ChatGPT

# Communicating Website Capabilities to AI Agents

Modern websites can expose machine-readable descriptions of their capabilities, features, and resources to help AI agents (including LLM-powered chatbots) understand and interact with them. This is achieved through standard files, endpoints, and metadata that advertise what the site offers and how to use it. Below, we outline the key methods and specifications, their purposes and structures, and what a checker tool should verify for each.

## The `.well-known/` Directory for Site Metadata

Many protocols use the special `/.well-known/` path to host site-wide metadata and configuration. This convention (defined by RFC 8615) provides a consistent location for information that user-agents or clients can discover before making requests [1] [2]. In practice, a site creates specific files under `/.well-known/` for different purposes, for example:

- **OpenID Connect Discovery** – An identity provider (OAuth/OIDC server) will expose `/.well-known/openid-configuration`, a JSON document listing its OAuth 2.0 endpoints (authorization, token, user info), supported scopes, public keys (JWKS URL), etc. [3]. This allows clients to automatically discover how to authenticate with the service. The JSON structure includes fields like `authorization_endpoint`, `token_endpoint`, `issuer`, `jwks_uri`, and more, following the OIDC Discovery spec.
- **App Links and Association Files** – Mobile app integration files such as `apple-app-site-association` (for iOS Universal Links) and `assetlinks.json` (for Android Digital Asset Links) are served in `/.well-known/`. These JSON files let the site declare that certain app IDs are associated with the domain [4] [5]. For instance, they allow a URL clicked on a mobile device to open in the corresponding native app if installed.
- **Security Policy (security.txt)** – A `/.well-known/security.txt` file is a plain text file (per a proposed standard) where a website lists contact information and instructions for security researchers (e.g. how to report vulnerabilities) [6]. This is a way to advertise the site's security contacts and policy in a standardized format.
- **AI Plugin Manifests** – (Detailed in the next section) ChatGPT and similar AI systems look for `/.well-known/ai-plugin.json` on a site to learn how to interact with its APIs.

**Purpose:** The `.well-known` directory serves as a discovery mechanism for site capabilities or policies. By checking a known URI, an agent can retrieve structured info about the site (authentication, API integration points, app associations, etc.) without prior knowledge of the site's URL structure [1] [2].

**Structure:** Each `.well-known` file has its own format defined by its respective standard. For example, the OpenID Connect config is a JSON with keys specified by the OIDC standard (as mentioned above), whereas `security.txt` is a plain text with fields like "Contact:" or "Acknowledgments:". All `.well-known` URIs are registered with IANA to avoid naming collisions (the IANA registry lists entries like `ai-plugin.json`, `openid-configuration`, etc.) [7].

**Checker considerations:** A checker tool should attempt to request relevant `.well-known` URLs on the site and verify: (1) that the resource is present (HTTP 200) and not blocked, (2) that it is correctly formatted (valid JSON, etc.), and (3) that required fields are present. For example, if the site is expected to support OIDC, the checker should confirm that `/.well-known/openid-configuration` exists and contains valid endpoints. If the site has a ChatGPT plugin, it should find `ai-plugin.json` (see below). Any missing or malformed `.well-known` files would be flagged to the developers.

## AI Plugin Manifest (`/.well-known/ai-plugin.json`)

When integrating a website's API with an LLM-based assistant (such as via ChatGPT plugins), the site provides a **plugin manifest** file. This is a JSON document located at `https://your-domain/.well-known/ai-plugin.json` [8]. The manifest communicates to the AI agent what the plugin can do and how to use it. In OpenAI's ChatGPT plugin system, the manifest is one of the core components (along with the actual API and an OpenAPI spec) [9] [8].

**Purpose:** The plugin manifest describes the plugin's capabilities, API endpoints, and authorization requirements to the LLM. It contains information that helps ChatGPT decide *when* and *how* to invoke the plugin during a conversation. For example, it includes a human-facing description (for the plugin store) and a model-facing description that tells the AI what the plugin can do, so the AI can incorporate that knowledge into its prompt reasoning [10] [11].

