - Discovery, Citation, Compare, Pricing, Confidence (miss any stage and the entire chain breaks)
- **Served vs Rendered HTML** - Why you must design for both states to support all agent types
- **The convergence principle** - How fixing structure for zero-tolerance agents automatically improves accessibility
- **Entity Asset Layer** - Sovereign, portable ownership of digital identity and commerce capabilities
- **Universal Commerce Protocol** - Open standard for agent-mediated ecommerce transactions
- **Real-world case studies** - The £203,000 pricing error, production audit findings, silent failure patterns
- **Implementation patterns** - Semantic HTML, Schema.org structured data, explicit state management
- **Testing strategies** - Validating agent compatibility across CLI, server-based, and browser agents
- **Strategic guidance** - Business case, ROI analysis, first-mover advantage, computational trust

The books share the same appendices and code examples, allowing readers to choose the format that best suits their needs. Implementation teams benefit from this handbook's focused approach, whilst strategists and architects gain deeper insight from the comprehensive guide.

Both books are available through standard channels. Visit https://allabout.network for details.

Access Information

Last updated: January 2026
Author: Tom Cranstoun
LinkedIn: https://www.linkedin.com/in/tom-cranstoun/
Contact: tom.cranstoun@gmail.com
Website: https://allabout.network

Copyright and Usage

© 2026 Tom Cranstoun. All rights reserved.

- **Content Usage:** All rights reserved, not licensed for public distribution
- **Attribution Format:** "MX-Handbook by Tom Cranstoun"

Thank you for reading. You now have the knowledge to build interfaces that work for all visitors - human and machine alike. The patterns in this handbook aren't theoretical. They're battle-tested approaches that improve accessibility, reduce maintenance burden, and prepare your sites for the agent-driven future.

The convergence principle means you're not creating separate experiences or adding cost centres. You're fixing underlying structure so it works for everyone. Semantic HTML that helps agents also helps screen readers. Structured data that enables agent comparison also powers search engine rich results. Explicit state that gives agents confidence also clarifies navigation for keyboard users. One implementation serves multiple audiences.

AI agents are visiting your site right now. When they succeed, they build computational trust and return for future queries. When they fail, they disappear from recommendations permanently - no analytics warning, no second chance, no angry email explaining what went wrong. Sites that work early become trusted sources. Sites that fail early become invisible.

The invisible users are here. Now you can see them.

Tom Cranstoun April 2026