

MCP 2.0

Stateless, New Auth (CIMD), and Exciting Extensions

MCP spec 2026-07-28

James Ward

Agent Experience @ AWS

AAIF TC

jamesward.com

Almost two years of MCP

Revision	Headline
2024-11-05	Initial release: stdio + HTTP+SSE
2025-03-26	Streamable HTTP, OAuth 2.1
2025-06-18	Elicitation, structured tool output, RFC 9728 resource metadata
2025-11-25	CIMD, URL elicitation, experimental Tasks, icons
2026-07-28	Stateless. MRTR. Extensions (Tasks, Skills, Apps). DCR deprecated

The new revision has the most breaking changes since launch. It also simplifies a lot.

What changed for server developers

1. **Stateless**: no `initialize`, no sessions, no SSE back-channel
2. **MRTR**: elicitation and sampling become "input required" results that the client answers on retry
3. **Auth**: Client ID Metadata Documents (CIMD) replace Dynamic Client Registration
4. **Extensions**: Tasks, Skills, and Apps are now official extensions, not part of core

Deprecated: Roots, Sampling, Logging, DCR, HTTP+SSE

Part 1: Stateless

Why sessions hurt

2025-11-25 Streamable HTTP:

- `initialize` handshake, then an `Mcp-Session-Id` on every request
- Server-to-client requests (elicitation, sampling) travel on a GET SSE stream bound to that session
- So you need **sticky load balancing** or a **shared session store**
- Serverless and horizontally scaled deployments had to work around it

"Stateless mode" existed in SDKs, but you lost elicitation, sampling, and progress.

2026-07-28: every request stands alone

```
{
  "jsonrpc": "2.0",
  "id": 7,
  "method": "tools/call",
  "params": {
    "name": "add",
    "arguments": {"a": 2, "b": 3},
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {"name": "my-agent", "version": "1.0"},
      "io.modelcontextprotocol/clientCapabilities": {"elicitation": {}}
    }
  }
}
```

- Version + capabilities ride on **every** request in `_meta` (SEP-2575)
- No `initialize`, `ping`, `logging/setLevel`, `Mcp-Session-Id`, GET stream, or resumability
- `Mcp-Method` / `Mcp-Name` headers let gateways route without parsing the body (SEP-2243)

server/discover replaces initialize

```
{
  "result": {
    "supportedVersions": ["2026-07-28", "2025-11-25"],
    "capabilities": {
      "tools": {}, "resources": {},
      "extensions": {
        "io.modelcontextprotocol/tasks": {},
        "io.modelcontextprotocol/skills": {"directoryRead": true}
      }
    },
    "serverInfo": {"name": "mcp2-zio-server", "version": "1.0.0"},
    "resultType": "complete", "ttlMs": 3600000, "cacheScope": "public"}
}
```

- Servers **MUST** implement it. Clients **MAY** call it first
- Wrong version → `UnsupportedProtocolVersionError` (`-32022`) lists what the server supports
- Clients can use it to probe and fall back to `initialize` on older servers

Results are cacheable and typed

- Every result has `resultType` : `complete` , `input_required` , or (Tasks) `task`
- `tools/list` , `resources/list` , `prompts/list` , `resources/read` carry `ttlMs` + `cacheScope` (SEP-2549)
- List results no longer vary per connection, and `tools/list` order **SHOULD** be deterministic, which helps LLM prompt caching
- Change notifications move to `subscriptions/listen` , a single opt-in POST stream

No sessions, so where does state go?

Explicit handles (SEP-2567): the server mints an id and returns it, and the model passes it back as an ordinary argument.

```
@McpTool(name = "create_shopping_list", generateOutputSchema = true)
public ShoppingList createShoppingList() {
    var listId = "list_" + UUID.randomUUID().toString().substring(0, 8);
    store.put(listId, new CopyOnWriteArrayList<>());
    return new ShoppingList(listId, List.of());
}

@McpTool(name = "add_item", generateOutputSchema = true)
public ShoppingList addItem(
    @McpToolParam(description = "listId handle from create_shopping_list") String listId,
    @McpToolParam(description = "the item to add") String item) { ... }
```

The model can see the handle, reason about it, and use it across conversations. Any replica can serve it if the store is shared.