**Structure:** The manifest follows a specific schema (currently versioned as `v1`). An example minimal manifest might look like:

```
{
  "schema_version": "v1",
  "name_for_human": "TODO Plugin",
  "name_for_model": "todo",
  "description_for_human": "Plugin for managing a TODO list.",
  "description_for_model":
"Plugin for managing a TODO list. You can add, remove and view your TODOs.",
  "auth": { "type": "none" },
  "api": {
    "type": "openapi",
    "url": "https://example.com/openapi.yaml",
    "is_user_authenticated": false
  },
  "logo_url": "https://example.com/logo.png",
  "contact_email": "support@example.com",
  "legal_info_url": "https://example.com/legal"
}
```【2†L69-L77】【2†L78-L87】

Key fields include:

- `name_for_human` / `name_for_model` – Human-readable name and a short

identifier for the model (no spaces)【2†L100-L108】. The model uses the `name_for_model` as a namespace when reasoning about the plugin.
- `description_for_human` / `description_for_model` – A brief description shown to users, and a longer description (up to ~8000 chars) for the AI itself【2†L127-L135】. The model-facing description is essentially a prompt telling the AI what the plugin can do (it should enumerate capabilities without instructing the AI when to use them).
- `auth` – The authentication scheme required. Examples: `"none"` (no auth), `"user_http"` (user-specific API keys or OAuth), or `"service_http"` (service-level auth)【2†L158-L166】. This tells the AI (and the plugin runtime) how to handle auth (e.g. whether user tokens are needed).
- `api` – This section points to the API specification. It includes the `type` (currently always `"openapi"` for OpenAI plugins) and a URL to the OpenAPI document (JSON or YAML) that defines the API endpoints【2†L146-L154】. The `is_user_authenticated` flag indicates if API calls will include user-specific auth tokens or not.
- `logo_url`, `contact_email`, `legal_info_url` – Metadata for the plugin listing (icon for the plugin, a support email, link to terms of service or privacy policy).

**Checker considerations:** To verify an AI plugin manifest, a tool should retrieve `/.well-known/ai-plugin.json` and ensure it's valid JSON and conforms to the expected schema. It should check that required fields (like all the above) are present and properly formatted (e.g. URLs are reachable, email is valid, etc.). The tool might attempt to fetch the `api.url` (OpenAPI spec) to confirm it's accessible and parseable (see next section). It could also verify the `logo_url` returns an image. Essentially, the checker should confirm that if an LLM were to attempt installation of this plugin, all pieces (manifest, API spec, authentication details) are in place and functional. Any errors (like a missing field, an invalid JSON, or a dead link in the manifest) should be highlighted as issues.

## Robots.txt: The Robots Exclusion Protocol

The `robots.txt` file is a conventional way for websites to communicate *crawling rules* to bots and crawlers. Placed at the site's root (e.g. `https://www.example.com/robots.txt`), this plain text file tells automated agents which URLs can or cannot be fetched from the site [12]. Search engines and other well-behaved web robots will read this file before crawling.

**Purpose:** Robots.txt primarily serves to manage crawler access and prevent overload on the site【8†L419-L427】. It's used to exclude sections of the site that are private, duplicate, or otherwise unwanted in indexing. By specifying disallowed paths for certain user-agents, a site can say "please don't crawl these areas." It is not an access control mechanism, but rather a voluntary guideline; compliant bots (like Googlebot, Bingbot, etc.) will honor it, while malicious bots might ignore it【26†L197-L204】.

```
In the context of AI agents, robots.txt can indicate if parts of the site (or the entire site) should not be scraped for content (for example, some sites block known AI data crawlers via this file【26†L299-L308】).

**Structure:** The file consists of records, each targeting a specific robot by its user-agent string. Basic directives include `User-agent` (to specify the bot) and `Disallow` or `Allow` (to specify URL path patterns that are disallowed or allowed). For example:

