Reliable Context

RELIABLE CONTEXT / RESEARCH & PRACTICAL GUIDANCE

Model Context Protocol (MCP) Explained

> Reliable Context is a kernel of truth: a system of record for organizations that fuse private and public information and must defend every value they act on. One canonical record per real thing, a cited source and an observation date behind every value, unknowns declared rather than left blank, and any past state of the record reproducible.

Published by Reliable Context · 2026-10-29 · model context protocol explained · what is mcp · mcp hosts clients servers · mcp tools resources prompts · mcp server

Original article: https://reliablecontext.com/articles/model-context-protocol-explained-hosts-clients-servers



In this article

1. The integration problem
2. Architecture: hosts, clients and servers
3. Server primitives: tools, resources and prompts
4. Client features and capability negotiation
5. Message format, statelessness and transports
6. Worked example: a supplier-facts server
7. What MCP does not guarantee
8. Design checklist for fact-returning servers

The integration problem

A language model is only as useful as the context it can reach. In an organization, that context is spread across document stores, databases, ticketing systems and line-of-business applications. Before a shared protocol existed, every pairing of an AI application with a data source needed its own connector. When Anthropic introduced the Model Context Protocol (MCP) on 25 November 2024, it described the problem in those terms: "Every new data source requires its own custom implementation, making truly connected systems difficult to scale", and presented MCP as "a new standard for connecting AI assistants to the systems where data lives" (Anthropic, 2024).

The current specification describes MCP as "an open protocol that enables seamless integration between LLM applications and external data sources and tools", and says it takes inspiration from the Language Server Protocol, which standardized how editors support programming languages (MCP specification, 2026-07-28). The analogy is useful: one protocol on each side replaces a matrix of one-off integrations.

This article explains MCP as an interface contract: who the parties are, what they can exchange, and how a request looks on the wire. It then asks a question the protocol leaves open: what does a server owe an agent about where a returned value came from?

Which revision this describes

MCP revisions are named by date. As of 28 October 2026, the specification's "latest" address resolves to **revision 2026-07-28**. That revision changed the protocol substantially from 2025-11-25 and earlier: it removed the `initialize` handshake and protocol-level sessions, made every request self-contained, added a mandatory `server/discover` method, and deprecated the Roots, Sampling and Logging features (MCP changelog, 2026-07-28). The specification calls revisions that use per-request metadata "modern" and those that use the `initialize` handshake "legacy" (MCP, Versioning and Compatibility). Much published material, including many tutorials, describes the legacy model. Everything below follows 2026-07-28 unless stated.

Architecture: hosts, clients and servers

MCP "follows a client-host-server architecture where each host can run multiple client instances" (MCP, Architecture). The three roles are distinct:

Role	What it is	Responsibilities in the specification
Host	The AI application the user interacts with: a chat interface, an IDE, an agent runtime	Creates and manages clients; controls connection permissions; enforces security policies and consent requirements; handles user authorization decisions; coordinates the model; aggregates context across clients
Client	A connector inside the host	Communicates with exactly one server; attaches protocol version and capabilities to every request; routes messages; maintains security boundaries between servers
Server	A program that exposes context and capabilities	Exposes resources, tools and prompts; can be a local process or a remote service; must respect security constraints

Source: MCP, Architecture.

The one-client-per-server rule matters for isolation. One of the stated design principles is that "servers should not be able to read the whole conversation, nor 'see into' other servers": the full conversation history stays with the host, and cross-server interactions are controlled by the host (MCP, Architecture). A server that returns supplier data never sees what a second server returned about contracts, unless the host chooses to pass it.

Server primitives: tools, resources and prompts

Servers offer three kinds of feature. The specification distinguishes them by **who decides when they are used**:

Primitive	Control model	Purpose	Main methods
Tools	Model-controlled: "the language model can discover and invoke tools automatically"	Functions the model can execute: query a database, call an API, compute	`tools/list`, `tools/call`
Resources	Application-driven: the host decides how to incorporate them	Context and data identified by a URI, such as files, records or schemas	`resources/list`, `resources/read`, `resources/templates/list`
Prompts	User-controlled: exposed so that "the user being able to explicitly select them"	Templated messages and workflows	`prompts/list`, `prompts/get`

Sources: Tools, Resources, Prompts.