Spring AI: stateless today

```
spring.ai.mcp.server.protocol=stateless
```

```
@McpTool(description = "add two numbers")  
public int add(int a, int b) { return a + b; }
```

- Spring AI **2.1.0-M1** → MCP Java SDK **2.0.0**, which implements 2025-11-25
- `protocol=stateless` : no `Mcp-Session-Id` , GET → 405, any replica serves any request
- 2026-07-28 (discover, MRTR, `_meta`) is on the Java SDK **2.2** milestone, and Spring AI 2.1 RC1 targets it
- Stateless tools get `McpTransportContext` and cannot use `McpSyncRequestContext` (no back-channel yet)

Part 2: Multi Round-Trip Requests (MRTR)

Server-to-client requests, without a back-channel

Before: the server sent `elicitation/create` to the client **mid-call**, over the session's SSE stream.

Now (SEP-2322): the server **returns** what it needs, and the client **retries**.

```
client — tools/call book_flight —————> server
  ← resultType: "input_required"
    inputRequests: { "confirm": { elicitation/create ... } }
    requestState: "<opaque, signed>"

client — tools/call book_flight (new id) —————> server (any replica)
  inputResponses: { "confirm": { action: accept, ... } }
  requestState: "<echoed>"
  ← resultType: "complete"
```

Works for `tools/call`, `prompts/get`, and `resources/read`.

MRTR on the wire

```
{
  "result": {
    "resultType": "input_required",
    "inputRequests": {
      "confirm": {
        "method": "elicitation/create",
        "params": {
          "message": "Book a flight to Denver for $420?",
          "requestedSchema": {
            "type": "object",
            "properties": {
              "confirm": {
                "type": "boolean"
              }
            }
          }
        }
      }
    }
  }
}
```

- Keys are server-assigned. Batch several asks in one round
- Only ask for what the client declared in `clientCapabilities`
- `requestState` is **attacker-controlled input**: sign it (HMAC/AEAD), bind it to the principal + request, and give it a TTL

MRTR in zio-http-mcp: just ask

```
val bookFlight = McpTool("book_flight")
  .description("Books a flight to a destination after the user confirms")
  .handleWithContext[Any, ToolError, FlightInput, String]: (in, ctx) =>
    ctx.elicit("confirm", s"Book a flight to ${in.destination} for $$420?", confirmSchema)
      .map: answer =>
        if answer.action == "accept" then s"Booked: flight to ${in.destination}"
        else s"Not booked (${answer.action})"
```

- Modern request → `input_required`. The handler replays on retry with the answer in hand
- Legacy request → the **same code** sends `elicitation/create` over SSE
- `requestState` is signed by default. Share a secret across replicas with `McpRequestStateStore.signed(...)`

Interop: old and new, same endpoint

A server can speak both revisions on one URL: `initialize` → legacy session, `_meta.protocolVersion` → stateless.

Client	Server	Negotiated
zio-http-mcp (modern)	zio-http-mcp	2026-07-28 via server/discover
zio-http-mcp (modern)	Spring AI 2.1.0-M1	falls back to 2025-11-25
Spring AI 2.1.0-M1	zio-http-mcp	2025-11-25 , elicitation over SSE

Keep serving the old revision during the transition. Your clients will lag behind your servers.

Demo: Era Negotiation, Elicitations

```
cd scala-demo
./sbt "runMain mcp2.Main"      # dual-era server on :8081

npx @modelcontextprotocol/inspector
```

1. `server/discover`
2. `tools/list`: `resultType`, `ttlMs`, `cacheScope`
3. `tools/call` with no handshake
4. MRTR: `input_required` → retry with `inputResponses`

Part 3: New Auth (CIMD)

MCP auth recap

The MCP server is an **OAuth 2.1 Resource Server**. It does not issue tokens.

1. Client calls `/mcp` → **401** + `WWW-Authenticate: Bearer resource_metadata="..."`
2. Client reads **Protected Resource Metadata** (RFC 9728) → `authorization_servers`
3. Client reads **AS metadata** (RFC 8414)
4. **Client registration** ← *this is what changed*
5. Authorization code + **PKCE** + `resource=` (RFC 8707) → token whose audience is this server
6. Retry with `Authorization: Bearer ...`

`2026-07-28` also requires validating the RFC 9207 `iss` response parameter and binding credentials to their issuer.