```txt
User-agent: *
Disallow: /private/

User-agent: Googlebot
Disallow: /no-google/
```

In the above snippet, all bots (`*`) are disallowed from crawling the `/private/` directory, and Google's bot specifically is additionally disallowed from `/no-google/`. A `Disallow:` with an empty value means "allow everything" (some sites explicitly allow all for certain bots). The rules are read top-down; usually a bot will find the record for its name (or the wildcard `*` if no specific match) and follow those directives [13] [14].

Robots.txt can also include a `Sitemap:` directive that provides the URL of the site's XML sitemap [15]. For instance, a line `Sitemap: https://www.example.com/sitemap.xml` tells crawlers where to find the sitemap (see next section).

**Checker considerations:** A checker should verify that the `robots.txt` file (if present) is accessible and correctly formatted. It might check for common mistakes (like invalid wildcard patterns or typos in directives). Importantly, the tool should flag if the robots.txt is overly restrictive – for example, a blanket `Disallow: /` for all user-agents would prevent any crawling (appropriate only if the site truly should not be indexed). For a manager-facing report, the checker could highlight which sections of the site are blocked from bots. If the goal is to allow AI agents (like search engine bots or GPT's web browser) to access content, the checker ensures that the content is not unintentionally disallowed. Additionally, the checker can extract any `Sitemap:` URLs from robots.txt and validate that those sitemap files are reachable.

## Sitemap.xml: XML Sitemaps for Content Discovery

An XML sitemap is a file where a website provides a list of its important pages in XML format. It acts as a **catalog of URLs** that search engines or other crawlers can use to find content on the site. Typically, the sitemap is located at `/sitemap.xml` or a similar URL, and its location can be advertised via robots.txt.

**Purpose:** The sitemap informs bots about which pages exist on the site, how often they change, and their relative importance. This helps ensure all key content is discovered and indexed, even if some pages are not well-linked. In essence, it's a roadmap for crawlers to navigate the site's content. This is very useful for search engine indexing, and by extension, any AI that crawls the web for information can leverage sitemaps to discover resources. As one definition puts it: an XML sitemap is a file listing the URLs of a website along
```

with metadata about each URL (last modified date, priority, change frequency), **helping search engines understand and crawl the site** [16].

**Structure:** The sitemap XML has a root `<urlset>` element (with a specific namespace). Inside are multiple `<url>` entries, each representing a page. A simple example of one entry:

```
<url>
  <loc>https://www.example.com/products/item123</loc>
  <lastmod>2025-09-30</lastmod>
  <changefreq>weekly</changefreq>
  <priority>0.8</priority>
</url>
```

Here, `<loc>` is the page URL. `<lastmod>` is the date it was last updated (in `YYYY-MM-DD` or full timestamp format). `<changefreq>` and `<priority>` are optional hints about how frequently the page changes and how important it is relative to other URLs (priority is a value between 0.0 and 1.0). Sitemaps can also be split into multiple files and indexed by a sitemap index file if there are tens of thousands of URLs. The sitemap protocol (defined on sitemaps.org) allows for up to 50,000 URLs per file, and multiple sitemaps can be referenced via a central index.

**Checker considerations:** The checker should verify that the sitemap exists and is accessible (HTTP 200, correct XML content type). It should then parse the XML to ensure it's well-formed and valid against the sitemap schema. Potential checks include: Are the `<loc>` URLs on the correct domain (sitemaps should usually list URLs from the same site)? Is the date format valid in `<lastmod>`? If multiple sitemaps are used, are they properly listed in a sitemap index file? The tool might also report the number of URLs found and possibly spot-check a few (e.g., do they return a 200 status code). For a non-technical manager, the checker could provide a visualization or count of how many pages are listed, indicating coverage of the site. Additionally, the checker should confirm that the sitemap URL is mentioned in `robots.txt` (or advise to do so if not) for maximal visibility [17]. If the sitemap is missing or has errors (malformed XML, unreachable URLs), the tool should flag this as something to fix, since it could impair the site's discoverability by AI and search engines.

## API Descriptions (OpenAPI/Swagger Specifications)

If the website offers a web API, providing an **OpenAPI (Swagger) specification** is a best practice to communicate the API's capabilities to both developers and AI agents. OpenAPI is a standardized, language-agnostic format (JSON or YAML) for describing RESTful APIs – including their endpoints, methods, request parameters, responses, and authentication.