Tools carry a `name`, an optional `title` and `description`, an `inputSchema` (JSON Schema), and optionally an `outputSchema` and `annotations` describing behavior. Results can contain unstructured `content` (text, image, audio, resource links, embedded resources) and structured content in a `structuredContent` field that conforms to the `outputSchema` if one is declared. The specification states that clients "MUST consider tool annotations to be untrusted unless they come from trusted servers", and that there "SHOULD always be a human in the loop with the ability to deny tool invocations" (MCP, Tools).

Resources are identified by a URI and have a `name`, optional `title`, `description`, `mimeType` and `size`. Their contents are returned as text or base64 binary. Resources, templates and content blocks support three optional **annotations**: `audience` (`"user"`, `"assistant"`), `priority` (0.0 to 1.0) and `lastModified` (an ISO 8601 timestamp) (MCP, Resources). Keep that list in mind; it is the only provenance-like metadata the core resource model defines.

Prompts are reusable templates with arguments. The prompts page notes that "user-controlled" refers to who decides when the prompt is used, not who authors its content (MCP, Prompts).

Client features and capability negotiation

Clients can also offer features to servers. In 2026-07-28, the active one is **elicitation**: "server-initiated requests for additional information from users", in a form mode (structured data validated against a JSON schema) or a URL mode, which directs the user to an external URL for "sensitive interactions that must *not* pass through the MCP client" (MCP specification; MCP, Elicitation). Roots (exposing file-system boundaries to a server) and

Sampling (a server asking the host's model for a completion) remain functional but are deprecated; the changelog suggests passing paths as tool parameters instead of Roots and integrating directly with model provider APIs instead of Sampling (MCP changelog).

Because servers no longer send requests to clients, a server that needs user input mid-operation uses the **Multi Round-Trip Requests** pattern: it answers with a result whose `resultType` is `"input_required"` and whose `inputRequests` describe what it needs; the client gathers the input and retries the original request, with a new JSON-RPC `id`, carrying `inputResponses` (MCP, Multi Round-Trip Requests; MCP, Tools).

Capability negotiation is per request. Clients put their capabilities in `_meta` under `io.modelcontextprotocol/clientCapabilities` on every request; servers advertise theirs in response to `server/discover`, which every server must implement and which clients may call first (MCP, Architecture; MCP, Discovery). A server "MUST NOT rely on capabilities the client has not declared" and returns a `MissingRequiredClientCapabilityError` if it needs one (MCP, Base protocol).

Message format, statelessness and transports

MCP messages are JSON-RPC 2.0, UTF-8 encoded. There are requests (with an `id`), responses (a `result` or an `error`, echoing the `id`) and notifications (no `id`) (MCP, Base protocol).

The 2026-07-28 revision is explicitly **stateless**: "all the information needed to process a request is contained in the request itself", and servers "MUST NOT rely on prior requests over the same connection to establish context (e.g., capabilities, protocol version, client identity)". State that spans requests "MUST be referenced by an explicit identifier the client passes on each request" (MCP, Base protocol). Every request must carry `io.modelcontextprotocol/protocolVersion` and `io.modelcontextprotocol/clientCapabilities` in `_meta`; a request missing them is rejected with error `-32602`.

Two standard transports are defined (MCP, Transports):

- **stdio**: newline-delimited JSON-RPC over the standard input and output of a subprocess the client launches; typical for local servers.
- **Streamable HTTP**: each message is an HTTP POST to a single endpoint; replies arrive as a JSON object or a request-scoped server-sent-events stream; typical for remote servers.

For HTTP, the specification provides an authorization framework; stdio servers instead take credentials from their environment (MCP, Base protocol).

Worked example: a supplier-facts server

Halden Therapeutics AG (a fictional company) runs an MCP server, `halden-supplier-facts`, that answers questions about approved suppliers. An agent in a procurement assistant asks for the GMP certificate expiry date of Velmar Chemie GmbH (also fictional).