The client registration problem

MCP clients and servers usually have **no prior relationship**. How does the AS know which client it is talking to?

Mechanism	Status in 2026-07-28	Problem
Pre-registration	OK	Doesn't scale to "any client × any server"
Dynamic Client Registration (RFC 7591)	Deprecated	Open registration endpoint, unbounded client DB, no stable identity
Client ID Metadata Documents	Preferred	

Client priority: pre-registered → **CIMD** (if `client_id_metadata_document_supported`) → DCR → ask the user

CIMD: the `client_id` is a URL

```
GET https://www.cimd.now/8085/authorize/oauth2/code/secure
```

```
{
  "client_id": "https://www.cimd.now/8085/authorize/oauth2/code/secure",
  "client_name": "cimdnw",
  "redirect_uris": ["http://localhost:8085/authorize/oauth2/code/secure"],
  "grant_types": ["authorization_code", "refresh_token"],
  "token_endpoint_auth_method": "none",
  "application_type": "native"
}
```

- The AS fetches the document when it sees a URL `client_id`. No registration call
- `client_id` **must equal** the document URL exactly. Serve it as `application/json`
- Identity is the HTTPS origin, which is portable across authorization servers
- [cimd.now](https://www.cimd.now) generates documents for localhost dev clients

Server side: unchanged, and audience is not optional

```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) {
    return http
        .authorizeHttpRequests(auth -> auth.anyRequest().authenticated())
        .with(mcpServerOAuth2(), mcp -> mcp
            .authorizationServer("https://login.jamesward.dev")
            // RFC 8707: reject tokens minted for some other server (off by default!)
            .validateAudienceClaim(true))
        .build();
}
```

`org.springframework.security:mcp-server-security:0.1.14` serves PRM, sends the 401 challenge, and validates JWTs.

CIMD is a concern of the **client and the AS**. A resource server doesn't care how the client registered, but it must check `aud`.

Client side: Spring AI + CIMD

```
@Bean
McpOAuth2CimdClientManager cimdClientManager(McpClientRegistrationRepository repo) {
    var manager = new DefaultMcpOAuth2CimdClientManager(discovery, repo, urlValidator);
    manager.setClientRegistrationCustomizer(reg -> ClientRegistration.withClientRegistration(reg)
        .clientId("https://www.cimd.now/8085/authorize/oauth2/code/" + reg.getRegistrationId())
        .build());
    return manager;
}

@Bean
OAuth2CimdHttpClientTransportCustomizer transportCustomizer(...) {
    return new OAuth2CimdHttpClientTransportCustomizer(authorizedClientManager, repo, cimdClientManager);
}
```

mcp-client-security:0.1.14 : on a 401 it discovers PRM and AS metadata, then uses the CIMD URL as the `client_id`. PKCE and `resource=` are added for you.

Demo: CIMD end to end

```
cd java-demo
./gradlew :secure-server:bootRun      # resource server :8090

npx @modelcontextprotocol/inspector

./gradlew :cimd-client:bootRun        # web app :8085
open http://localhost:8085            # log in as demo / pw
```

```
whoami → user=demo aud=[http://localhost:8090/mcp]
```

The AS is login.jamesward.dev (Spring Authorization Server + `mcp-authorization-server` with `cimd(true)`).

Part 4: Extensions

Extensions are now first class

- `capabilities.extensions` on both client and server, keyed by `{vendor-prefix}/{name}`
- Official: `io.modelcontextprotocol/*`, in `ext-*` repos with their own SEPs and release cadence
- Core stays small. Features evolve without a spec revision

Extension	Id
MCP Apps	<code>io.modelcontextprotocol/ui</code>
Tasks	<code>io.modelcontextprotocol/tasks</code>
Skills over MCP	<code>io.modelcontextprotocol/skills</code>
OAuth client credentials, Enterprise-managed auth	<code>ext-auth</code>

MCP Apps

Interactive UI rendered inline in the conversation.