**Purpose:** An OpenAPI spec allows machines (and humans) to **discover and understand the service's capabilities without needing to read custom docs or code** [18]. For AI agents, this means the agent can be given the API spec and know what operations are possible. For example, in the ChatGPT plugin scenario, the plugin manifest points to an OpenAPI file, which ChatGPT uses to formulate calls to the API. More generally, any tool or "AI checker" can parse the OpenAPI to see what endpoints exist and how to use them. This is crucial for integration: the spec defines the contract of the API.

**Structure:** An OpenAPI document (version 3.x is current) begins with a top-level `openapi` version and an `info` section (title, version, description). It then has a `paths` object that enumerates each API endpoint (URL path) and the operations (GET, POST, etc.) supported on that path, along with parameters and response formats. For example, a snippet in YAML format:

```
openapi: 3.0.1
info:
  title: Example API
  version: "1.0"
paths:
  /items:
    get:
      summary: List all items
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Item'
```

In this snippet, the API has a GET endpoint `/items` which returns a JSON array of `Item` objects (with a schema defined elsewhere). The spec can include complex schemas for data models under `components/schemas`. It can also describe authentication (e.g., an `securitySchemes` section might say it uses API key or OAuth2 bearer tokens). In the context of a ChatGPT plugin, currently the OpenAPI spec is required to have a `servers` entry pointing to the API's base URL, and every operation the plugin might call must be defined here.

**Checker considerations:** A checker tool should verify the presence and correctness of the OpenAPI/Swagger spec. If the site advertises an OpenAPI URL (for instance, the plugin manifest's `api.url` field or a link on the site), the tool would fetch that spec and attempt to parse it. It should validate that the JSON/YAML conforms to the OpenAPI schema (there are validators for this). It could also check for completeness – e.g., does the spec have a `200 OK` response defined for each path, does it list all expected endpoints, are the parameter schemas valid? If the site's API requires auth, the checker should see that the spec defines a security scheme. For example, if the manifest says `auth: none` but the OpenAPI spec has auth requirements, that's a discrepancy the tool could warn about. The checker might also test a couple of endpoints (if safe to do so) to see if calling them matches the spec (though that crosses into API testing). At minimum, from a documentation perspective, the tool should ensure the OpenAPI file is accessible and *usable*. Because the OpenAPI spec is so central to AI agents using the API, any errors (like invalid format or wrong URL) will break the integration – the tool should highlight those.

*(Note:* Additionally, some sites may employ **HTTP headers** to point to their API spec – for instance using the `Link` header with relation types defined in RFC 8631. A header like \ `Link: <https://api.example.com/openapi.json>; rel="service-desc"` could be present, indicating a machine-

readable description (service description) is available [19]. A robust checker might look at response headers on the main site or API endpoints for such clues, though this is less common.)

## Schema.org Structured Data (JSON-LD Microdata)

Beyond APIs, websites communicate their content and services through **structured data markup** on HTML pages. The de facto standard vocabulary is Schema.org, often embedded using JSON-LD in script tags. This markup gives semantic context to page content – effectively teaching AI (or search engine algorithms) what the content *means* (not just what it says).

**Purpose:** Structured data provides explicit clues about the meaning of a page to search engines and other consumers [20]. For example, if you have a product page, adding Schema.org markup could tell a crawler: "This page is about a Product with name X, price $Y, availability InStock, and an average rating of 4.5". Similarly, a blog article page can declare the headline, author, publish date, etc. This information is used by search engines to create rich search results (like rich snippets, knowledge panels, etc.) and can be leveraged by AI systems to answer queries more accurately. Google uses JSON-LD structured data to, for instance, display recipe information (ingredients, cook time) directly in search results [21]. In an AI chat context, one can imagine an LLM that has indexed such data can respond with precise answers (e.g., "the price of product X is $Y as of yesterday") thanks to structured data.