1. Discover the tool. The client lists tools. This request shows the required `_meta`; the specification's own examples omit it for brevity.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/list",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": { "name": "procurement-assistant", "version": "3.2.0" },
      "io.modelcontextprotocol/clientCapabilities": {}
    }
  }
}
```

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "resultType": "complete",
    "tools": [
      {
        "name": "get_supplier_fact",
        "description": "Return one fact about an approved supplier, with its source.",
        "inputSchema": {
          "type": "object",
          "properties": {
            "supplier_id": { "type": "string" },
            "property": { "type": "string" }
          },
          "required": ["supplier_id", "property"]
        },
        "outputSchema": {
          "type": "object",
          "properties": {
            "value": { "type": "string" },
            "status": { "type": "string", "enum": ["asserted", "unknown", "conflict"] },
            "source_uri": { "type": "string" },
            "evidence": { "type": "string" },
            "observed_at": { "type": "string" }
          },
          "required": ["status"]
        }
      }
    ],
    "ttlMs": 300000,
    "cacheScope": "private"
  }
}
```

The `ttlMs` and `cacheScope` fields are required on list results in this revision (MCP changelog). The `outputSchema` is Halden's design, not part of MCP.

2. Call the tool. (The `_meta` block is omitted from here on, as in the specification's examples.)

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "get_supplier_fact",
    "arguments": { "supplier_id": "S-77", "property": "gmp_certificate_expiry" }
  }
}
```

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "resultType": "complete",
    "structuredContent": {
      "value": "2027-05-31",
      "status": "asserted",
      "source_uri": "halden://documents/gmp-cert/DE-XX-2025-0193",
      "evidence": "valid until 31.05.2027",
      "observed_at": "2026-02-10T09:47:00Z"
    },
    "content": [
      { "type": "text", "text": "{\"value\":\"2027-05-31\",\"status\":\"asserted\",...}" },
      {
        "type": "resource_link",
        "uri": "halden://documents/gmp-cert/DE-XX-2025-0193",
        "name": "GMP certificate DE-XX-2025-0193",
        "mimeType": "application/pdf"
      }
    ],
    "isError": false
  }
}
```

A tool returning structured content "SHOULD also return the serialized JSON in a TextContent block" for backward compatibility, which is why the text block repeats it (MCP, Tools). The `resource_link` lets the host fetch the source.

3. Read the source.

```
{ "jsonrpc": "2.0", "id": 3, "method": "resources/read",
  "params": { "uri": "halden://documents/gmp-cert/DE-XX-2025-0193" } }
```

The server returns the document in `contents` (here as a base64 `blob` with `mimeType` `application/pdf`), plus the required `ttlMs` and `cacheScope`.

Every field that makes this answer checkable (`status`, `source_uri`, `evidence`, `observed_at`) was defined by Halden. A different server could return `{"value": "2027-05-31"}` and be equally conformant.

What MCP does not guarantee

MCP standardizes the **channel**. It does not standardize the **evidence**. The specification is direct about its own limits: "While MCP itself cannot enforce these security principles at the protocol level", implementors should build consent and authorization flows, access controls and data protections (MCP specification). For data quality and provenance, the gaps are these:

Question an auditor asks	What core MCP provides	Gap
Where did this value come from?	Nothing specific; a server may add a `resource_link` or put fields in `structuredContent`	No standard source or evidence field
When was it true, and when was it read?	`lastModified` annotation on resources and content blocks	One timestamp, meaning "last modified", not valid time or observation time
Which server answered?	`serverInfo` in result `_meta`	The specification says `clientInfo` and `serverInfo` "are self-reported by the sender and are not verified by the protocol" and should not be relied on for security decisions (Base protocol)
Is the tool what it claims to be?	`description` and `annotations`	Annotations "MUST" be treated as untrusted unless from trusted servers (Tools)
Is "no result" a negative or a failure?	`isError` flag for tool execution errors	No standard way to say "checked, not found" versus "not checked"
Can the answer be replayed later?	Statelessness and explicit identifiers make requests self-describing	The server's data may have changed; nothing in MCP pins a version

None of this is a defect in a transport protocol. HTTP does not tell you whether a web page is true either. But it means that putting a system behind an MCP server does not make its answers defensible. Whether an agent's context can be audited depends on what the server returns, which is a server design question. The site's glossary entry on context engineering covers the wider discipline; the live article on permission-aware RAG evaluation addresses the access side.