- A tool points to a `ui://` resource in `_meta.ui.resourceUri`
- The resource is HTML (`text/html;profile=mcp-app`), sandboxed in an iframe by the host
- The view talks to the host over `postMessage` (`ui/*`): it can call tools, update model context, and send messages
- Always return a **text fallback** for hosts that don't render Apps

```
McpApps.tool(  
  McpTool("weather").handle(ZIO.succeed(McpApps.result(fallbackText))),  
  McpAppsToolMeta(uri),    // uri: McpUiUri = "ui://weather/dashboard"  
)
```

Demo: MCP Apps in `claude.ai`

Tasks: long-running work without a long-lived connection

Moved out of core into `io.modelcontextprotocol/tasks` and redesigned (SEP-2663).

```
tools/call deep_research    →  resultType: "task"  
                             task: { taskId, status: "working", pollIntervalMs: 500 }  
tasks/get { taskId }       →  status: working ... completed + result  
tasks/update { taskId }    →  answer a task's inputRequests (status: input_required)  
tasks/cancel { taskId }    →  cooperative cancellation
```

- The **server** decides per request whether to return a task. Clients opt in once via capabilities
- `tasks/result` (blocking) and `tasks/list` are gone. Use polling, or `notifications/tasks` on `subscriptions/listen`
- Task ids are durable handles that survive reconnects and replicas (if the store is shared)

Skills over MCP

Agent Skills (`SKILL.md` + files) served by the MCP server that they describe.

```
{"skills": [{
  "uri": "skill://mcp2-demo/flight-booking/SKILL.md",
  "frontmatter": {"name": "flight-booking", "description": "How to book a flight with this server's tools..."},
  "resources": [
    {"uri": "skill://mcp2-demo/flight-booking/SKILL.md", "digest": "sha256:c223...", "size": 484},
    {"uri": "skill://mcp2-demo/flight-booking/references/airports.md", "digest": "sha256:1114...", "size": 128}]]}]}
```

- `skills/list` , `skills/get` , plus optional `resources/directory/read`
- Content is read with ordinary `resources/read` . Digests let hosts verify before loading
- Reading a skill doesn't activate it. The host decides, and may ask the user

Skills from SkillsJars

```
libraryDependencies +=  
  "com.skillsjars" % "anthropics__skills__brand-guidelines" % "2026_02_25-3d59511"
```

```
for skills <- McpSkillsJars.load // META-INF/skills/** on the classpath  
yield McpServer("mcp2-zio-server", "1.0.0")  
  .tool(Tools.bookFlight)  
  .withExtensions(skills.extensions) // skills/list, skills/get, directory/read  
  .resourceSource(skills.resources) // resources/read
```

Your own skills live in `src/main/resources/META-INF/skills/...` and are versioned and shipped with the server.

Demo: Tasks & Skills

```
cd scala-demo  
./sbt "runMain mcp2.Main"      # dual-era server on :8081  
  
npx @modelcontextprotocol/inspector
```

Wrapping up

Migration checklist for server developers

- Run **stateless**. Replace session state with **explicit handles**
- Stop relying on Sampling, Roots, and Logging (deprecated). Use tool params, direct LLM calls, and OTel
- Auth: validate **audience**, and use an AS with **CIMD**
- Move long-running work to **Tasks**. Ship **Skills** with your server. Add **Apps** where UI helps
- Serve **2025-11-25** **and** **2026-07-28** during the transition

Implementations used today

Project	Version / notes
Spring AI	2.1.0-M1 (MCP Java SDK 2.0.0 , stateless 2025-11-25 ; 2026-07-28 in progress)
MCP Security	org.springaicomunity:mcp-*-security:0.1.14
Tachyon MCP	tachyonmcp.dev (Java, Virtual Threads, 2025-11-25 and 2026-07-28)
zio-http-mcp	com.jamesward::zio-http-mcp:0.8.3 (dual-era, MRTR, Tasks, Skills, Apps)
CIMD for localhost	cimd.now
Spec	modelcontextprotocol.io/specification/2026-07-28

Thanks!

James Ward

Code: github.com/jamesward/mcp2