**Structure:** Schema.org provides a vocabulary of types (Product, Article, Organization, FAQ, Event, etc.). JSON-LD is a common format to embed this. It's usually placed in the `<head>` or at the bottom of the HTML. Example of JSON-LD for an organization profile:

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "Organization",
  "name": "Example Corp",
  "url": "https://www.example.com",
  "logo": "https://www.example.com/logo.png",
  "sameAs": [
    "https://twitter.com/example",
    "https://www.linkedin.com/company/example"
  ]
}
</script>
```

This snippet tells any agent that the website is about "Example Corp" (an Organization), and gives its official URL, logo, and social media profiles. Another example: a product page might use `"@type": "Product"` with properties like `"sku"`, `"price"`, `"aggregateRating"`, etc. The JSON-LD format is simply key-value pairs following the schema definitions. Some sites also use Microdata or RDFa (inline annotations in HTML) for structured data, but JSON-LD is preferred for ease of adding and not affecting page rendering.

**Checker considerations:** The checker tool should scan a representative set of pages for structured data. It could look for `<script type="application/ld+json">` blocks and parse them as JSON. For each, it should verify that the JSON is valid and conforms to known schema.org types (for example, using Google's Structured Data Testing Tool API or a library). It should check required properties for each type (Google's guidelines often list mandatory fields for a type to be eligible for rich results [22]). If errors are found (e.g., invalid JSON, or using a schema.org type incorrectly), the tool would report them. It might also summarize what types of structured data were found (e.g., "Product schema on 50 pages, Article schema on 10 pages, Organization schema on homepage, etc."). This could be visualized for a manager to see if the site is exposing the expected info. If no structured data is found at all, and the site would benefit from it, the tool might recommend adding it. In short, the checker ensures that the site's content is enriched with machine-readable semantics, which is increasingly important in the era of AI-driven search [23] [24].

## Relevant HTTP Headers and Additional Manifest Files

In addition to the above, there are a few other channels through which a site can advertise capabilities or preferences to automated agents:

- **HTTP Headers (Robots and Service Info):** While the `robots.txt` file handles most crawler directives, individual pages can send a header `X-Robots-Tag` with values like `noindex` or `nofollow`. This header serves the same purpose as an HTML `<meta name="robots" content="...">` tag, but can apply to non-HTML resources as well (like PDF files) [25]. A checker might inspect HTTP responses (especially for important pages or file downloads) to see if any `X-Robots-Tag` is present and whether it's advising bots not to index or follow. It should confirm that these headers align with the intended policy (e.g., if the site *wants* to be indexed, make sure there's no accidental `noindex` header).

- **HTTP Headers (Link relations):** As mentioned, the HTTP `Link` header can be used to advertise related resources. Aside from sitemaps (some sites use `<link rel="sitemap">` in HTML or headers), two notable relations from RFC 8631 are `service-desc` and `service-doc`. A `Link` header with `rel="service-desc"` might point to an OpenAPI definition URL [19], and `rel="service-doc"` could point to human-readable API documentation [26]. For example: \ `Link: <https://api.example.com/openapi.yaml>; rel="service-desc", <https://docs.example.com/api-guide>; rel="service-doc"`. \ A checker can look at the headers of the homepage or known API endpoints for such links and verify that they are correct and accessible. While not ubiquitous, this is a standardized way to make APIs discoverable to clients automatically, and an AI agent or developer tool could leverage it.

- **Web App Manifest:** Websites that function as Progressive Web Apps (PWAs) provide a **web app manifest** (typically a JSON file named `manifest.json` or `site.webmanifest`). This file, usually linked from HTML (`<link rel="manifest" href="/manifest.json">`), describes the web app's name, icons, theme colors, start URL, and other settings for installation on devices [27] [28]. Its purpose is more for user experience (installing the site like an app), but it is a form of machine-readable capability description – it tells the browser/OS what the app can do (like orientation lock, offline capabilities via service worker, etc.). A checker can ensure this manifest file is present and valid JSON, and that it contains important fields (like `name`, `short_name`, `icons`, `start_url`) required for a good PWA install experience. For instance, if a manager wants to know "Do we have a

PWA manifest and is it correct?", the checker would report on that. While not directly used by LLMs, having a proper manifest indicates a level of site capability (it might be relevant if an AI agent is evaluating how mobile-friendly or app-like the site is).