Design checklist for fact-returning servers

For teams building MCP servers that return facts an agent will act on:

1. **Declare an `outputSchema`** and return `structuredContent`, so clients can validate results instead of parsing prose.
2. **Return the source with the value**: a stable source identifier, the exact supporting passage, and a `resource_link` to the document.

3. **Separate the times**: when the fact applies, when the source was observed, when the value was last changed. Do not overload `lastModified`.
4. **Make negatives explicit**: return `unknown` or `conflict` as statuses, with the reason, rather than an empty result or an error.
5. **Version the answer**: include a record or version identifier so that a later reviewer can retrieve the same value, as the stateless model's "explicit identifier" rule encourages.
6. **Use a vendor prefix for custom `_meta` keys**: the specification reserves the `io.modelcontextprotocol/` prefix and asks third-party extensions to use their own (Base protocol).
7. **Keep consent and authorization in the host**: the specification places user consent, tool confirmation and authorization decisions there.
8. **Track the revision**: support `server/discover`, return `UnsupportedProtocolVersionError` for versions you do not serve, and plan for the deprecations listed in the changelog.

An MCP server that fronts a system of record can only pass on the evidence that system holds; Reliable Context is designed so that every value it returns carries a cited, dated source.

Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.

1. https://www.anthropic.com/news/model-context-protocol
2. https://modelcontextprotocol.io/specification/2026-07-28
3. https://modelcontextprotocol.io/specification/2026-07-28/changelog
4. https://modelcontextprotocol.io/specification/2026-07-28/basic/versioning
5. https://modelcontextprotocol.io/specification/2026-07-28/architecture
6. https://modelcontextprotocol.io/specification/2026-07-28/server/tools
7. https://modelcontextprotocol.io/specification/2026-07-28/server/resources
8. https://modelcontextprotocol.io/specification/2026-07-28/server/prompts
9. https://modelcontextprotocol.io/specification/2026-07-28/client/elicitation
10. https://modelcontextprotocol.io/specification/2026-07-28/basic/patterns/mrtr
11. https://modelcontextprotocol.io/specification/2026-07-28/server/discover
12. https://modelcontextprotocol.io/specification/2026-07-28/basic
13. https://modelcontextprotocol.io/specification/2026-07-28/basic/transports

THE PUBLISHER

About Reliable Context

Reliable Context is a system of record for organizations that fuse private and public information and must defend every value they act on.

The guarantees

One canonical record per real thing: duplicates resolve by rule, ambiguity is shown rather than hidden, and related organizations relate without ever being merged. A verbatim, cited source and an observation date sit behind every value. Unknowns are declared as not found, blocked or not applicable with a reason, never left as an empty cell. Any past state of the record is reproducible, together with what changed since.

Where it runs

It runs inside the customer's perimeter, one database per organization. Permitted sources, model routing, reader clearance and retention are set as the customer's own policies, and every value is exportable with its chain of custody for an auditor or regulator.

What it is not

It is not a vector database or a RAG framework, not a data catalog, MDM or data-quality suite, not a CRM or data-enrichment vendor, and not a chatbot or AI assistant. Where a reader needs one of those, the honest answer is to say so and point them there.

Visit

See Reliable Context.

Reliable Context is in production use in two sectors; more than six thousand of its identity decisions were made by rule and none has been reversed. Articles published here explain general practice, cite their primary and academic sources, and name the jurisdiction of any regulatory rule they describe. They are not legal, regulatory or compliance advice for a specific organization, and requirements change.

Website: https://reliablecontext.com

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.