- **Other Headers:** The checker might also verify headers that, while not describing capabilities per se, affect how an AI or any agent can access the site. For example, **CORS headers** (`Access-Control-Allow-Origin`) are important if the AI agent is running in a web context (like a browser extension or embedded tool) and needs to call the site's APIs. A developer should ensure that public APIs have appropriate CORS headers so that tools can actually use them. Another header is **Content-Type** – ensuring APIs return `application/json` for JSON data, etc., which a client would expect. The tool could flag mismatches (e.g., an API endpoint returning JSON but with `text/html` header could confuse clients). These are more on the technical side, but a comprehensive checker would include them to guarantee that any consuming agent (AI or otherwise) can properly interact with the site's resources.

**Checker considerations:** In summary, the checker should aggregate all these additional signals – it would fetch a page (or a HEAD request) and record relevant headers like `X-Robots-Tag`, `Link`, `Content-Type`, and confirm they meet best practices. For the manifest, it should attempt to retrieve the manifest URL if one is linked and validate its content (perhaps even checking that icons listed are accessible). All this can be presented as a list of "additional integrations": e.g., *Web App Manifest: present/absent, X-Robots-Tag: none found (good if indexing is desired), Link headers: found API description link (verified OK), etc.*.

## Conclusion

By implementing the above standards, a website makes itself more transparent and accessible to AI-driven services. **Capabilities files** like `ai-plugin.json` or OpenAPI let an AI agent know *how to interact* with the site's APIs. **Content descriptors** like sitemaps and schema.org data help AI *find and understand* the site's content. **Policies and preferences** expressed via robots.txt or related headers tell AI what is *allowed* or *restricted*.

For a developer building a checker tool, the focus should be on verifying that each of these features is present (if applicable) and correctly configured. The checker's output for a non-technical manager could be a dashboard or report highlighting, for example: *"Robots.txt: **Found**, no critical issues; Sitemap: **Found**, 120 URLs listed; OpenAPI: **Found**, validated OK; AI Plugin Manifest: **Not found** (recommended if integrating with chatbots); Structured Data: **Found** on 80% of pages (types: Article, Product); Web App Manifest: **Found**, OK"*. Such a report demystifies the site's readiness for AI integration and highlights any gaps.

Ensuring these pieces are properly exposed and functional not only helps AI agents and chat interfaces interact with the site, but also generally improves the site's interoperability and SEO. In short, **clear communication of a website's capabilities through standard specs and files is becoming essential in a landscape where AI tools autonomously consume and act on web content**. The above guide provides a blueprint for implementing and checking these features comprehensively and clearly.

**Sources:** The information and examples above draw on standard web specifications and current best practices, including OpenAI's plugin documentation, Google Search Central guidelines, and relevant RFCs and schema references [8] [18] [20] [1] [16] (full citations provided in context).

1 2 4 5 6 http - What is the purpose of the ".well-known"-folder? - Server Fault
https://serverfault.com/questions/795467/what-is-the-purpose-of-the-well-known-folder

3 OpenID Connect (OIDC) on the Microsoft identity platform - Microsoft identity platform | Microsoft Learn
https://learn.microsoft.com/en-us/entra/identity-platform/v2-protocols-oidc

7 Well-known URI - Wikipedia
https://en.wikipedia.org/wiki/Well-known_URI

8 9 10 11 What Is the ChatGPT Plugin Manifest? | HackWithGPT
https://www.hackwithgpt.com/blog/what-is-the-chatgpt-plugin-manifest/

12 13 14 15 25 robots.txt - Wikipedia
https://en.wikipedia.org/wiki/Robots.txt

16 17 XML sitemap - definition, explanation + SEO tips
https://www.link-assistant.com/seo-wiki/xml-sitemap/

18 OpenAPI Specification - Version 3.1.0 | Swagger
https://swagger.io/specification/

19 26 RFC 8631 - Link Relation Types for Web Services
https://datatracker.ietf.org/doc/html/rfc8631

20 21 22 Intro to How Structured Data Markup Works | Google Search Central | Documentation | Google for Developers
https://developers.google.com/search/docs/appearance/structured-data/intro-structured-data

23 Structured Data in the AI Search Era - BrightEdge
https://www.brightedge.com/blog/structured-data-ai-search-era

24 The role of structured data in AI Search visibility - insightland
https://insightland.org/blog/structured-data-ai-search/

27 28 Web application manifest - Progressive web apps | MDN
https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Manifest